# Optimizing Large-Scale Data Processing on Multicore Systems

**Sistemas Multinúcleo e Distribuídos**

**2º Semestre**

**2024/2025**

Guilherme Pacheco da Cunha 1201506 – Turma 1MB

Rui Jorge Castro Lemos Lopes 1230210 – Turma 1MB

Prof. Doutor Jorge Manuel Neves Coelho (JMN)

# Table of Contents

## Introduction

This project focuses on processing large amounts of data efficiently using multicore systems. The main goal is to analyze different ways of counting word frequencies from a large Wikipedia XML dump using Java. To do this, several implementations were developed, starting with a basic sequential solution and followed by multithreaded and asynchronous approaches.

Each version takes advantage of Java's concurrency features to improve performance, including thread management, thread pools, the Fork/Join framework, and Completable Futures. The performance of each solution is measured and compared based on execution time, CPU usage, memory usage, and garbage collection activity.

All the implementations were placed into a benchmark class that facilitates the testing and the comparison of results.

This project helps us understand how different concurrency models can affect the speed and efficiency of data processing on multicore systems.

## Objectives

The goal of this project is to process a large Wikipedia XML file and count how often each word appears. To do this, we'll create several versions of the program using different ways to run tasks in parallel with a multicore system.

We want to:

- Start with a basic sequential version to use as performance reference.
- Build a multithreaded version without thread pools, where we manage threads manually.
- Create a version using thread pools to improve efficiency and reduce thread overhead.
- Use the Fork/Join framework to split work into smaller parts and run them in parallel.
- Try an asynchronous version with **CompletableFuture** to handle tasks without blocking.
- Tune the garbage collector to improve memory usage and app performance.
- Measure execution time, CPU, and memory usage, and see how well each version scales.

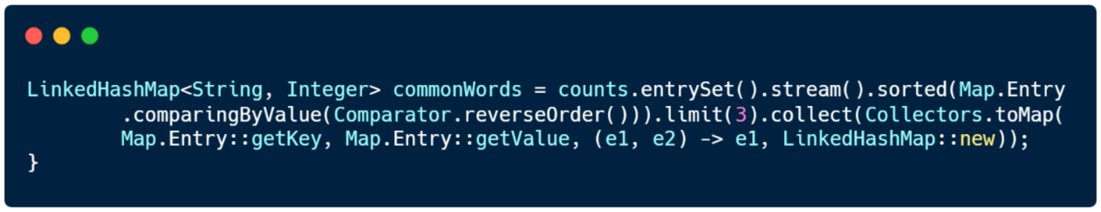
The aim is to see which method works best and what trade-offs each one has.

## Implementation Approaches

### Sequential Solution

This is the basic version that runs in a single thread. It reads Wikipedia pages one by one, gets the text, splits it into words, and counts how often each word appears.

It uses a HashMap to store word counts and increases the count with **merge()**. At the end, it sorts the words by frequency and shows the top 3 most common ones.



```
LinkedHashMap<String, Integer> commonWords = counts.entrySet().stream().sorted(Map.Entry
        .comparingByValue(Comparator.reverseOrder())).limit(3).collect(Collectors.toMap(
        Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
}
```

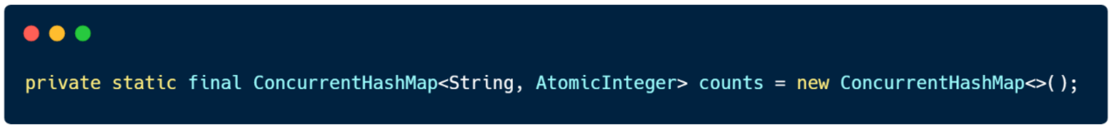
This version is easy to understand but doesn't use multiple cores, so it's slower with large files. It's used as a baseline to compare with faster, parallel versions.

### Multithreaded Solution (Without Thread Pools)

This implementation is a multithreaded approach designed to efficiently process many pages without using thread pools. The solution leverages low-level thread management combined with thread-safe data structures to ensure efficient and concurrent word counting.

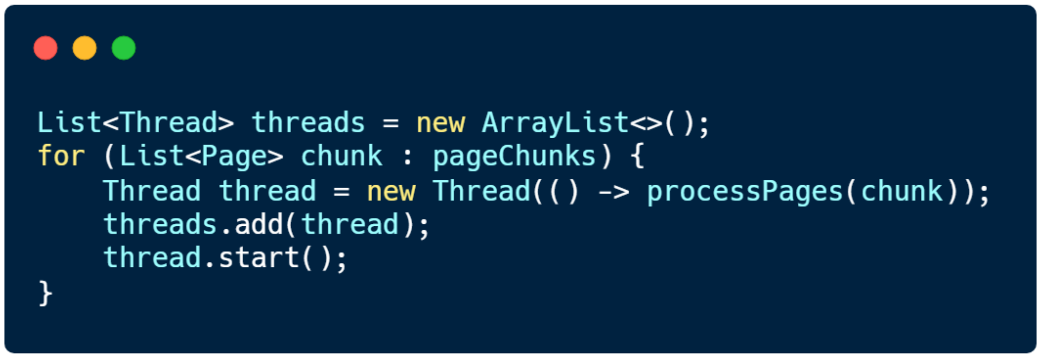
**Concurrency and Synchronization**

- **ConcurrentHashMap** and **AtomicInteger**: The counts map is defined as a **ConcurrentHashMap<String, AtomicInteger>**, ensuring that word counting is both thread-safe and with good performance (compared with the sequential solution). The **ConcurrentHashMap** provides concurrent access without explicit synchronization, while **AtomicInteger** ensures that each word count is updated atomically.



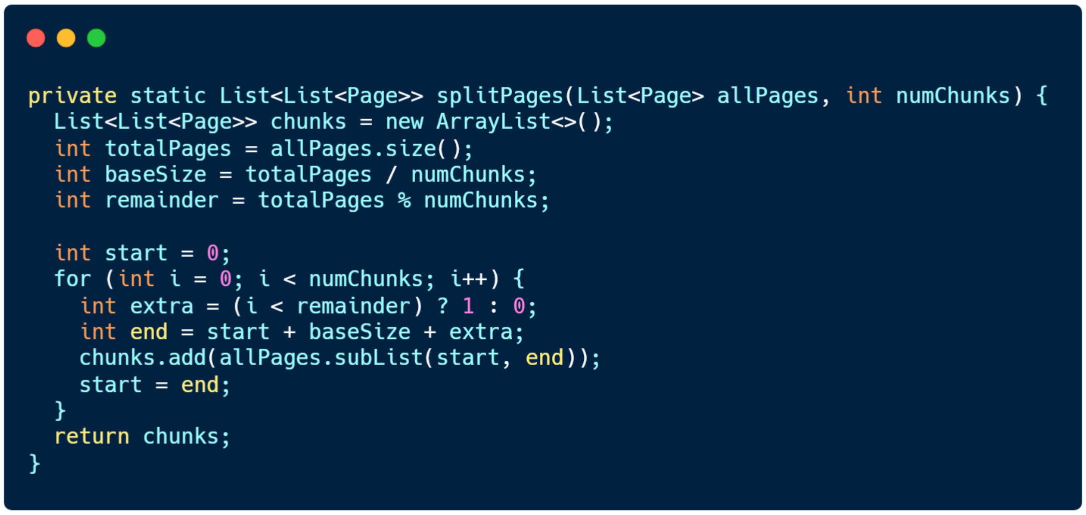
```
private static final ConcurrentHashMap<String, AtomicInteger> counts = new ConcurrentHashMap<>();
```

- **Manual Thread Management**: The algorithm creates and manages threads manually instead of using a thread pool. Each thread is responsible for processing a chunk of pages, enhancing parallelism.



```java
List<Thread> threads = new ArrayList<>();
for (List<Page> chunk : pageChunks) {
    Thread thread = new Thread(() -> processPages(chunk));
    threads.add(thread);
    thread.start();
}
```

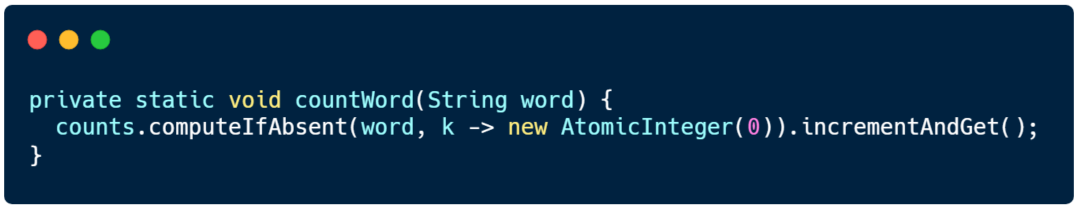
- **Page Chunking**: To balance the workload among threads, the pages are divided into chunks using the **splitPages** method. This method splits the pages into a number of chunks specified by **numChunks**, ensuring an even distribution of pages across threads.



```java
private static List<List<Page>> splitPages(List<Page> allPages, int numChunks) {
  List<List<Page>> chunks = new ArrayList<>();
  int totalPages = allPages.size();
  int baseSize = totalPages / numChunks;
  int remainder = totalPages % numChunks;

  int start = 0;
  for (int i = 0; i < numChunks; i++) {
    int extra = (i < remainder) ? 1 : 0;
    int end = start + baseSize + extra;
    chunks.add(allPages.subList(start, end));
    start = end;
  }
  return chunks;
}
```

### Atomic Updates for Word Counting

The **countWord** uses the **computeIfAbsent** method of **ConcurrentHashMap** to ensure that the **AtomicInteger** for a word is initialized only once, even when accessed by multiple threads. The **incrementAndGet** method then ensures atomic updates to the word count.

