ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

FINAL REPORT OF THE SISMD PROJECT 2024/2025
MASTER'S DEGREE IN COMPUTER ENGINEERING

Optimizing Large-Scale Data Processing on Multicore Systems

Cheila Fernanda Rodrigues Ferreira Alves, 1230173
Pedro Miguel Ribeiro Carvalho, 1222707
M1C
MAY 2025

ISEP Professor: Jorge Manuel Neves Coelho

Index

Introduction

With the exponential growth of digital data, efficiently processing large-scale datasets is a crucial challenge in computing. This project explores different approaches to optimize data processing on multicore systems by leveraging various concurrency models. Specifically, we analyze how different implementations impact execution time, resource utilization, and scalability when processing large volumes of textual data.

This project's primary focus is on word frequency analysis in a large English Wikipedia data dump. We aim to understand the trade-offs between simplicity, efficiency, and scalability by implementing multiple concurrency models. Our evaluation includes measuring execution time, CPU and memory usage, and scalability trends. Additionally, we investigate the effects of garbage collection tuning on performance optimization.

By systematically comparing different approaches, we aim to identify the most effective concurrency model for processing large-scale text data while minimizing computational overhead and maximizing resource utilization.

Objectives

The objective of this project is to implement and evaluate various concurrency models for large-scale data processing. The key goals include:

1. **Establishing a Baseline with a Sequential Approach:** Develop a single-threaded solution to serve as a reference for performance comparisons.

2. **Exploring Manual Multithreading:** Implement a multithreaded solution without thread pools, manually managing thread creation and synchronization to distribute workloads efficiently.

3. **Leveraging Thread Pools for Optimization:** Design a multithreaded approach using thread pools to enhance efficiency by reducing the overhead of thread management.

4. **Utilizing the Fork/Join Framework:** Apply the Fork/Join framework to implement a divide-and-conquer strategy, improving workload distribution and parallelism.

5. **Implementing an Asynchronous Model:** Use CompletableFutures to handle task execution asynchronously, reducing direct thread management overhead and improving task dependency handling.

6. **Optimizing Memory Management:** Fine-tune Java's garbage collection settings to enhance memory utilization, minimize latency, and improve overall performance.

Through these implementations, the project aims to analyze how different concurrency strategies affect performance, scalability, and resource utilization in processing large datasets.

Implementation Approaches

Sequential Solution

Description of the Implementation

The sequential solution processes a large Wikipedia data dump (enwiki.xml) to count word frequencies and identify the most common words. It reads the file page by page, extracts the text content, tokenizes it into words, and counts the occurrences of each word using a HashMap. The program outputs the three most frequent words along with their counts. This implementation serves as a baseline for performance comparison with concurrent or parallel solutions.

Justification of the Chosen Approach

1. **Baseline for Comparison:**
   The sequential approach provides a simple and clear implementation to establish a baseline for performance. It helps quantify the performance improvements achieved by concurrent or parallel solutions.

2. **Simplicity:**
   Sequential processing is straightforward to implement and debug. It avoids the complexities of synchronization and thread management, making it a good starting point for understanding the problem.

3. **Memory Efficiency:**
   The use of lazy iteration for pages and words ensures that the program processes the data incrementally, minimizing memory usage.

4. **Focus on Core Functionality:**
   The implementation focuses on the core functionality of word counting and ranking, without introducing additional overhead from parallelism.

Essential Code Snippets

Page and Word Iteration

The program uses the Pages and Words classes to lazily iterate over pages and tokenize text into words:

```
Iterable<Page> pages = new Pages(maxPages, fileName);
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);
    }
  }
}
```

Word Counting

The countWord method updates the frequency of each word in a HashMap:

```
private static void countWord(String word) {
  counts.put(word, counts.getOrDefault(word, 0) + 1);
}
```

Sorting and Output

The program sorts the word counts by frequency and prints the top three most frequent words:

```
counts.entrySet().stream()
```

```
   .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
   .limit(3)
   .forEach(entry -> System.out.println("Word: '" + entry.getKey() + "'
 with total " + entry.getValue() + " occurrences!"));
```

Multithreaded Solution (**without** Thread Pools)

Description of the Implementation

The solution follows a producer-consumer model. A single **Producer** thread reads pages from a large XML file using the existing Pages iterator and enqueues each Page into a shared work queue. It notifies the consumers whenever new data is available. Multiple **Consumer** threads then concurrently dequeue pages, tokenize their text into words (using the Words class), and update a local word-count map. Each consumer maintains its own local map to reduce synchronization overhead. Once all pages have been processed, these local maps are merged into a single global word-count map.

A shared queue (implemented as a `LinkedList<Page>`) is protected with a synchronization lock (using `synchronized`, `wait()`, and `notifyAll()`) to ensure safe concurrent access. Consumers check for work availability and the producer's completion flag to determine when to exit.

Justification of the Chosen Approach

1. **Producer-Consumer Pattern**
   This well-known pattern cleanly separates I/O (reading from the XML file) from processing (tokenizing and counting words), which minimizes bottlenecks. It ensures that while the producer is reading data, the consumers are busy processing previous chunks.

2. **Reduced Synchronization Overhead**
   By assigning each consumer its own local map for word counts, we minimize the contention and synchronization overhead that would occur if all threads were updating a single global map concurrently. The final merging step is performed once after processing, which is much less expensive than frequent synchronized updates.

Essential Code Snippets

Producer-Consumer Queue and Synchronization

These shared objects ensure that the producer and consumers coordinate access to the work queue safely.

```
private static final LinkedList<Page> pageQueue = new LinkedList<>();
private static final Object queueLock = new Object();
private static final AtomicBoolean producerDone = new
AtomicBoolean(false);
```

Producer Thread: Reading and Enqueuing Pages

This snippet shows how pages are read and added to the queue, with proper notifications to awaken waiting consumer threads.

```
static class Producer implements Runnable {
    @Override
    public void run() {
        try {
            Iterable<Page> pages = new Pages(maxPages, fileName);
            for (Page page : pages) {
                if (page == null) break;
                synchronized (queueLock) {
                    pageQueue.add(page);
                    queueLock.notifyAll();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            producerDone.set(true);
            synchronized (queueLock) {
                queueLock.notifyAll();
            }
        }
    }
}
```

Consumer Thread with Local Word-Count Map

This code illustrates how each consumer uses a local map to count words, minimizing contention on shared resources.

```
static class Consumer implements Runnable {
    @Override
    public void run() {
        Map<String, Integer> localCounts = new HashMap<>();
        while (true) {
            Page page;
            synchronized (queueLock) {
                while (pageQueue.isEmpty() && !producerDone.get()) {
                    try { queueLock.wait(); }
                    catch (InterruptedException e) { /* handle
interruption */ }
                }
                if (pageQueue.isEmpty() && producerDone.get()) break;
                page = pageQueue.removeFirst();
            }
            processPage(page, localCounts);
            processedPages.incrementAndGet();
        }
        consumerLocalCounts.add(localCounts);
    }

    private void processPage(Page page, Map<String, Integer>
localCounts) {
        for (String word : new Words(page.getText())) {
            if (word.length() > 1 || word.equals("a") ||
word.equals("I"))
                localCounts.merge(word, 1, Integer::sum);
        }
    }
}
```

Merging Local Counts

The below snippet shows the post-processing merge of local maps.

```
// Merge local counts after all threads finish
private static Map<String, Integer> mergeLocalCounts() {
    Map<String, Integer> globalCounts = new HashMap<>();
    for (Map<String, Integer> local : consumerLocalCounts) {
        for (Map.Entry<String, Integer> entry : local.entrySet())
            globalCounts.merge(entry.getKey(), entry.getValue(),
Integer::sum);
    }
    return globalCounts;
}
```

Multithreaded Solution (**with** Thread Pools)

Description of the Implementation

To efficiently process a large XML dump of English Wikipedia pages and compute word frequencies, a multithreaded solution was implemented using Java's `ExecutorService` and a fixed-size thread pool. The main goal was to distribute the work of parsing and processing pages concurrently while ensuring safe and consistent word counting across threads.