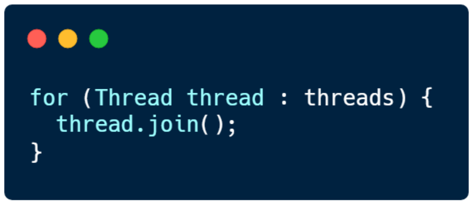


```java
private static void countWord(String word) {
  counts.computeIfAbsent(word, k -> new AtomicInteger(0)).incrementAndGet();
}
```

This combination of **ConcurrentHashMap** and **AtomicInteger** avoids the need for **explicit locking**, maintaining **high performance** even under **concurrent** access.

**Thread Synchronization with Join**

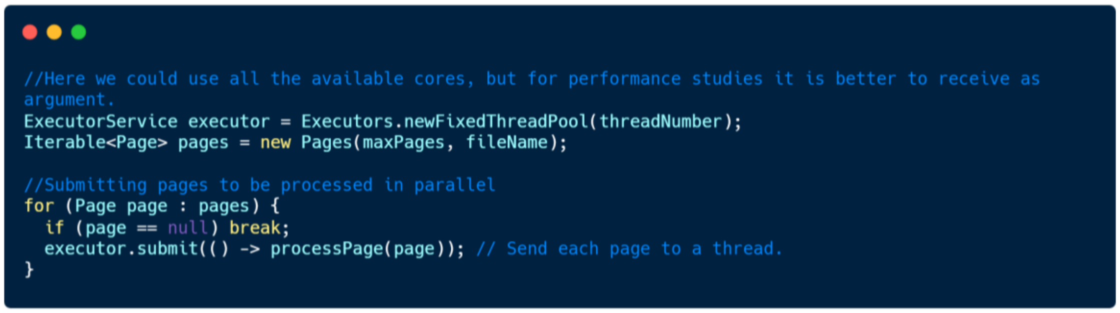
The main thread waits for all worker threads to complete using the join method, ensuring that the final word count is only displayed after all pages have been processed.



```java
for (Thread thread : threads) {
  thread.join();
}
```

## Multithreaded Solution (With Thread Pools)

This implementation uses a fixed-size thread pool to process text pages in parallel. Each page is submitted as a task to the thread pool, where it's processed by extracting words and updating a shared word frequency map (the number of threads in the pool is passed in as a parameter, giving flexibility to experiment with performance on different systems).



```java
//Here we could use all the available cores, but for performance studies it is better to receive as
argument.
ExecutorService executor = Executors.newFixedThreadPool(threadNumber);
Iterable<Page> pages = new Pages(maxPages, fileName);

//Submitting pages to be processed in parallel
for (Page page : pages) {
  if (page == null) break;
  executor.submit(() -> processPage(page)); // Send each page to a thread.
}
```

To safely update word counts across multiple threads, it uses a **ConcurrentHashMap** combined with **AtomicInteger** (this ensures that word counts are incremented safely without needing synchronization).

Once all pages have been submitted, the program waits for all tasks to be completed, then calculates and prints the three most frequent words.

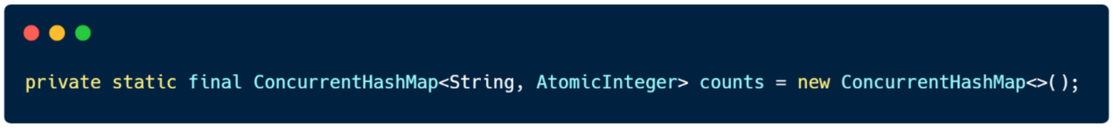
This approach is designed to make good use of multi-core CPUs and can significantly reduce processing time for large text inputs. There's also room for future optimization, such as batching small pages to reduce overhead from thread scheduling.

## Fork/Join Framework Solution

This implementation uses **Fork/Join Framework** to process several pages in parallel. It pulls **work-stealing** and recursive task decomposition to efficiently divide the workload across multiple threads, enabling high performance word counting through parallelism and optimized thread management.

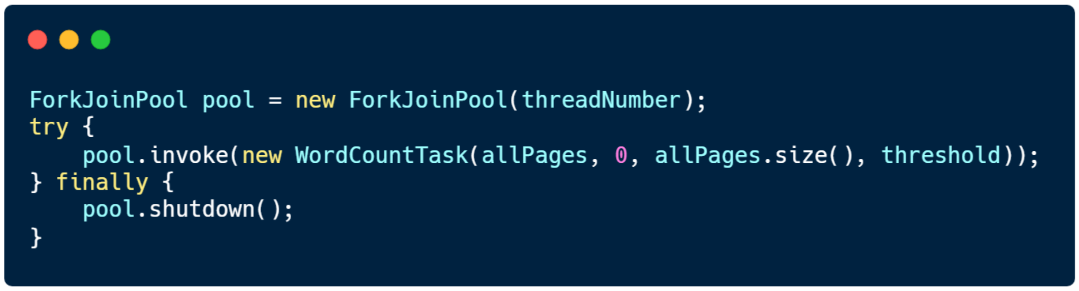
### Concurrency and Synchronization

- **ConcurrentHashMap and AtomicInteger**: The counts map is defined as a **ConcurrentHashMap<String, AtomicInteger>**, ensuring that word counting is both thread-safe and performant. The **ConcurrentHashMap** provides efficient concurrent access without the need for explicit synchronization, while **AtomicInteger** ensures that each word's count is updated atomically.



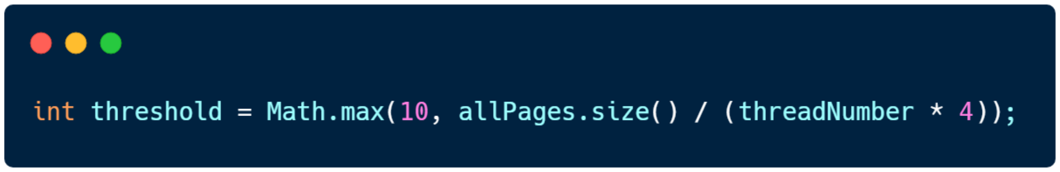
```java
private static final ConcurrentHashMap<String, AtomicInteger> counts = new ConcurrentHashMap<>();
```

- **ForkJoin Framework**: This solution uses the Fork/Join framework, leveraging the ForkJoinPool and the RecursiveAction class to achieve efficient parallelism.

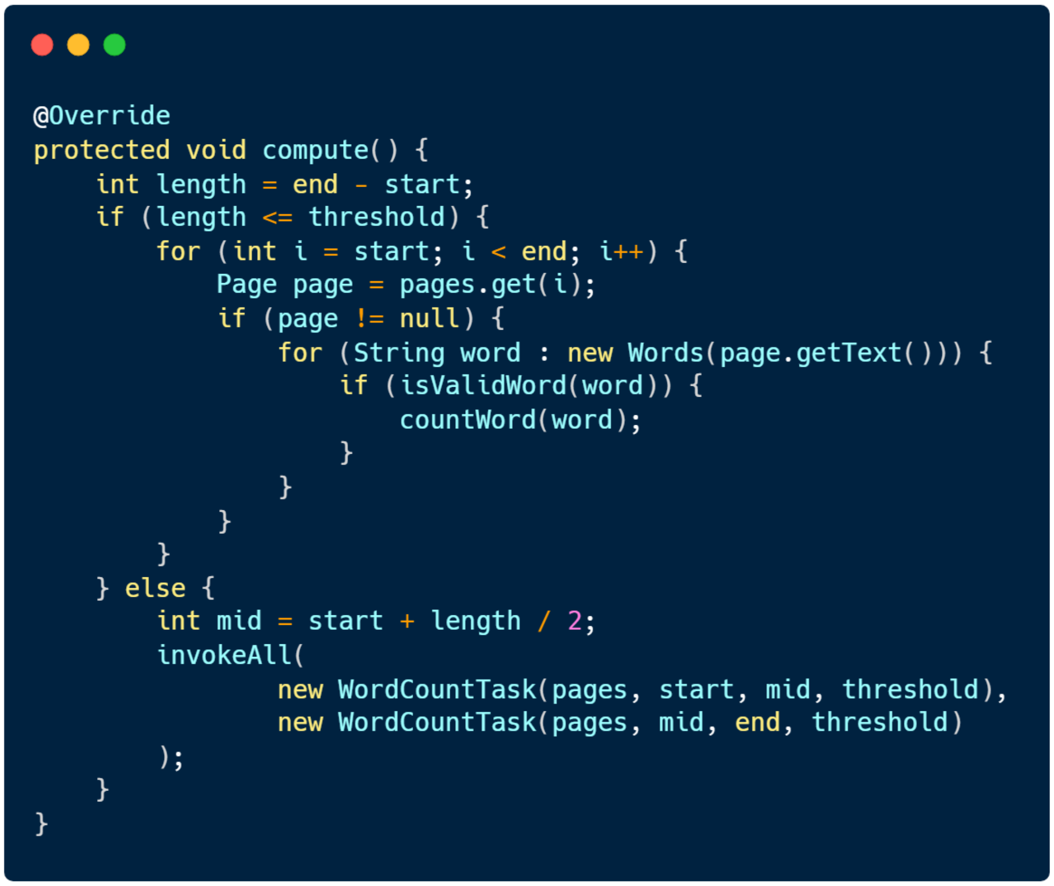


```java
ForkJoinPool pool = new ForkJoinPool(threadNumber);
try {
    pool.invoke(new WordCountTask(allPages, 0, allPages.size(), threshold));
} finally {
    pool.shutdown();
}
```

- **Task Splitting with RecursiveAction**: The **WordCountTask** class extends **RecursiveAction** and recursively splits tasks when the threshold is exceeded, ensuring a balanced workload among threads. The threshold was tested in a few ways, and that performed best was the one where we split the number of pages by the thread number multiplied by 4, so threads have a balanced workload.



```java
int threshold = Math.max(10, allPages.size() / (threadNumber * 4));
```

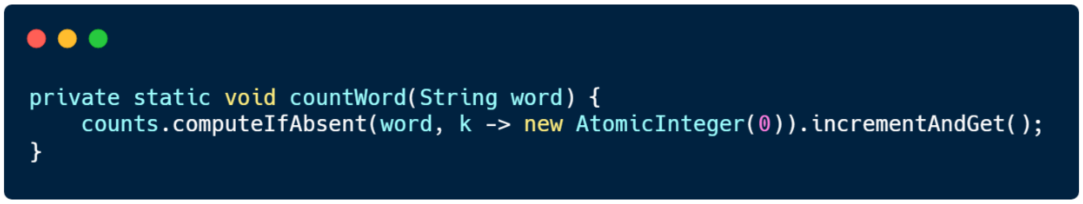


```java
@Override
protected void compute() {
    int length = end - start;
    if (length <= threshold) {
        for (int i = start; i < end; i++) {
            Page page = pages.get(i);
            if (page != null) {
                for (String word : new Words(page.getText())) {
                    if (isValidWord(word)) {
                        countWord(word);
                    }
                }
            }
        }
    } else {
        int mid = start + length / 2;
        invokeAll(
                new WordCountTask(pages, start, mid, threshold),
                new WordCountTask(pages, mid, end, threshold)
        );
    }
}
```

This approach provides automatic load balancing and efficient task splitting, making it highly suitable for CPU-bound tasks like text processing. It reduces the overhead of manual thread management while maximizing parallelism.

**Atomic Updates for Word Counting**

The **countWord** method uses the **computeIfAbsent** method of **ConcurrentHashMap** to ensure that the **AtomicInteger** for a word is initialized only once, even when accessed by multiple threads. The **incrementAndGet** method then ensures atomic updates to the word count.

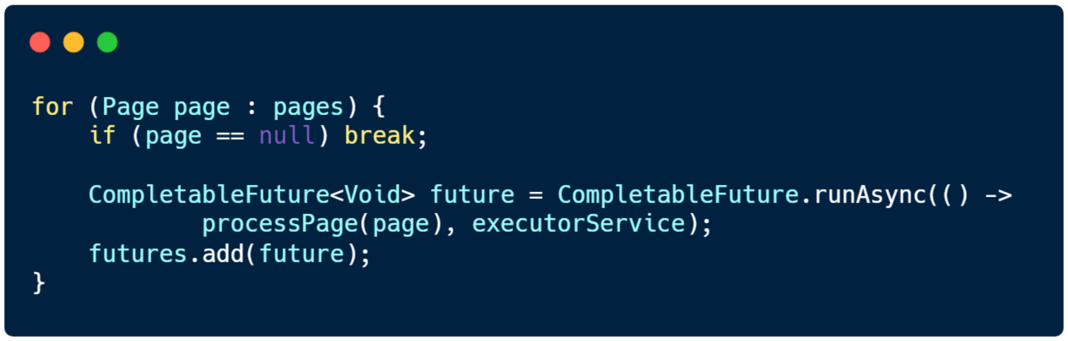


```java
private static void countWord(String word) {
    counts.computeIfAbsent(word, k -> new AtomicInteger(0)).incrementAndGet();
}
```

This combination of **ConcurrentHashMap** and **AtomicInteger** avoids the need for explicit locking, maintaining high performance even under concurrent access.

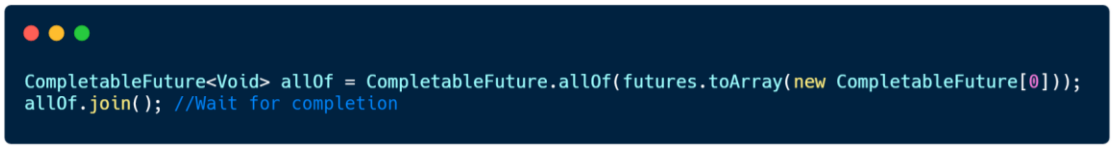
## Completable-Future Based Solution

In this version, we used **CompletableFuture** to run the processing pages in parallel without having to manage the threads ourselves. For each page, we launch a task asynchronously using **CompletableFuture.runAsync**, which lets multiple pages be processed at the same time using a thread pool.



```
for (Page page : pages) {
    if (page == null) break;

    CompletableFuture<Void> future = CompletableFuture.runAsync(() ->
            processPage(page), executorService);
    futures.add(future);
}
```

All the word counts go into a shared ConcurrentHashMap<String, AtomicInteger>, which keeps things thread-safe and avoids any weird concurrency issues. Once all the tasks are running, we just wait for them to finish using CompletableFuture.allOf(...).join().



```
CompletableFuture<Void> allOf = CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]));
allOf.join(); //Wait for completion
```

After everything's done, we go through the word counts and print out the top 3 most common ones (just like is done in other algorithms).

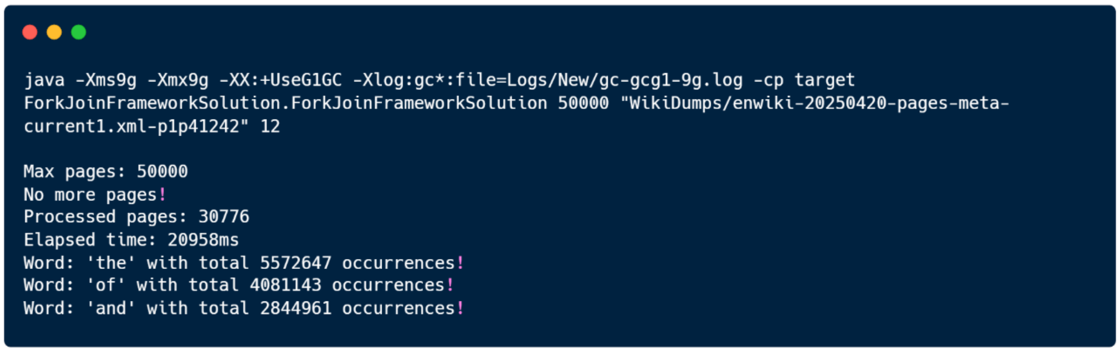
## Garbage Collector Tuning

We will evaluate the performance impact of different garbage collectors on a large-scale data processing system. The focus is on three popular Java garbage collectors: **ZGC (Z Garbage Collector)**, **G1GC (Garbage-First Collector)**, and **ParallelGC**, with particular emphasis on execution time efficiency. We will use **GCeasy** to generate tables based on the logs produced during the execution of the garbage collectors.

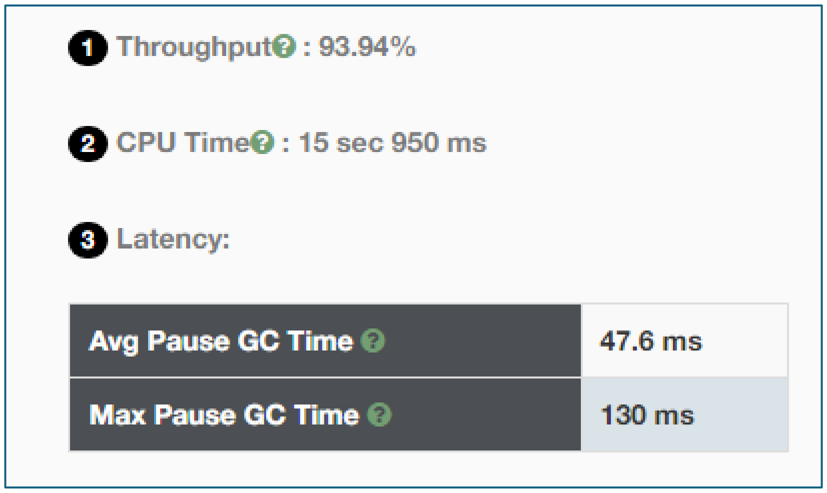
### G1GC

After several attempts, we concluded that the optimal values for Xms and Xmx when using **G1GC** were **9GB**, based on execution time and throughput. With both higher and lower memory values, the throughput was below the average achieved with 9GB. We used a sample of 50000 pages with the Fork/Join solution.