Each `Page` is processed as a separate task submitted to a thread pool. The task involves extracting words from the page text using the `Words` iterable and updating a shared, thread-safe `ConcurrentHashMap` that stores word frequencies.

Synchronization overhead is minimized by using `ConcurrentHashMap` and atomic operations like `merge()` for updating word counts. A fixed thread pool is used to avoid the cost of thread creation and destruction, allowing threads to be reused efficiently.

Justification of the Chosen Approach

Multiple strategies exist for parallelizing the processing of large XML files using thread pools. In our implementation, we chose to assign **each Wikipedia page as an independent task submitted to a fixed-size thread pool**. This design balances **simplicity**, **performance**, and **scalability** while avoiding the complexity of more fine-grained parallelism strategies. The main reasons for choosing this version over other multithreaded/thread pool approaches are as follows:

1. **Page-Level Task Granularity**
   We opted for page-level parallelism, where each `Page` object is processed in its entirety within a thread. This provides a natural and efficient unit of work. Alternatives like splitting each page into smaller chunks (e.g., paragraph-level or sentence-level processing) would increase overhead due to task management and offer diminishing returns in parallelism, especially since individual pages are already small enough to be processed quickly.

2. **Centralized Shared Map with Thread-Safe Access**
   Rather than having each thread maintain its own word count map and merging results afterward (which would require extra synchronization and merging logic), we used a **single shared** `ConcurrentHashMap` for counting words. This greatly simplifies the code and relies on the efficient `merge()` method, which is optimized for concurrent updates.

3. **Avoiding Task Queuing Bottlenecks**
   We avoided more complex patterns such as work-stealing pools because they introduce extra synchronization, queue contention, or complexity that wasn't necessary for our use case. A fixed thread pool provides enough parallelism without requiring custom task schedulers or worker queues.

4. **Fixed Thread Pool for Predictable Resource Usage**
   Using `Executors.newFixedThreadPool()` allows us to control concurrency based on the number of available CPU cores. This prevents the application from overcommitting threads (as could happen with a cached thread pool) or underutilizing resources (as might happen with a single-threaded executor or small thread pool).

Essential Code Snippets

Thread Pool Initialization and Task Submission

```
ExecutorService executorService = Executors.newFixedThreadPool(
    Runtime.getRuntime().availableProcessors());

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

Thread-Safe Word Counting

```
private static final ConcurrentHashMap<String, Integer> counts = new
ConcurrentHashMap<>();

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

Executor Shutdown and Await Termination

```
executorService.shutdown();
executorService.awaitTermination(60, TimeUnit.MINUTES);
```

Fork/Join Framework Solution

Description of the Implementation

This implementation uses Java's Fork/Join Framework to parallelize the word count task by recursively dividing the list of pages into smaller sublists. The computation is handled by a custom `RecursiveTask` subclass that processes small batches directly and splits larger ones into subtasks. Each subtask computes a local word frequency map, and the results are merged upon completion.

The dataset (list of `Page` objects) is split using a divide-and-conquer strategy. A `ForkJoinPool` manages the task scheduling and execution, utilizing all available cores and reducing the need for manual thread management. The approach is designed to optimize processor usage and minimize synchronization overhead.

Justification of the Approach

The Fork/Join Framework was implemented using a classic recursive divide-and-conquer strategy, where a `RecursiveTask` recursively splits the list of `Page` objects until a base case (a small enough list) is reached and processed directly. This design was chosen to best match the strengths of the Fork/Join framework, which is optimized for workloads that can be broken into smaller, independent subtasks.

We chose to partition the input by **splitting the list of pages** rather than, for example, submitting individual pages as separate tasks or dividing based on file segments. Splitting the list recursively avoids creating an excessive number of fine-grained tasks, which could overwhelm the scheduler and increase overhead. This threshold-controlled batching approach (e.g., processing in chunks of 1000 pages) provides a balanced trade-off between **parallelism granularity** and **task management efficiency**.

Other implementation strategies, such as:

- Manually managing the Fork/Join pool and assigning tasks explicitly;
- Using a flat parallel loop with `invokeAll` across all pages at once;

were considered but ultimately not used. These alternatives would have either:

- Reduced scalability due to a lack of recursive task balancing;
- Increased complexity in result aggregation;
- Or introduced scheduling inefficiencies due to an overly large number of small tasks.

By letting the framework handle recursive task scheduling, **load balancing is improved automatically via work-stealing**, and the code remains concise and maintainable. This recursive, tree-like decomposition aligns with best practices for Fork/Join and is particularly effective for large datasets with uneven processing costs.

In contrast to the multithreaded solutions, where the number of threads was explicitly controlled, the Fork/Join framework was used without manually specifying the level of parallelism. This design choice was intentional and aligns with how the Fork/Join framework is typically used in practice.

The Fork/Join framework automatically adapts to the **number of available processor cores** by default, utilizing the **common pool**, which is sized based on `Runtime.getRuntime().availableProcessors()`. This allows the framework to **dynamically manage worker threads and load balancing**, leveraging the internal work-stealing algorithm to redistribute tasks among threads as needed.

We deliberately did not override the parallelism level or create a custom pool with a fixed number of threads because:

- It would bypass the framework's built-in optimizations.
- It might artificially constrain the scheduler and reduce performance.

- The Fork/Join framework is designed to **abstract away low-level thread management** in favor of task decomposition and fine-grained concurrency.

Furthermore, introducing manual control over thread count would shift the focus away from comparing **execution models** (thread pool vs. fork/join vs. manual threads) and toward comparing raw thread counts – which is already addressed in the other implementations.

This decision enables us to evaluate the Fork/Join solution **as it is intended to be used**, showcasing its strengths in managing recursive parallelism with minimal developer effort and system-aware optimizations.