In this first test with **G1GC**, no -**XX:MaxGCPauseMillis** value was provided to observe its default behavior. We observed a throughput of 93.926% along with an execution time of 20958ms, which is a strong result compared to other memory configurations.



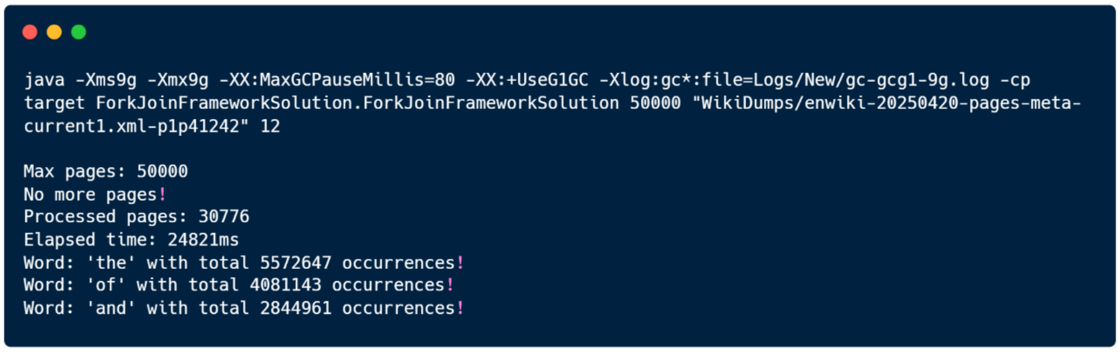
```
java -Xms9g -Xmx9g -XX:+UseG1GC -Xlog:gc*:file=Logs/New/gc-gcg1-9g.log -cp target
ForkJoinFrameworkSolution.ForkJoinFrameworkSolution 50000 "WikiDumps/enwiki-20250420-pages-meta-
current1.xml-p1p41242" 12

Max pages: 50000
No more pages!
Processed pages: 30776
Elapsed time: 20958ms
Word: 'the' with total 5572647 occurrences!
Word: 'of' with total 4081143 occurrences!
Word: 'and' with total 2844961 occurrences!
```



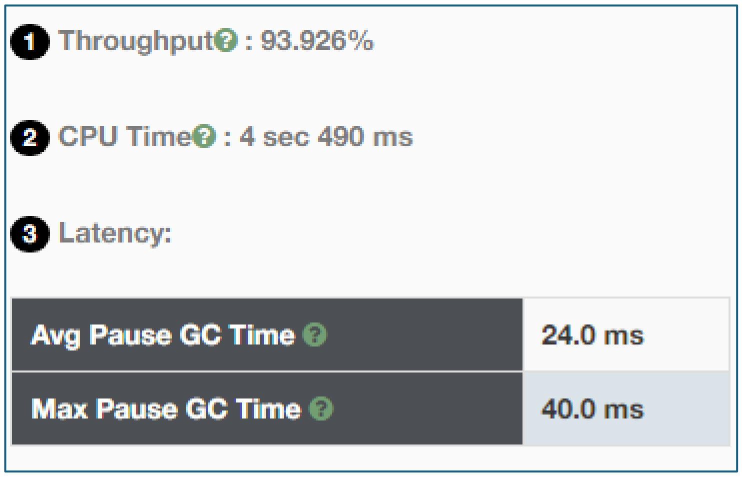
1. Throughput : 93.94%
2. CPU Time : 15 sec 950 ms
3. Latency:

| | |
|---|---|
| Avg Pause GC Time | 47.6 ms |
| Max Pause GC Time | 130 ms |

We tested 100ms and 80ms as the maximum pause time values to reduce throughput and improve execution time. The best result was achieved with default values, which led to both a lower throughput and shorter execution time.



```
java -Xms9g -Xmx9g -XX:MaxGCPauseMillis=80 -XX:+UseG1GC -Xlog:gc*:file=Logs/New/gc-gcg1-9g.log -cp
target ForkJoinFrameworkSolution.ForkJoinFrameworkSolution 50000 "WikiDumps/enwiki-20250420-pages-meta-
current1.xml-p1p41242" 12

Max pages: 50000
No more pages!
Processed pages: 30776
Elapsed time: 24821ms
Word: 'the' with total 5572647 occurrences!
Word: 'of' with total 4081143 occurrences!
Word: 'and' with total 2844961 occurrences!
```



1 Throughput : 93.926%

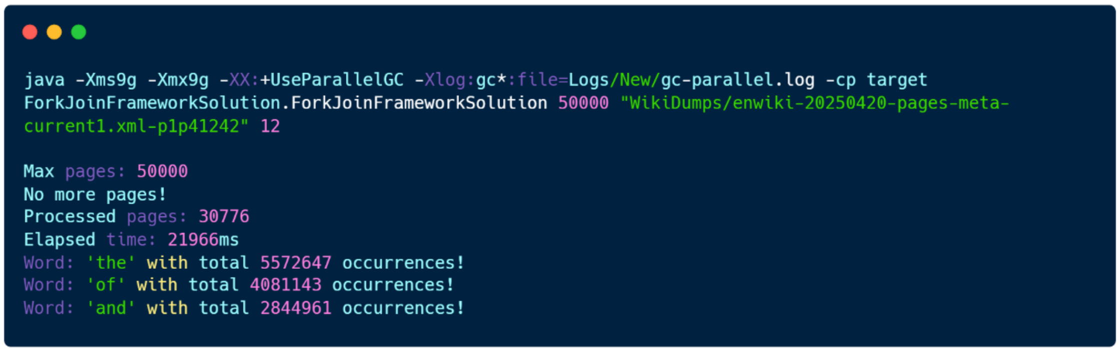
2 CPU Time : 4 sec 490 ms

3 Latency:

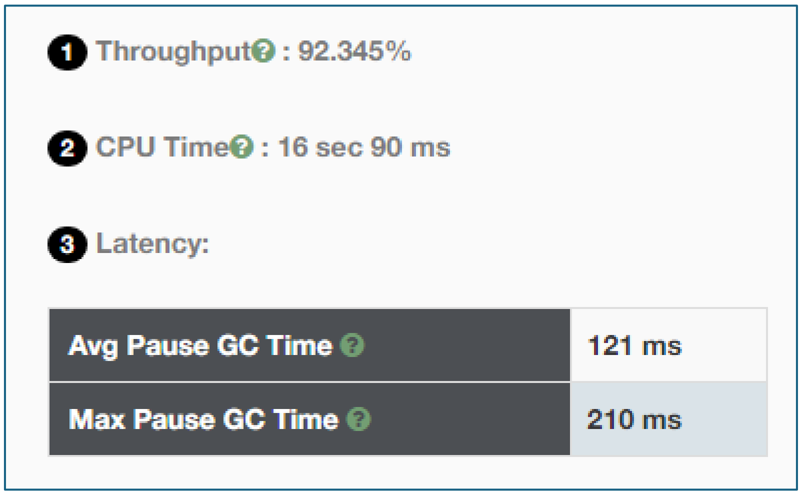
| | |
|---|---|
| Avg Pause GC Time | 24.0 ms |
| Max Pause GC Time | 40.0 ms |

## Parallel GC

ParallelGC presented a higher execution time compared to G1GC and ZGC, along with a lower throughput percentage. Although it is designed to maximize throughput by utilizing multiple threads during garbage collection, in our specific scenario it did not offer competitive performance. The longer execution time and reduced efficiency led us to exclude ParallelGC as a viable option



```
java -Xms9g -Xmx9g -XX:+UseParallelGC -Xlog:gc*:file=Logs/New/gc-parallel.log -cp target
ForkJoinFrameworkSolution.ForkJoinFrameworkSolution 50000 "WikiDumps/enwiki-20250420-pages-meta-
current1.xml-p1p41242" 12

Max pages: 50000
No more pages!
Processed pages: 30776
Elapsed time: 21966ms
Word: 'the' with total 5572647 occurrences!
Word: 'of' with total 4081143 occurrences!
Word: 'and' with total 2844961 occurrences!
```


1 Throughput : 92.345%

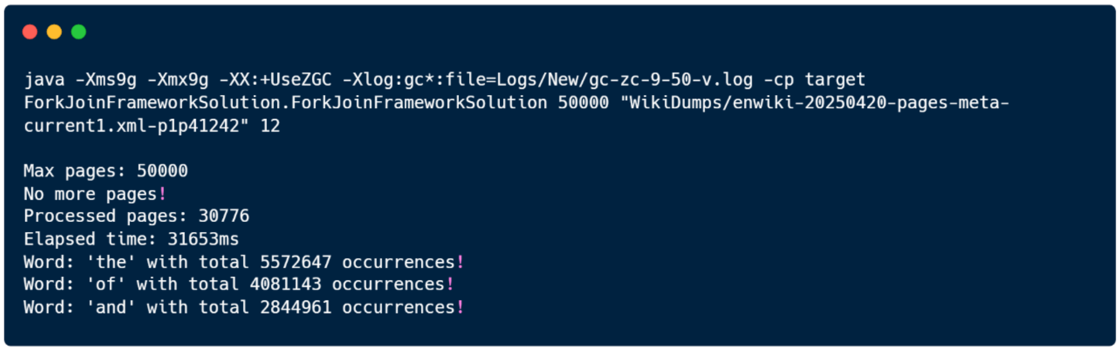
2 CPU Time : 16 sec 90 ms

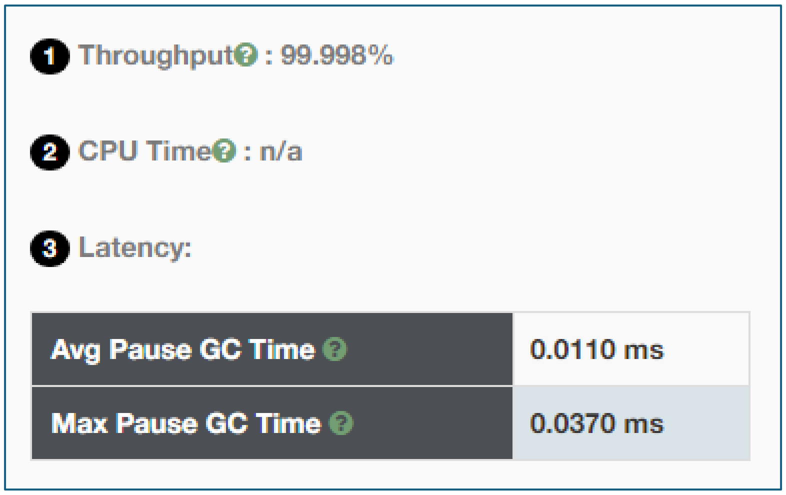
3 Latency:

| | |
|---|---|
| Avg Pause GC Time | 121 ms |
| Max Pause GC Time | 210 ms |

## Z GC

ZGC showed extremely high throughput, indicating that the system processes data at a very fast rate. However, it also showed the highest execution time (31,653 ms) when compared to the other collectors. Despite its strong performance in terms of latency, it proved to be less efficient overall for the specific requirements of this workload.


```
java -Xms9g -Xmx9g -XX:+UseZGC -Xlog:gc*:file=Logs/New/gc-zc-9-50-v.log -cp target
ForkJoinFrameworkSolution.ForkJoinFrameworkSolution 50000 "WikiDumps/enwiki-20250420-pages-meta-
current1.xml-p1p41242" 12

Max pages: 50000
No more pages!
Processed pages: 30776
Elapsed time: 31653ms
Word: 'the' with total 5572647 occurrences!
Word: 'of' with total 4081143 occurrences!
Word: 'and' with total 2844961 occurrences!
```


1 Throughput : 99.998%

2 CPU Time : n/a

3 Latency:

| | |
|---|---|
| Avg Pause GC Time | 0.0110 ms |
| Max Pause GC Time | 0.0370 ms |

The table below summarizes the execution time and throughput of each tested garbage collector configuration. **ZGC** provided the highest throughput (99.998%), reflecting his ability to minimize pause times and keep application threads active. However, the execution time was very high with 31653ms, making it less suitable for scenarios where efficiency is a greater factor.

**G1GC** without pause time tuning achieved the best balance, with a low execution time (20958 ms) with a stable throughput of 93.826%. When tuning MaxGCPauseMillis to 100ms and 80ms, G1GC showed some variations in throughput and execution time, but no significant gains, leading us to choose the default configuration as the most efficient for this workload.

**ParallelGC**, while designed for high throughput, did not outperform G1GC. It delivered a longer execution time (21966ms) and lower throughput (92.350%), making it a less optimal choice.

Based on these results, **G1GC,** with **default** settings, was selected as the most efficient garbage collector for this task, offering the best balance between throughput and execution time, leading us to proceed with it.

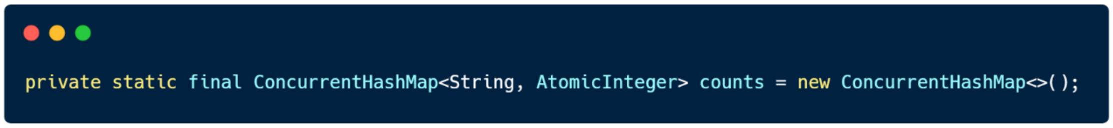
| Garbage Collector | Execution Time (ms) | Throughput (%) |
|---|---|---|
| ZGC | 31,653 | 99.998 |
| G1GC (default) | 20,958 | 93.826 |
| G1GC (MaxPause=100ms) | 25,013 | 91.913 |
| G1GC (MaxPause=80ms) | 24,821 | 93.940 |
| ParallelGC | 21,966 | 92.350 |

*Table 1- Garbage Collector performance comparation*

## Concurrency and Synchronization

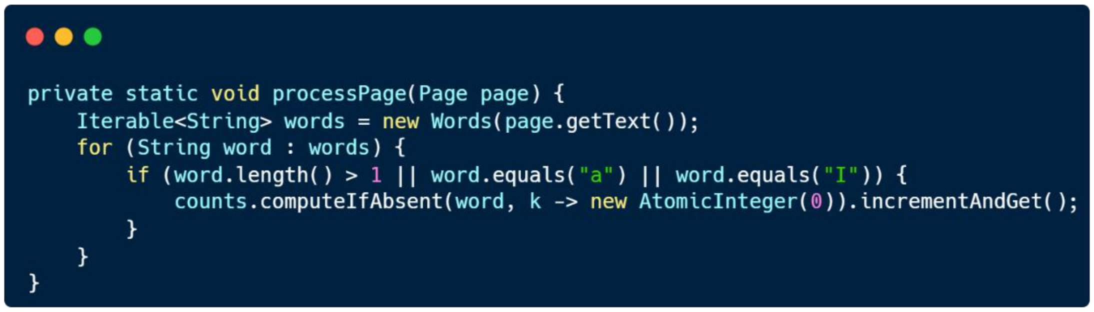
Concurrency and Synchronization is handled by breaking the work into smaller pieces and handling them at the same time. But doing things in parallel is not so straightforward, especially when threads need to share and update the same data. That's where synchronization implementation comes in.

To manage this efficiently, all solutions (that use threads) use a combination of **ConcurrentHashMap** and **AtomicInteger**. These help multiple threads update shared word counts without overlapping on each other. This avoids common mistakes like race conditions and keeps the code simple, without needing to manually lock and unlock things (semaphores).



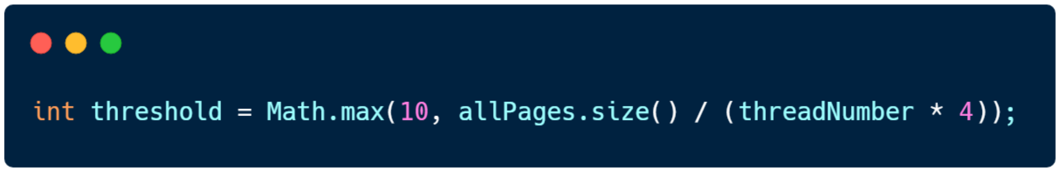
```java
private static final ConcurrentHashMap<String, AtomicInteger> counts = new ConcurrentHashMap<>();
```

Instead of locking the whole data structure (which would slow everything down), **ConcurrentHashMap** allows multiple threads to update different parts at the same time. And **AtomicInteger** gives each word's counter a thread-safe way to increase, ensuring accurate counts no matter how many threads are active.



```java
private static 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")) {
            counts.computeIfAbsent(word, k -> new AtomicInteger(0)).incrementAndGet();
        }
    }
}
```

The use of a threshold and dynamic work splitting across threads significantly contributed to improved performance results.



```java
int threshold = Math.max(10, allPages.size() / (threadNumber * 4));
```

In short, these solutions have a nice balance between performance and correctness, showing how it is possible to safely coordinate work between threads in Java.

## Performance Analysis

To evaluate how each implementation performs under different conditions, we benchmarked them using varying input sizes and thread counts. Specifically, we tested multiple implementations with increasing numbers of Wikipedia pages (from 500 t 25,000) and, for the multithreaded approaches, with different thread counts (2 to 16). This allows us to analyze how well each solution scales with both data volume and available parallelism.

For the inputs we have a variety of tests, some of them with a lot of pages but not many words, and others with a smaller number of pages but with a lot of words. Our biggest test has around 1GB, which was the one used for most of the performance measurements.

Machine 1 (1201506) – Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 16.0 GB RAM, 477 SSD.

Machine 2 (1230210) - AMD Ryzen 5 3600 6-Core Processor, 16 GB RAM, 1TB SSD.

Tests were performed in Machine 1, since the machines specs are similar and no relevant differences were identified during the benchmark in both machines.

### Sequential Solution

The Sequential Solution in terms of performance is consistently the slowest, as expected. It only uses one core and processes everything step by step.

We can verify this by the following spreadsheet which the **maxPages** were defined as 500.

These results are consistent even if the **maxPages** are bigger (in which the gap between the solutions becomes clearer). We can understand by the following graph exactly how slow the sequential solution is compared to others

### Multithreaded Solution (Without Thread Pools)

The multithreaded solution demonstrates substantial performance improvements over the sequential baseline across all input sizes. This information can be complemented with the data available in Attachments - Multithreaded Solution (Without Thread Pools).

| maxPages | Sequential (ms) | Multithreaded (ms) | Speedup |
|---|---|---|---|
| 1,000 | 2375.19 | 613.7 | 3.87 |
| 10,000 | 20836.00 | 6,278.16 | ~3.78 |
| 25,000 | 53498.53 | 15253.73 | ~3.73 |
| 50,000 | 56263.67 | 18,184.71 | ~3.74 |

*Table 2-Execution Time Comparison (12 threads vs Sequential)*

Despite the increased workload, the speedup remains stable at around **3.7–3.9×**, indicating that the algorithm distributes work effectively across threads, even as maxPages increases. This consistency highlights that the solution sustains its efficiency under growing data volume, suggesting it handles computational load well in a parallel environment.