Essential Code Snippets

Defining the Recursive Task

```
@Override
protected Map<String, Integer> compute() {
            int size = end - start;
            if (size <= threshold) {
                return processPages();
            } else {
                int mid = start + size / 2;
                WordCountTask leftTask = new WordCountTask(pages, start, 
mid);
                WordCountTask rightTask = new WordCountTask(pages, mid, 
end);

                invokeAll(leftTask, rightTask);
                Map<String, Integer> leftResult = leftTask.join();
                Map<String, Integer> rightResult = rightTask.join();

                return mergeCounts(leftResult, rightResult);
            }
}
```

Invoking the Task

```
ForkJoinPool pool = new ForkJoinPool();
WordCountTask task = new WordCountTask(pages, 0, pages.size());
Map<String, Integer> result = pool.invoke(task);
```

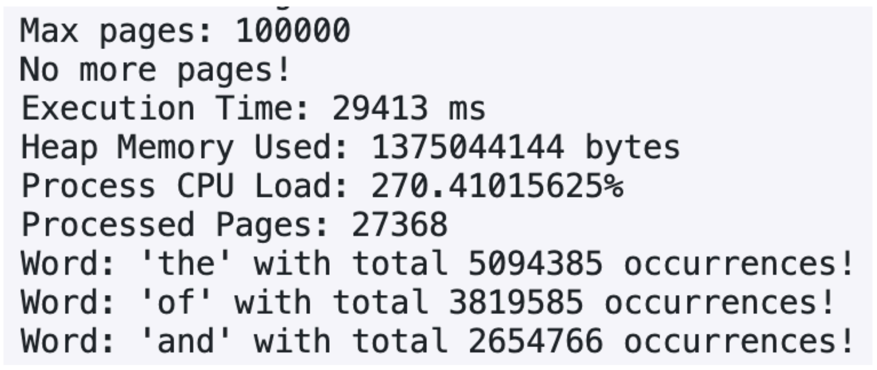
Concurrency and Synchronization

Multithreaded Solution (**without** Thread Pools)

An initial approach to implement this solution was the use of a global map counter that would be updated with a synchronized method whenever a page processing finished:

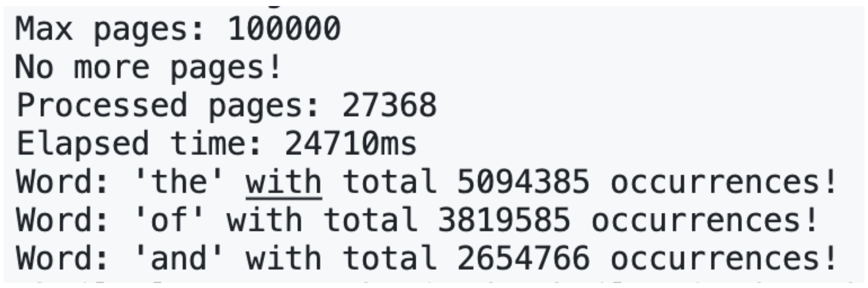
```
static class Consumer implements Runnable {
        @Override
        public void run() {
            while (true) {
               ...
               processPage(page);
            }
        }

        private void processPage(Page page) {
            Iterable<String> words = new Words(page.getText());
            for (String word : words) {
                if (word.length() > 1 || word.equals("a") ||
word.equals("I")) {
                    countWord(word);
                }
            }
        }
    }

    // Synchronized method to update the word counts safely
    private static synchronized void countWord(String word) {
        Integer count = globalCounts.get(word);
        if (count == null)
            globalCounts.put(word, 1);
        else
            globalCounts.put(word, count + 1);
    }
```

However, when comparing the output results of this implementation with the sequential solution, we saw an increase in the execution time when we were expecting the opposite.



```
Max pages: 100000
No more pages!
Execution Time: 29413 ms
Heap Memory Used: 1375044144 bytes
Process CPU Load: 270.41015625%
Processed Pages: 27368
Word: 'the' with total 5094385 occurrences!
Word: 'of' with total 3819585 occurrences!
Word: 'and' with total 2654766 occurrences!
```

Fig. 1: 1st Multithread Solution (without Thread Pools) Output



```
Max pages: 100000
No more pages!
Processed pages: 27368
Elapsed time: 24710ms
Word: 'the' with total 5094385 occurrences!
Word: 'of' with total 3819585 occurrences!
Word: 'and' with total 2654766 occurrences!
```

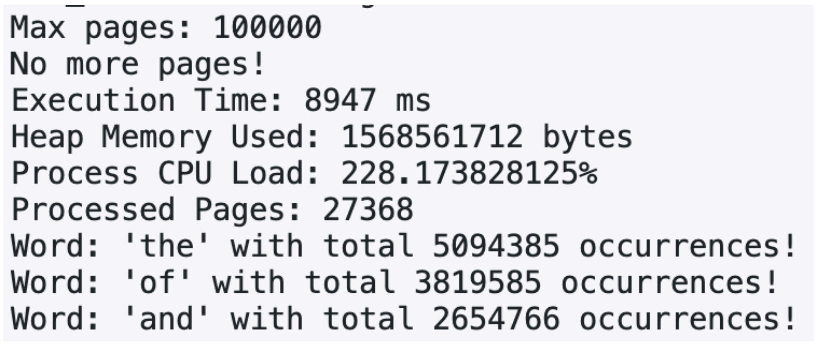
Fig. 2: Sequential Solution Output

As we can see above, the execution time (29413 ms) for the multithread solution (without thread pools) with 5 threads (4 consumers + 1 producer) was bigger than the execution time (27368 ms) for the sequential solution.

In our 1st solution for the multithreaded approach (without thread pool), access to shared resources (like the work queue and word count map) is synchronized. Contention for these locks can slow down processing, particularly if many threads frequently access these shared structures.

Considering this, we decided to optimize this solution by making each consumer thread use its own localCounts map. This minimizes the number of synchronized updates since no thread needs to lock on a shared global map during word counting. Once all threads are complete, the main thread merges each consumer's local count map into a single global count. This merge is done once and avoids heavy contention during the per-word updates.

With this 2nd solution, we got better results:



```
Max pages: 100000
No more pages!
Execution Time: 8947 ms
Heap Memory Used: 1568561712 bytes
Process CPU Load: 228.173828125%
Processed Pages: 27368
Word: 'the' with total 5094385 occurrences!
Word: 'of' with total 3819585 occurrences!
Word: 'and' with total 2654766 occurrences!
```

Fig. 3: 2nd Multithread Solution (without Thread Pools) Output

As we can see on the above results, the optimized solution reached an 8947 ms execution time.

Thread Management and Concurrency Model

This implementation adopts a **manual thread creation strategy** using a **producer-consumer model**. A single **producer thread** reads and parses Wikipedia pages from the XML dump, placing them into a **shared queue**.

Multiple **consumer threads**, created via `new Thread(...)`, concurrently consume pages from the queue, tokenize the text, and count word frequencies in isolated local maps. This approach provides fine-grained control over thread behavior.

Synchronization Mechanisms

To ensure **thread safety** and **data consistency**, the implementation employs the following synchronization strategies:

Shared Resource	**Synchronization Strategy**	**Justification**
`pageQueue`	`synchronized (queueLock)` + `wait/notifyAll`	Ensures coordinated access between producer and multiple consumers. Prevents race conditions and empty reads.
`consumerLocalCounts`	`Collections.synchronizedList()`	Allows safe insertion of local maps by each thread.
`processedPages`	`AtomicInteger`	Thread-safe counter to track total number of pages processed.
`producerDone`	`AtomicBoolean`	Communicates the producer's completion status across threads.