**Scalability**

Scalability is analyzed along two axes: increasing **number of threads** and increasing **data size**.

| Threads | Time (ms) | CPU Usage (%) |
|---|---|---|
| 1 | 56263.67 | 138.12 |
| 2 | 33532.32 | 126.44 |
| 4 | 23873.77 | 96.48 |
| 8 | 19380.17 | 72.39 |
| 12 | 18184.71 | 56.97 |
| 16 | 19242.92 | 40.92 |

*Table 3-Thread-Level Scalability (maxPages = 50,000):*

Performance scales well up to **8-12 threads**, after the improvements, indicating that the workload becomes too fine-grained or thread contention becomes a limiting factor. CPU usage also decreases, suggesting underutilization at higher thread counts.

| maxPages | Time (ms) | Memory (MB) | CPU Usage (%) |
|---|---|---|---|
| 1,000 | 613.70 | 0.16 | 35.43 |
| 10,000 | 6278.16 | 0.16 | 36.47 |
| 25,000 | 15253.73 | 0.16 | 41.92 |
| 50,000 | 18184.71 | 0.16 | 40.92 |

*Table 4-Data-Level Scalability (12 threads)*

Execution time increases almost linearly with the number of pages, confirming the algorithm's scalability with respect to data volume. Notably, memory usage remains constant, indicating that the design does not suffer from memory leaks or per-page memory growth. The **best runtime is achieved at 12 threads**, corresponding exactly to the **number of logical threads** on the CPU (machine 1).

**Overhead Analysis**

Using raw threads instead of a thread pool introduces measurable overhead:

- **Thread Creation Overhead:** Each thread is created and destroyed per execution, increasing the startup cost and limiting reusability.
- **Garbage Collection (GC):**
    - At low thread counts, GC events and time are significantly higher (e.g., 2 threads: 112 ms GC time).
    - At higher thread counts, GC behavior stabilizes (6–7 collections, ~76–80 ms), likely due to better memory partitioning and shorter-lived thread lifecycles.
- **CPU Efficiency:** CPU usage drops as threads increase, suggesting that excessive threads may result in contention or idle time due to synchronization delays and core saturation.

Overall, manual thread management proves inefficient at scale and becomes a bottleneck for further parallelism.

**Bottlenecks and Limitations**

Some performance bottlenecks are evident:

1. **Thread Saturation:**

    - Beyond 12 threads, performance slightly **declines**, which is expected, as more threads than hardware contexts lead to context switching and scheduler overhead.
    - Indicates diminishing returns due to increased context switching and thread contention.
2. **Atomic Operations on Shared Map:**
    - While ConcurrentHashMap and AtomicInteger provide thread-safe updates, high write concurrency on shared keys may introduce contention, especially for frequent words.
3. **Lack of Task Reuse:**
    - The absence of a thread pool leads to excessive thread lifecycle overhead.
    - No work-stealing or dynamic load balancing exists; threads process fixed-size chunks regardless of the actual processing cost per page.
4. **No Batching of Word Counting:**
    - Each word is counted individually using atomic increments, which may create overhead under high concurrency.

## Multithreaded Solution (With Thread Pools)

Looking at the Multithreaded Solution with Thread Pools results, it's clear that as the number of pages increases, the time it takes to process them also grows, is expected.

We can observe that by the following table (looking at the graph is also helpful).

| **maxPages** | **Time (ms)** | **Memory (MB)** | **CPU Usage (%)** |
|---|---|---|---|
| 5000 | 2418.76 | 0.27 | 37.1 |
| 10000 | 7270.17 | 0.03 | 46.99 |
| 25000 | 15548.35 | 0.28 | 55.88 |

*Table 5-Data-Level Scalability (12 threads)*

Adding more threads helps with performance, but the improvement starts to slow down as the dataset gets bigger. For example, moving from 2 threads to 16 threads makes a big difference with 5000 pages, but the gap is small with 10000 and 25000 pages. So, while more threads help, the gains become less noticeable on larger datasets.

Memory usage stays steady around 1 MB once you have 4 threads or more, showing that the solution with Thread Pool is efficient in managing resources. However, GC time and GC count do rise with the larger datasets, but not too much.

In short, solution with Thread Pool works well for smaller data (like 5000 pages), but as the data grows, the performance boost from adding more threads becomes less noticeable.

## Fork/Join Solution

The Fork/Join solution demonstrates substantial performance improvements over the sequential baseline across all input sizes:

**Efficiency Gains (12/16 threads vs Sequencial)**

Across all tested data sizes, the Fork–Join implementation consistently outperforms the sequential algorithm by a substantial margin. On the smallest workload, it reduces execution time from 3489.75ms to 671.25ms, a 5.20x speedup, by leveraging all 12–16 logical threads on the CPU. As the data size grows, the absolute time savings increase (e.g., a 49,408.66ms reduction at 50 000 pages), while the relative speedup stabilizes around 3.2x to 3.4x. These results demonstrate that the divide and collect model of Fork/Join effectively reduces per-task overhead even as workload scales, ensuring that thread-level parallelism yields consistent and predictable gains over single-threaded execution**.**

| maxPages | Sequential (ms) | Best Fork/Join (ms) | Best Threads | Speedup |
|---|---|---|---|---|
| 1 000 | 3489.75 | 671.25 | 16 | 5.20x |
| 10,000 | 24211.03 | 7178.15 | 12 | 3.37x |
| 25,000 | 62032.69 | 19516.35 | 16 | 3.18x |
| 50,000 | 72262.06 | 22853.40 | 12 | 3.16x |

*Table 6-Algorithms speedup comparation*

**Scalability**

When varying the thread count for a fixed small workload (1 000 pages), execution time improves steeply from 3489.75ms (1 thread) to 685.92ms (12 threads), adding threads beyond the CPU's 12 logical processors only benefit (down to 671.25ms at 16 threads). This behavior indicates efficient utilization of available hardware until core saturation and context-switch overhead dominate. On the data axis (fixed at 12 threads), runtime grows sub linearly with input size: a tenfold increase in pages (1000 to 10000) yields approximately a 10.5x increase in time, and a further fivefold growth (10000 to 50000) results in only a 3.2x increase. This scaling reflects the Fork/Join framework's ability to balance workload dynamically and reduce the relative impact of task startup and synchronization costs as the problem size grows.

| Threads | Time (ms) | CPU Usage (%) |
|---|---|---|
| 1 | 3489.75 | 197.35 |
| 2 | 1755.26 | 117.18 |
| 4 | 894.69 | 67.92 |
| 8 | 712.50 | 50.18 |
| 12 | 685.92 | 42.73 |
| 16 | 671.25 | 37.56 |

*Table 7-Thread-Level Scalability (maxPages = 50000)*

| maxPages | Time (ms) | CPU (%) | Memory (MB) | GC Time (ms) |
|---|---|---|---|---|
| 1,000 | 685.92 | 42.73 | 0.16 | 75 |
| 10,000 | 7178.15 | 44.60 | 0.03 | 459 |
| 25,000 | 19615.72 | 62.79 | 0.03 | 953 |
| 50,000 | 22853.40 | 64.90 | 0.03 | 1090 |

*Table 8-Data-Level Scalability (12 threads)*

**Overhead Analysis**

1. **Task Division & Scheduling**
    - The dynamic threshold (N/(T×4)) keeps subtasks appropriately sized, but frequent splitting for small workloads adds overhead.
    - Fork/Join work-stealing reduces idle time, evident in stable CPU usage for large runs, but incurs bookkeeping cost.

2. **Synchronization Contention**
   - All threads update a shared ConcurrentHashMap<…AtomicInteger>.
   - High-frequency words become hotspots, limiting write throughput and reducing marginal speedup.

**Bottlenecks**

- **Hardware Saturation:** Beyond 12 threads, context-switching overhead outweighs parallel gains.
- **Shared-Map Contention:** Atomic increments on the global map are serialized under heavy writing load.
- **Static Work Partitioning:** Although Fork/Join splits dynamically, the initial threshold calc may produce uneven chunk sizes if page processing times vary.

## Completable-Future Solution

Looking at **CompletableFutures**-based solution, we can see that it handles increasing workloads well, the time to process more pages does go up, as expected, but it scales in a relatively smooth way. The jump in processing time from 5000 to 25000 pages is consistent with the growth in data, which suggests the parallelism is being put to good use.

| maxPages | Time (ms) | Memory (MB) | CPU Usage (%) |
|---|---|---|---|
| 5,000 | 2744.96 | 1 | 53.32 |
| 10,000 | 5661.68 | 1 | 55.38 |
| 25,000 | 13821.27 | 1 | 57.89 |

*Table 9-Data-Level Scalability (12 threads)*

Interestingly, memory usage remains very stable across all workloads (around 0.3 MB), indicating efficient memory handling even when the task size increases. However, we also notice that as the dataset grows, garbage collection (GC) becomes more active — with more GC events and a rise in total GC time, especially noticeable at 25000 pages. This suggests a bit more pressure on the memory system as tasks pile up, which is natural in a CompletableFutures model where many tasks can be scheduled simultaneously.

CPU usage seems to be rising steadily but slowly as the input grows, which is perfectly normal, acceptable and expected. Overall, the CPU usage seems great for this algorithm.

The CompletableFutures-based solution has a strong balance, it's fast, maintains low memory usage, and scales well. While GC activity increases with the workload, it doesn't seem to significantly impact performance, making this a solid option for large-scale parallel processing.

## Overall Performance Analysis

We compared four parallel strategies, manual thread creation, fixed-size ThreadPool , Fork–Join framework, and **CompletableFuture** solutions against a sequential baseline, across four data sizes (maxPages = 1000, 10000, 25000, 50000) on the machines described in Performance Analysis section. All parallel runs used the optimal thread count (12 or 16) determined by the earlier scaling experiments.

### Efficiency Gains

Best performances:

| **maxPages** | **Sequential (1 T)** | **Best Parallel (T)** | **Implementation** | **Speedup** |
|---|---|---|---|---|
| **1000** | 2446 ms | 264ms (12 T) | **ThreadPool** | **9.3x** |
| **10000** | 17289 ms | 4207ms (8 T) | **ThreadPool** | **4.1x** |
| **25000** | 43495 ms | 12054ms (8 T) | **ThreadPool** | **3.6x** |
| **50000** | 72262 ms | 12340ms (12 T) | **CompletableFuture** | **5.9x** |

*Table 10- Overall algorithms performances*

- **ThreadPool** delivers the largest absolute reductions (e.g., 2 182 ms → 264 ms on 1 000 pages; 43.5 s → 12.1 s on 25 000 pages).
- **CompletableFuture** excels on the smallest and largest workloads, achieving a 5.9× speedup at 50 000 pages by leveraging efficient task composition and minimal GC pressure

### Scalability

Thread Level Scalability - For 10000 pages:

| Threads | ThreadPool (ms) | CompletableFuture (ms) | Fork–Join (ms) | Manual Threads (ms) |
|---|---|---|---|---|
| 1 | 12015 | 8439 | 24112 | 12106 |
| 2 | 7321 (x1.64) | 5026 (x1.68) | 10096 (x2.39) | 6937 (x1.75) |
| 4 | 5987 (x2.01) | 4078 (x2.07) | 6947 (x3.47) | 5902 (x2.05) |
| 8 | 4361 (x2.75) | 4316 (x1.96) | 5875 (x4.10) | 5941 (x2.04) |
| 12 | 4361 (x2.75) | 4464 (x1.89) | 5369 (x4.49) | 5243 (x2.31) |
| 16 | 4408 (x2.73) | 4632 (x1.82) | 5501 (x4.38) | 5234 (x2.32) |

*Table 11-Thread level scalability (10000 pages)*

- **Ideal scaling** would halve time when doubling threads (i.e., a 2x speedup at 2 T, 4x at 4 T, etc.).
- **ThreadPool** achieves **2.75x** speedup at 8 T (versus ideal 8x), reflecting overheads and hardware limits.
- **CompletableFuture** scales similarly to ThreadPool up to 4 T, then yields diminishing returns beyond 8 T.
- **Fork/Join** and **Manual Threads** show strong speedup up to 4 T but slower gains, thereafter, indicating higher overhead per subtask and contention.

Data level Scalability – For 12 threads

| maxPages | ThreadPool (ms) | CompletableFuture (ms) | Fork–Join (ms) | Manual Threads (ms) |
|---|---|---|---|---|
| 1000 | 1287 | 435 | 685 | 613 |
| 10000 | 6048 (x4.7) | 4464 (x10.3) | 5369 (x7.8) | 5902 (x9.6) |
| 25000 | 12363 (x9.6) | 12484 (x28.7) | 16517 (x24.1) | 15254 (x24.9) |
| 50000 | 17797 (x13.8) | 12340 (x28.4) | 22853 (x33.4) | 18184 (x29.7) |

*Table 121- Data level scalability (12 threads)*

- **ThreadPool** shows sublinear growth: a 50x increase in pages yields only ~14x slower execution, showing that per-task overhead is amortized.
- **Fork–Join** and **Manual Threads** have higher growth factors (x24–33), indicating that repeated task-splitting and contention dominate as data grows.
- **CompletableFuture** shows sublinear scaling on large inputs (x28.4 for x50 data), though its absolute times remain very low.

**Performance conclusions**

- **Best Across All Sizes: ThreadPool** delivers the most consistent top performance from small to medium workloads, with low per-task overhead.
- **Best for Extremely Large Workloads: Fork/Join** slightly outperforms on the largest data set, thanks to its dynamic work-stealing.
- **CompletableFuture:** Clean interface and near **ThreadPool** performance.
- **Manual Threads:** Educational but inferior in every metric.

## Conclusion

Looking at all the data, we can draw the following conclusions:

- The sequential approach quickly becomes slow as workload increases, consuming high CPU for relatively low performance.
- Multithreaded solutions, especially with Thread Pool and CompletableFutures show strong performance, especially with 12 to 16 threads, where they maintain low CPU usage while still achieving fast execution times.
- ForkJoin performs reasonably well but tends to have higher CPU and GC overhead, especially with more threads, making it less efficient overall.
- Manual multithreading struggles to scale, often using more CPU than needed without matching performance gains.
- CompletableFutures stands out by having a great balanced delivering fast results, staying light on memory, keeping GC under control, and using CPU resources efficiently even as workloads grow.

In the end, for anything beyond small tasks, a well-tuned parallel solution is essential,not just for speed, but for making smarter use of system resources.

## References

Instituto Superior de Engenharia do Porto. (2024). *Sistemas Multinúcleo e Distribuídos - Lecture Slides*. Retrieved from Moodle - Course: DEI - Multicore and Distributed Systems - 2nd Semester, 2024/2025.

Instituto Superior de Engenharia do Porto. (2024). *Sistemas Multinúcleo e Distribuídos - TP*. Retrieved from Moodle - Course: DEI - Multicore and Distributed Systems - 2nd Semester, 2024/2025.

Instituto Superior de Engenharia do Porto. (2024). *Sistemas Multinúcleo e Distribuídos - PLs*. Retrieved from Moodle - Course: DEI - Multicore and Distributed Systems - 2nd Semester, 2024/2025.

# Attachments

## Multithreaded Solution (Without Thread Pools)

### 1000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 1000 | 1 | 2375.19 | 32.95 | 15 | 104 | 243.13 |
| MultithreadedManual | 1000 | 2 | 1654.02 | 36.19 | 10 | 112 | 117.78 |
| MultithreadedManual | 1000 | 4 | 881.63 | 0.88 | 7 | 78 | 64.61 |
| MultithreadedManual | 1000 | 8 | 660.87 | 0.16 | 7 | 78 | 41.42 |
| MultithreadedManual | 1000 | 12 | 613.7 | 0.16 | 7 | 80 | 35.43 |
| MultithreadedManual | 1000 | 16 | 620.4 | 0.16 | 6 | 76 | 28.79 |

**Table 1 - Multithreaded Solution (Without Thread Pools )results with 1000 max pages**

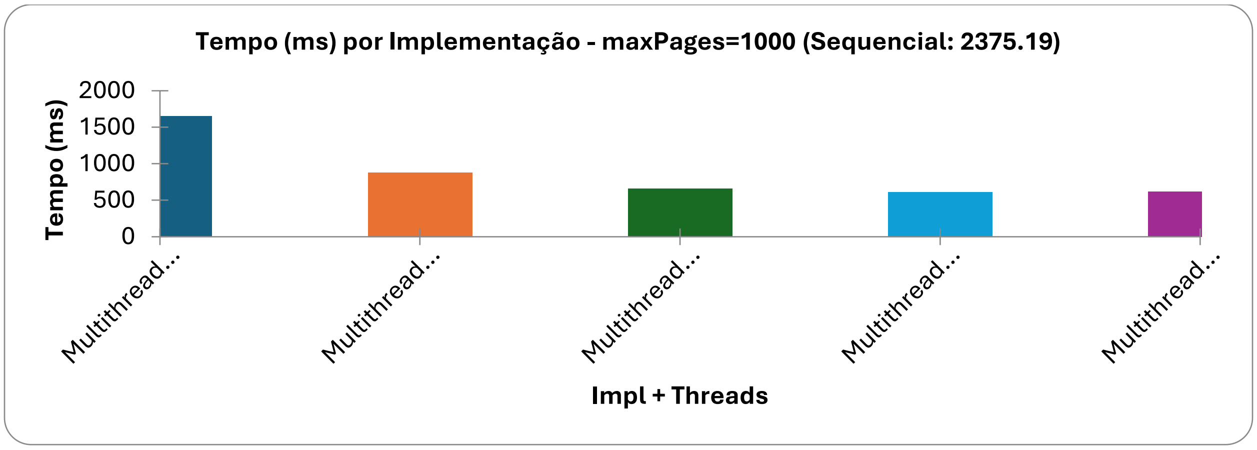


**Figure 1 - Multithreaded Solution (Without Thread Pools ) results with 1000 max pages**

### 10,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 10000 | 1 | 20836.65 | 144.25 | 37 | 570 | 124.8 |
| MultithreadedManual | 10000 | 2 | 12106.64 | 167.24 | 33 | 774 | 120.87 |
| MultithreadedManual | 10000 | 4 | 8752.65 | 0 | 20 | 467 | 79.92 |
| MultithreadedManual | 10000 | 8 | 6393.42 | 0.03 | 22 | 461 | 58.94 |
| MultithreadedManual | 10000 | 12 | 6278.16 | 0 | 22 | 443 | 49.62 |
| MultithreadedManual | 10000 | 16 | 6459.2 | 0.39 | 19 | 535 | 36.47 |