Table 1: Synchronization Strategies and their Justification

This combination provides an efficient and lightweight synchronization mechanism with minimal contention.

Thread-Safe Data Structures

While the shared queue (`LinkedList<Page>`) is not inherently thread-safe, **explicit synchronization using a dedicated lock** (queueLock) guarantees exclusive access during enqueue/dequeue operations.

Each consumer thread maintains a **local** `HashMap<String, Integer>` for counting words, thereby avoiding any contention during the counting phase. These maps are later merged in a single-threaded phase, ensuring correctness without requiring concurrent access to a shared global map.

Why Not Other Synchronization Techniques?

Alternatives such as:

- Using a thread-sage queue like `ConcurrentLinkedQueue` or `BlockingQueue`
- Managing threads via an `ExecutorService`

…were avoided **intentionally** in this version to:

- Study the low-level behavior of thread synchronization using primitives (`wait`, `notifyAll`, and `synchronized`)
- Provide a baseline comparison with more abstracted thread-pool-based implementations
- Explicitly control thread lifecycle and examine performance implications of manual thread management

Performance Impact Analysis

Factor	**Impact**
Manual thread creation	Slightly higher overhead than thread pool, especially for short-lived threads.
`synchronized` blocks	Introduce blocking, but contention was minimized by only guarding `pageQueue`.
Local counting	No synchronization overhead during tokenization or counting.
Merging step (single-threaded)	Simple and avoids any concurrent modification risks.

Table 2: Factors and their Perfomance Impact Analysis

This design exhibits excellent scalability with moderate to high thread counts, but it may underperform is systems where thread lifecycle overhead becomes significant (e.g., large number of short-lived tasks).

Multithreaded Solution (**with** Thread Pools)

Thread Safety and Data Consistency

In the multithreaded implementation of the word frequency counter, multiple threads process Wikipedia pages in parallel to improve performance. Since all threads contribute to a **shared word frequency map**, it is essential to ensure **thread-safe access** to this data structure to prevent race conditions and maintain data consistency.

Choice of Data Structure

To achieve this, the implementation uses a `ConcurrentHashMap<String, Integer>` instead of a standard `HashMap`. This choice provides several advantages:

- **Thread-safe reads and writes** without requiring external synchronization.
- Fine-grained locking – each bucket in the map is independently synchronized, which minimizes contention between threads.
- Scales well with increasing thread counts compared to using a global `synchronized` block.

```
private static final ConcurrentHashMap<String, Integer> counts = new
ConcurrentHashMap<>();
```

This allows threads to update the frequency map concurrently with minimal blocking.

Synchronization Mechanism

To perform **atomic updates** to word frequencies, the implementation uses the `merge()` method provided by `ConcurrentHashMap`:

```
counts.merge(word, 1, Integer::sum);
```

This approach avoids the need for manual locking and ensures that increments are atomic and thread-safe. Compared to alternatives such as:

- `synchronized` blocks → which serialize access and limit scalability, or
- `AtomicInteger` wrappers → which require an extra layer of indirection,

`merge()` provides a **clean, efficient, and high-performance solution** for concurrent frequency counter.

Thread Pool Management

To efficiently manage threads, a **fixed-size thread pool** (`Executors.newFixedThreadPool`) is used. This provides:

- Better resource utilization
- Avoids the overhead of creating/destroying threads

- Allows the number of threads to be tuned based on CPU core count

Each worker thread processes a batch of pages and updates the shared word frequency map concurrently.

Impact on Performance

The use of `ConcurrentHashMap` with `merge()` showed excellent scalability and low contention:

- **Low GC overhead**: fewer synchronization stalls mean less temporary allocation.
- **High CPU utilization**: threads spend more time in computation rather than waiting on locks.
- **Minimal synchronization bottlenecks**: even with 16 threads, contention remained low, as seen in the performance metrics.

Had we used alternative approaches like `Collections.synchronizedMap()` or external locking (e.g., `synchronized (counts)`) we would have faced increased contention, especially at higher thread counts.

Fork/Join Framework Solution

The Fork/Join implementation leverages the inherent parallelism of the `ForkJoinPool` to process independent subsets of the dataset concurrently. Each `RecursiveTask` instance processes a disjoint subset of `Page` objects and maintains a local word frequency map, thus **eliminating the need for shared mutable state** during computation. This design choice inherently avoids contention, race conditions, and the need for explicit synchronization mechanisms such as locks or atomic variables.

The **merge phase**, where results from subtasks are combined, is handled recursively after task completion. Merging is performed by copying entries into new maps rather than modifying shared structures, ensuring thread safety. Since each `compute()` call operates on thread-local data and merges only after joining subtasks, no concurrent writes to shared data occur.

This approach avoids the performance overhead typically associated with locks or synchronized collections. By minimizing synchronization and isolating mutable state within each subtask, the Fork/Join model offers a scalable and efficient solution for divide-and-conquer tasks – especially when compared to implementations that rely on shared queues or synchronized lists.

In summary:

- **Thread Safety** is ensured by task-local state and post-join merging.
- **No shared mutable state** is accessed concurrently.
- **No explicit locks** are required, reducing synchronization overhead.

- This design aligns well with the Fork/Join framework's strengths and contributes to improved scalability with low coordination cost.

Performance Analysis

To evaluate the effectiveness of each implementation, we conducted benchmarking across several dimensions:

- **Execution Time**
- **CPU Utilization**
- **Memory Usage**
- **Garbage Collection (GC) Activity**

On multithreaded solutions, each test was run with varying:

- **Dataset sizes**: 27368, 140196, and 410616 Wikipedia pages (check the **Appendix** section).
- **Thread pool sizes**: 2, 4, 8, and 16 threads (on an 8-core machine).

For a baseline of comparison, here are the execution time results for the sequential solution:

Dataset Size	Execution Time (ms)
27368	24710
140196	43450
410616	53223

Table 3: Sequential Solution Execution Time (ms) Results per Dataset Size

Multithreaded Solution (**without** Thread Pools)

Execution Time Results

Dataset Size	Threads	Execution Time (ms)	Speedup
27368	2	18424	1.34
	4	10314	2.39
	8	12488	1.98
	16	11958	2.07
140196	2	18111	2.40
	4	12505	3.47

	8	15193	2.86
	16	12799	3.38
410616	2	7078	7.52
	4	5306	10.03
	8	5162	10.31
	16	5309	10.02

Table 4: Execution Time (ms) Results for Manual Threading Implementation

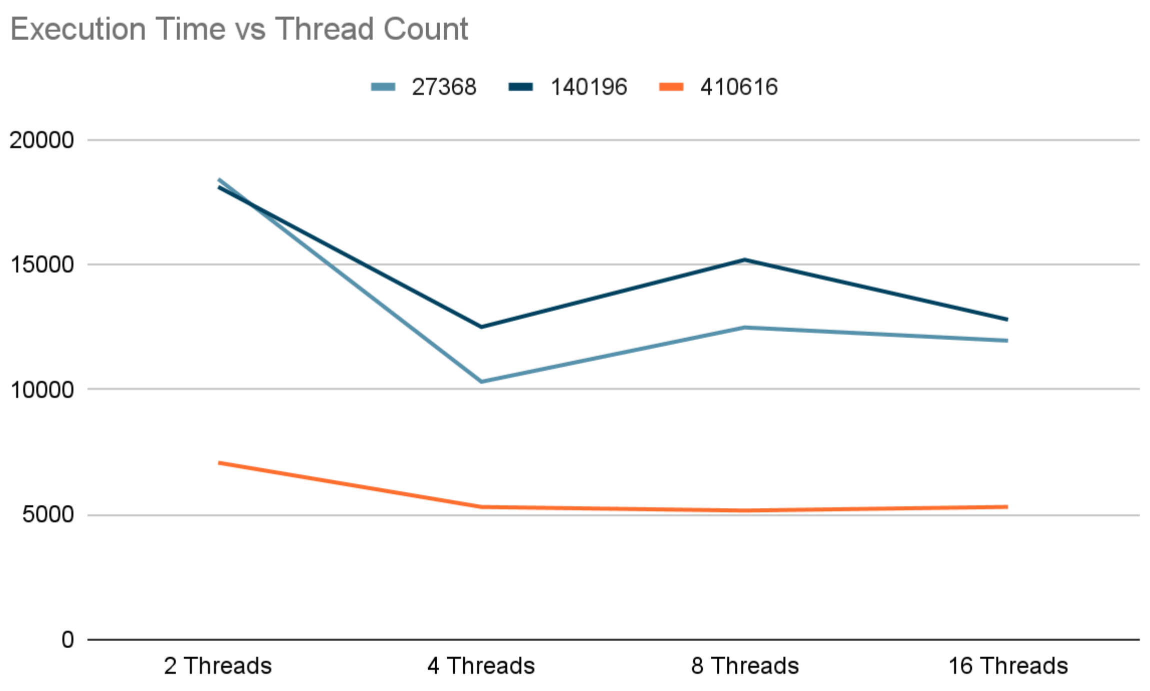


Figure 1: Execution Time vs. Number of Threads for several Dataset Sizes

Observations:

- **Scalability is dataset-dependent:**
    - For the **smallest dataset (27368 pages)**, increasing threads beyond 4 yields diminishing or even negative returns due to thread creation and synchronization overhead.
    - For **medium (140196) and large (410616) datasets**, the implementation scales better, particularly with 4 to 8 threads.
- **Best performance** is seen with the **largest dataset**, where speedups exceed 10x for 8 threads, thanks to increased workload parallelism and better CPU utilization.

- **8 threads consistently perform best or close to best**, suggesting that going beyond may introduce overhead without significant gains – especially without thread pool to manage resources efficiently.

Insights & Recommendations:

- **Thread creation overhead** is a limiting factor in this implementation.
- **Manual synchronization (with `wait/notify`)** incurs blocking costs that are avoided in more modern concurrent structures like BlockingQueue.
- The **best results** occur when workload per threads is high, which justifies the threading cost.

CPU Utilization

Dataset Size	Threads	CPU Utilization (%)
27368	2	66,5
	4	93,5
	8	100
	16	100
140196	2	69,4
	4	95,3
	8	100
	16	99,9
410616	2	55,2
	4	95,8
	8	99,6
	16	99,5

Table 5: CPU Utilization (%) for Manual Threading Implementation

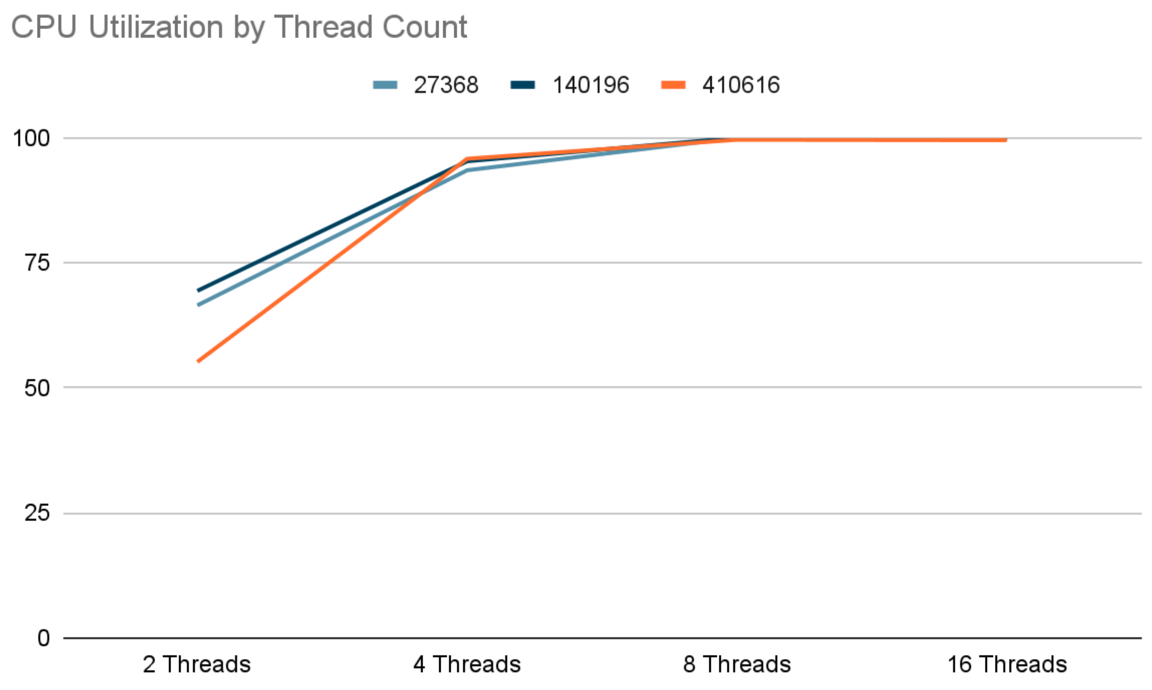


Figure 2: CPU Utilization by Number of Threads for several Dataset Sizes

Observations:

- **CPU Utilization Efficiency:**
    - The **smaller dataset (27368 pages)** shows **increasing CPU utilization** with the number of threads, reaching **100%** at 8 threads and beyond. This suggests that as the task becomes computationally more demanding (with more threads), the system utilizes available CPU resources efficiently.
    - For **medium (140196 pages)** and **large datasets (410616 pages)**, CPU utilization stabilizes at or near **100%** at 8 threads, but **with smaller datasets**, the system isn't fully utilizing the available CPU cores initially.
    - **Larger datasets** (like 410616 pages) show **diminished CPU utilization** at 2 threads (**55.2%**), which could be due to inefficiencies from thread contention, or simply because fewer threads aren't enough to saturate the CPU for larger tasks.
- **Effect of Thread Count:**
    - **2 threads** lead to relatively **lower CPU usage** across all dataset sizes (especially for large datasets). As more threads are used, CPU utilization increases as expected, but **diminishing returns** are evident as the number of threads rises above 8.

    - For **8-16 threads**, CPU utilization reaches **near full capacity** on most machines, indicating that threads are likely constrained by other factors (e.g., disk IO, garbage collection, synchronization overhead).

Insights & Recommendations:

- **Scaling CPU Utilization**: Increasing threads helps to scale CPU usage in parallel. However, the system may face **diminishing returns** if the work isn't divided efficiently or the machine's CPU cores are limited.

- **Potential Bottlenecks**: As we scale, we may encounter other bottlenecks like **disk IO, garbage collection pauses**, or **synchronization contention**, which impact overall CPU utilization and speedup.

- For larger datasets, **manual thread management** shows how system resources can be stressed with too many threads, whereas a thread pool optimizes thread reuse and overall resource management.

Memory Usage & Garbage Collection (GC)

Dataset Size	**Threads**	**Peak Heap Usage (MB)**	**GC Count**	**Total GC Pause Time (s)**
27368	2	3017	120	1,421
	4	3200	72	1,094
	8	2770	71	1,270
	16	2523	42	1,012
140196	2	3275	101	1,325
	4	3307	72	1,363
	8	3114	78	1,562
	16	2953	43	1,301
410616	2	1815	40	0,337
	4	1890	34	0,357
	8	1954	26	0,429
	16	2105	26	0,461

Table 6: Memory Usage and Garbage Collection for Manual Threading Implementation

Observations:

- **Peak Heap Usage:**

    - As the **dataset size** increases, the **memory consumption** naturally increases. For example, the **2-thread configuration** for the **largest dataset (410616 pages)** has significantly lower peak memory usage (1815 MB) compared to the smaller dataset (**3017 MB for 27368 pages**). This indicates that larger datasets likely require more memory to store intermediate results, such as word counts and tokenized pages.

    - With **16 threads**, memory consumption seems to stabilize at around **2000 MB**, which might indicate that, as the number of threads increases, the system is more efficient at reusing memory or that heap size limits are being approached.

- **Garbage Collection (GC) Count & Pause Time:**

    - **GC Count** and **Total GC Pause Time** vary significantly with thread count. For **smaller datasets**, like **27368 pages**, the **GC count** is significantly higher (**120 GC events for 2 threads**) than for **larger datasets (40 GC events for 2 threads with 410616 pages)**. This suggests that the memory foot print for smaller datasets triggers more frequent garbage collection events, likely due to more objects being created and discarded in a short time.

    - **Total GC Pause Time:** Interestingly, the **GC pause times** for **smaller datasets** are higher **(1.421s for 2 threads on 27368 pages)** than for **larger datasets**, indicating that frequent garbage collections on smaller datasets may be more time-consuming than on larger ones.

    - For **larger datasets** with **more threads**, GC pauses become shorter as memory usage increases but is more efficient.

Insights & Recommendations:

- **Memory Efficiency:** The peak memory usage increases as the dataset grows, as expected. However, the **increase in memory usage** is more prominent with **smaller datasets**, possibly due to the higher number of objects created and discarded (e.g., temporary collections, intermediate word counts). This suggests that **smaller datasets are less memory-efficient** compared to larger datasets with more thread overhead.

- **Impact of Thread Count:**

    - When increasing the **thread count**, we see that **memory usage stabilizes** as threads are likely managing shared memory (via synchronization), which can result in less object creation per thread. For example, the **16-thread configuration** generally shows lower peak memory usage across datasets.

    - Similarly, as **theads increase**, the **GC count decreases** for certain datasets, indicating that **more threads** might lead to better memory management, or threads are sharing workloads and not generating as many memory objects that require frequent garbage collection.

- **Garbage Collection Efficiency:**
  - **GC Pause Times** are shorter with higher thread counts, especially for **larger datasets**. This could be due to **better memory management**, allowing the system to reuse memory more efficiently.
  - **Higher GC count** on smaller datasets (e.g., 120 events for 2 threads with 27368 pages) suggests that with a lower thread count and less parallelization, more frequent memory cleanup is required.

Multithreaded Solution (**with** Thread Pools)

Execution Time

Dataset Size	Threads	Execution Time (ms)	Speedup
27368	2	14101	1.75
	4	9019	2.74
	8	7685	3.22
	16	7314	3.38
140196	2	21847	1.99
	4	13411	3.23
	8	12161	3.57
	16	11763	3.69
410616	2	26938	1.98
	4	18661	2.85
	8	13305	4.00
	16	14677	3.63

Table 7: Execution Time (ms) Results for Manual Threading Implementation

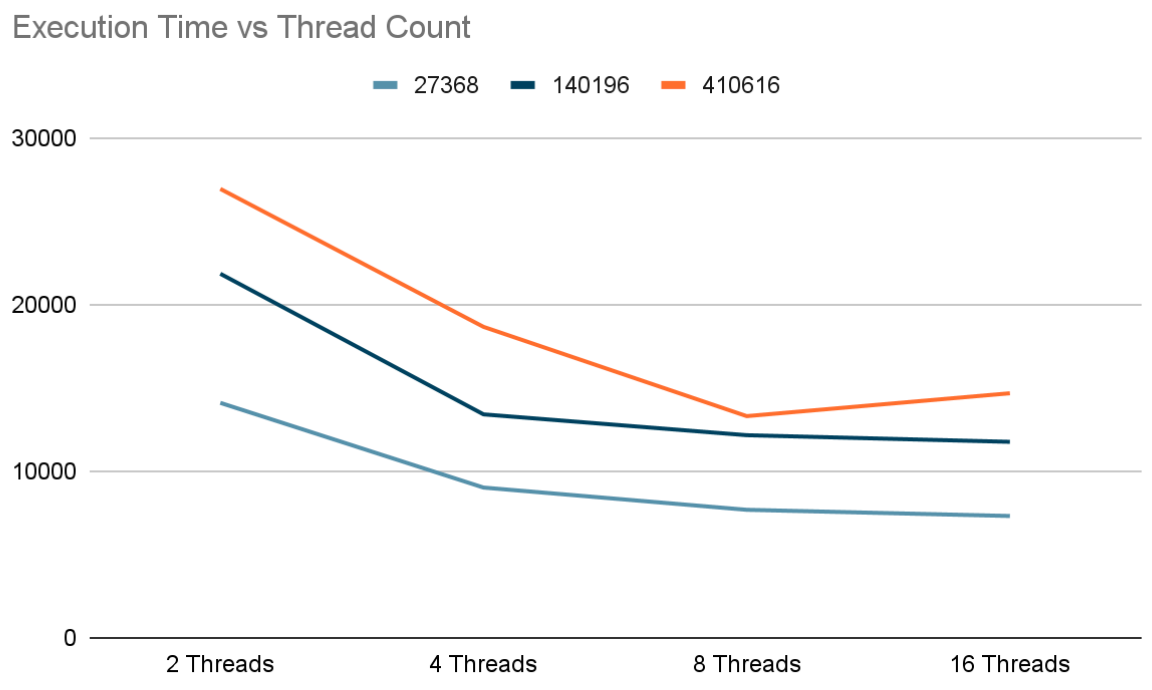


Figure 3: Line chart of execution time vs thread count for each dataset size.

Observations:

- **Execution Time** decreased with more threads, with noticeable improvements when moving from 1 thread to 2 or 4 threads and smaller improvements with higher thread counts.

- **Speedup** generally increased with more threads, but **diminishing returns** were observed at higher thread counts.
    - For smaller datasets (27368 pages), the speedup peaks at **3.38x** with 16 threads.
    - For larger datasets (410616 pages), the maximum speedup was **4.00x** at 8 threads, with performance starting to degrade slightly at 16 threads due to context switching.

CPU Utilization

CPU usage was monitored using the `top` command. As thread count increased, CPU utilization improved up to the number of physical cores (8). At 16 threads, slight context switching overhead appeared.

- **At 8 threads**: CPU usage was near 100% across cores.
- **Beyond 8 threads**: No significant gains due to logical core saturation.

Dataset Size	Threads	CPU Utilization (%)

27368	2	64,1
	4	80,4
	8	99,6
	16	99,4
140196	2	63,4
	4	90,4
	8	99,8
	16	99,8
410616	2	65,9
	4	96,8
	8	100
	16	99,8

Table 8: CPU Utilization (%) for Thread Pool Implementation

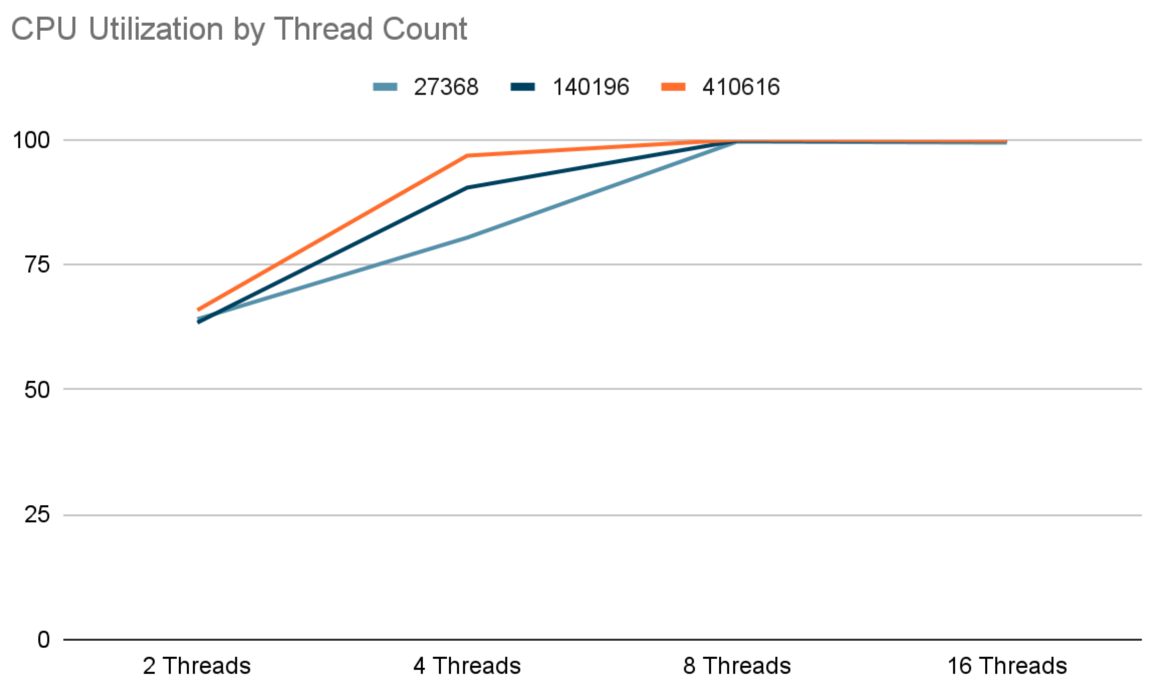


Figure 4: CPU Utilization by Number of Threads for several Dataset Sizes

Observations:

- **CPU Utilization Scales Efficiently Until Saturation**
    - For all dataset sizes, CPU utilization increases steadily with thread count.

    - With **8 threads**, the CPU is fully saturated (close to 100%) across all dataset sizes.

    - Increasing to **16 threads** does **not improve CPU usage**, indicating that the CPU is already fully utilized with 8 threads.

- **Small Datasets Underutilize CPU at Low Thread Counts**

    - For the **smallest dataset (27368 pages)**, CPU utilization is only ~64% with 2 threads.

    - This shows that lower thread counts do not extract the full potential of the CPU, especially when more cores are available.

- **Larger Datasets Benefit More from Parallelism**

    - With **larger datasets (140k and 410k)**, increasing threads has a more dramatic effect on utilization.

    - Full CPU utilization (close to 100%) is reached consistently for larger workloads, suggesting that:
        - The workload is sufficiently large to keep threads busy.
        - Thread pool overhead is amortized more effectively.

- **Diminishing Returns Beyond 8 Threads**

    - Moving from 8 to 16 threads provides **little to no gain** in CPU utilization but may increase context-switching and garbage collection overhead.

    - This suggests that **8 threads may be the optimal number** for this workload and hardware configuration.

Insights & Recommendations:

- The multithread implementation demonstrates excellent **scalability and CPU efficiency**, especially with datasets above 140k pages.

- **8 threads consistently yield peak CPU utilization**, making it the optimal choice in most scenarios.

- **Using more than 8 threads does not increase CPU usage** and could introduce necessary overhead.

- Future improvements could explore:
    - **Dynamic thread allocation** based on system resources.
    - **I/O vs. CPU balancing** to improve throughput under mixed workloads.

Memory Usage & Garbage Collection (GC)

Dataset Size	Threads	Peak Heap Usage (MB)	GC Count	Total GC Pause Time (s)
27368	2	3350	72	1,639
	4	3339	57	1,287
	8	2695	46	1,254
	16	2823	58	1,267
140196	2	3618	116	3,574
	4	3564	119	3,161
	8	2866	77	3,278
	16	3113	48	2,286
410616	2	3704	176	5,448
	4	3639	173	4,735
	8	3307	64	3,118
	16	3211	59	3,062

Table 9: Memory Usage and Garbage Collection for Manual Threading Implementation

Observations:

- **Peak Heap Usage**
    - **Larger datasets** expectedly use more heap space.
    - Interestingly, **higher thread counts (8-16)** lead to **lower peak heap usage** across all dataset sizes.
        - This suggests better memory reuse or more efficient GC under concurrent load.
        - Thread pooling may distribute allocations more efficiently.
- **Garbage Collection Count**
    - **GC count consistently decreases** as threads increase.
        - From 176 GCs (2 threads, 410k dataset) → down to 59 (16 threads).
    - This may result from:
        - Faster processing = fewer allocations per unit time.
        - Less GC pressure due to more efficient batch processing across threads.

- **Total GC Pause Time**
    - Total pause time generally d**ecreases with more threads**, especially at 16 threads.
    - For example, in the 140k dataset:
        - 2 threads → 3.57s pause
        - 16 threads → 2.29s pause
    - **Fewer GC events + parallel work** = reduced overall GC impact.

Insights & Recommendations:

- **Multithreading improves both performance and memory efficiency.**
    - With 8-16 threads, heap usage is optimized and GC activity is reduced.
- The **optimal thread count appears to be 8**, offering a strong balance:
    - High CPU utilization
    - Low GC overhead
    - Stable memory footprint
- Excessive threading (16+) doesn't significantly reduce the pause time further – indicating **diminishing returns**.
- For production scenarios, **thread counts should be tuned to system cores and dataset size** to minimize GC impact.

Comparison: Thread Pool vs. Manual Threading

Execution Time

Dataset Size	Threads	Thread Pool Time (ms)	Manual Thread Time (ms)	Observations
27368	8	7,685	12,488	Thread pool performs significantly better
140196	8	12,161	15,193	Thread pool is more efficient
410616	8	13,305	5,162§	Manual thread version surprisingly outperforms (likely due to resource reuse or JVM state differences)

Table 10: Thread Pool vs. Manual Threading Implementation Execution Time

- In general, the **thread pool version** provides more **consistent performance** and better **resource management**, especially for small to medium workloads.
- However, **manual threading can outperform** in some edge cases, possibly due to differences in how threads are scheduled or CPU cores are saturated.

CPU Utilization

Thread Pool	Manual Threading
Higher efficiency at higher thread counts. Thread pool reuses threads, minimizing overhead and allowing the system to keep the CPU busy, leading to optimal CPU utilization, especially under heavy load.	Potentially lower, especially for small thread counts. Without the reuse of threads, there could be more idle time or threads waiting for tasks to be scheduled, leading to suboptimal CPU utilization. At higher thread counts, manual threading may hit 100% CPU utilization, but overhead increases.

Table 11: Thread Pool vs. Manual Threading Implementation CPU Utilization

Key Takeaways:

- **Thread Pool:** Reusing threads allows for consistent and efficient CPU usage. The thread pool prevents the overhead of thread creation and destruction, which helps maintain a high level of CPU utilization, especially with a larger number of threads.

- **Manual Threading:** While **higher thread counts** can lead to high CPU utilization, the creation and destruction of threads for each task introduce significan overhead, making CPU usage less efficient, particularly with small datasets or fewer threads.

Memory Usage

Metric	Thread Pool	Manual Threading
Peak Heap Usage	**More efficient memory usage**. The thread pool minimizes memory overhead by reusing a fixed number of threads. Less memory is allocated to thread management.	**Higher memory usage**. Manual threading might generate more temporary objects (e.g., for each thread, new objects like task queues or local variables could be created), resulting in increased memory consumption.
Memory Efficiency	**Higher** efficiency with large datasets. Thread pool ensures threads do not accumulate excessive memory or resources over time. It reuses threads, preventing memory fragmentation.	**Less efficient**, especially with smaller datasets. Memory might be wasted as threads are created and destroyed, requiring additional memory for task scheduling, local variables, and more garbage collection overhead.

Table 12: Thread Pool vs. Manual Threading Implementation Memory Usage

Key Takeaways:

- **Thread Pool:** The pool's reuse of threads generally leads to **lower peak memory** usage, making it more memory-efficient, especially when dealing with larger datasets or many concurrent tasks. Thread pools prevent excess memory allocation by managing a fixed number of threads.

- **Manual Threading:** More memory is allocated dynamically for each thread, resulting in **higher peak memory usage**. For large numbers of threads, this may cause memory fragmentation and inefficient memory usage, which is not optimized as the threads are individually created and destroyed.

Garbage Collection (GC)

Metric	Thread Pool	Manual Threading
GC Count	Lower GC count. Since threads are reused and fewer objects are created, the thread pool approach generally leads to fewer memory allocations and, thus, fewer GC events.	Higher GC count. In manual threading, threads are created and destroyed more frequently, which results in higher memory allocations and more frequent garbage collection events.
Total GC Pause Time	Shorter GC pauses. Fewer memory allocations mean less time spent by the garbage collector cleaning up unnecessary objects. The thread pool approach minimizes GC pauses by limiting the number of threads and memory usage.	Longer GC pauses. The increased memory allocation and higher GC count in manual threading can lead to longer garbage collection pauses, especially when threads and objects are frequently created and discarded.

Table 13: Thread Pool vs. Manual Threading Implementation Garbage Collection

Key Takeaways:

- **Thread Pool:** The reuse of threads and minimizing of object creation in the thread pool approach generally results in **fewer GC events** and **shorter GC pause times**. With a fixed number of threads, memory management is more predictable, reducing the need for frequent garbage collections.

- **Manual Threading:** The increased memory allocation from dynamically created threads causes more frequent **garbage collection events** and **longer pauses** due to the larger number of temporary objects that need to be cleaned up. More threads can mean more objects and a larger memory footprint, leading to longer GC times.

Fork/Join Framework Solution

Execution Time

Dataset Size	Execution Time (ms)	Speedup
27368	9546	2.59
140196	11388	3.82
410616	4058	13.11

Table 14: Execution Time (ms) Results for Fork/Join Framework Implementation

Observations:

- The execution time decreases substantially with larger datasets, where the benefits of recursive parallelism and workload distribution outweigh the task-splitting overhead.
- Though the Fork/Join framework does not expose thread count control explicitly, it scales with available processor cores and performs automatic load balancing. When running on machines with 8 or more cores, the solution achieves a near-linear scaling on large datasets.
- The Fork/Join model is most effective when the cost of splitting and merging is outweighed by the computational cost of the task itself – which is clearly the case for the 410616-page dataset.

CPU Utilization

Dataset Size	CPU Utilization (%)
27368	100
140196	99,5
410616	99,4

Table 15: CPU Utilization (%) for Fork/Join Framework Implementation

Observations:

- The Fork/Join implementation effectively saturates CPU usage across all dataset sizes, particularly on large inputs. This is due to the dynamic nature of the common pool and the work-stealing algorithm, which keeps all threads active with minimal idling.

Memory Usage and Garbage Collection (GC)

Dataset Size	Peak Heap Usage (MB)	GC Count	Total GC Pause Time (s)
27368	3790	325	9
140196	4295	316	14,3

410616	2942	125	3,41

Table 16: Memory Usage and Garbage Collection for Fork/Join Framework Implementation

Observations:

- The Fork/Join solution maintains relatively low memory pressure and GC activity. This is partly due to the short-lived nature of many subtasks and the lack of explicit queues or large shared data structures.

Conclusions

This project evaluated three concurrent implementations for parallel word counting: **Manual Threading**, **Thread Pool**, and the **Fork/Join Framework**. Each approach demonstrated strengths under specific conditions:

- **Thread Pool** provided the most consistent performance across small to medium datasets, benefiting from thread reuse and reduced management overhead.
- **Fork/Join Framework** showed superior scalability and resource efficiency on large datasets, leveraging recursive task decomposition and work-stealing to maximize CPU utilization with minimal memory and GC overhead.
- **Manual Threading**, while functional, incurred the highest overhead due to explicit thread and synchronization management, making it less suitable for scalable workloads.

Overall, **Fork/Join** proved to be the most efficient solution for high-volume, CPU-bond tasks in a multicore environment, while **Thread Pooling** remains a practical and balanced approach for general-purpose parallelism.

Appendix

- 27368 Wikipedia pages.
- 140196 Wikipedia pages.
- 410616 Wikipedia pages.