**Table 2 - Multithreaded Solution (Without Thread Pools ) results with 10000 max pages**

## 25,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 25000 | 1 | 53498.53 | 190.31 | 55 | 1162 | 130.75 |
| MultithreadedManual | 25000 | 2 | 31366.68 | 221.7 | 50 | 1806 | 127.81 |
| MultithreadedManual | 25000 | 4 | 21967.57 | 0 | 47 | 974 | 94.18 |
| MultithreadedManual | 25000 | 8 | 16528.15 | 0 | 46 | 950 | 73.67 |
| MultithreadedManual | 25000 | 12 | 15846.91 | 0 | 45 | 928 | 57.39 |
| MultithreadedManual | 25000 | 16 | 15253.73 | 0.03 | 43 | 995 | 41.92 |

**Table 3 - Multithreaded Solution (Without Thread Pools ) results with 25000 max pages**

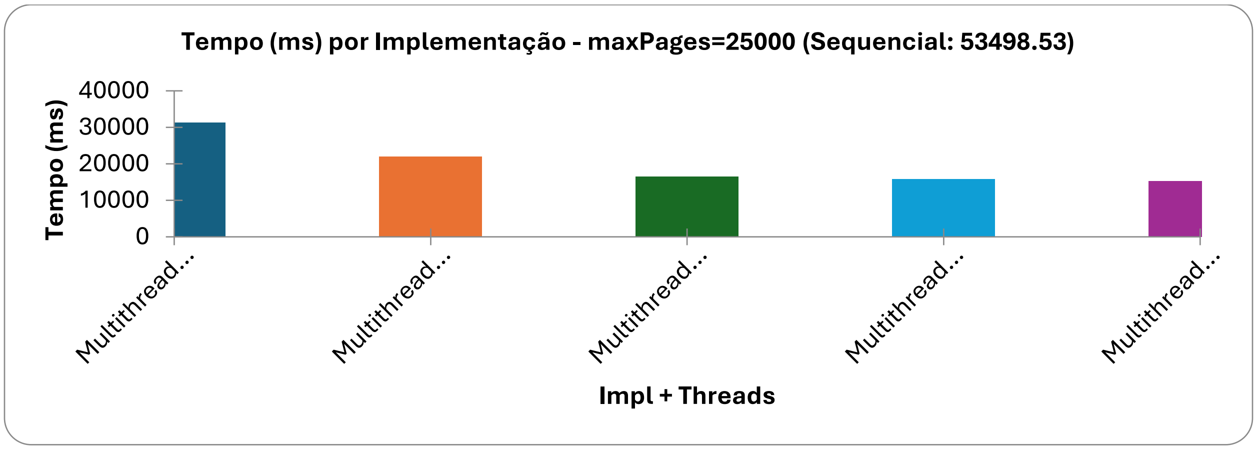


**Figure 2 - Multithreaded Solution (Without Thread Pools ) results with 25000 max pages**

## 50,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 50000 | 1 | 56263.67 | 31.96 | 45 | 905 | 138.14 |
| MultithreadedManual | 50000 | 2 | 33532.32 | 36.62 | 64 | 1283 | 126.44 |
| MultithreadedManual | 50000 | 4 | 23873.77 | 0 | 65 | 1061 | 96.48 |
| MultithreadedManual | 50000 | 8 | 19380.71 | 0 | 56 | 1063 | 72.39 |
| MultithreadedManual | 50000 | 12 | 18184.71 | 0 | 55 | 1129 | 56.97 |
| MultithreadedManual | 50000 | 16 | 19242.92 | 0.05 | 55 | 1383 | 40.92 |

**Table 4 - Multithreaded Solution (Without Thread Pools ) results with 50000 max pages**

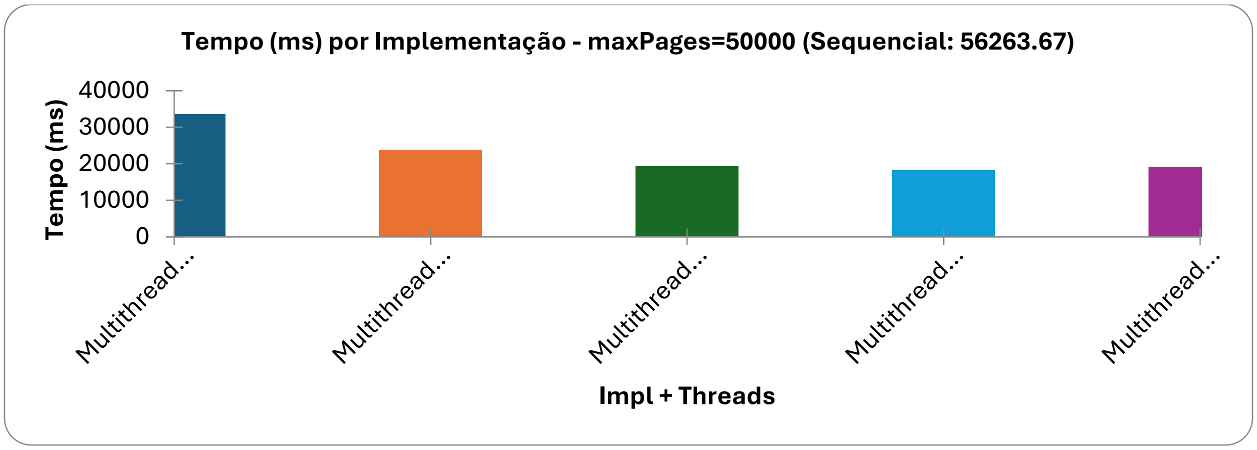


**Figure 3 - Multithreaded Solution (Without Thread Pools ) results with 50000 max pages**

### CPU usage chart (12 threads – 50,000 pages)

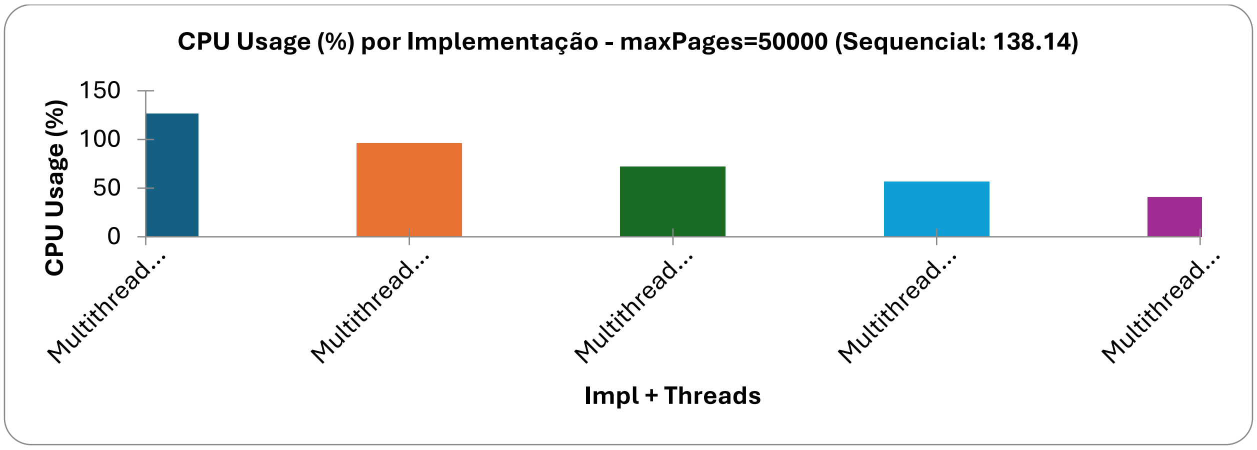


**Figure 4 - Multithreaded Solution (Without Thread Pools – 12 threads) CPU usage with 50000 max pages**

## Multithreaded Solution (With Thread Pools)

### 1000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 1000 | 1 | 2742.7 | 33 | 15 | 106 | 217.88 |
| ThreadPool | 1000 | 2 | 1529.01 | 35.87 | 10 | 104 | 128.48 |
| ThreadPool | 1000 | 4 | 1317.86 | 0.15 | 7 | 77 | 48.61 |
| ThreadPool | 1000 | 8 | 1265.69 | 0.16 | 7 | 78 | 22.96 |
| ThreadPool | 1000 | 12 | 1259.74 | 0.15 | 7 | 78 | 15 |
| ThreadPool | 1000 | 16 | 1287.39 | 0.16 | 7 | 79 | 12.03 |

**Table 5 - Multithreaded Solution (With Thread Pools ) results with 1000 max pages**

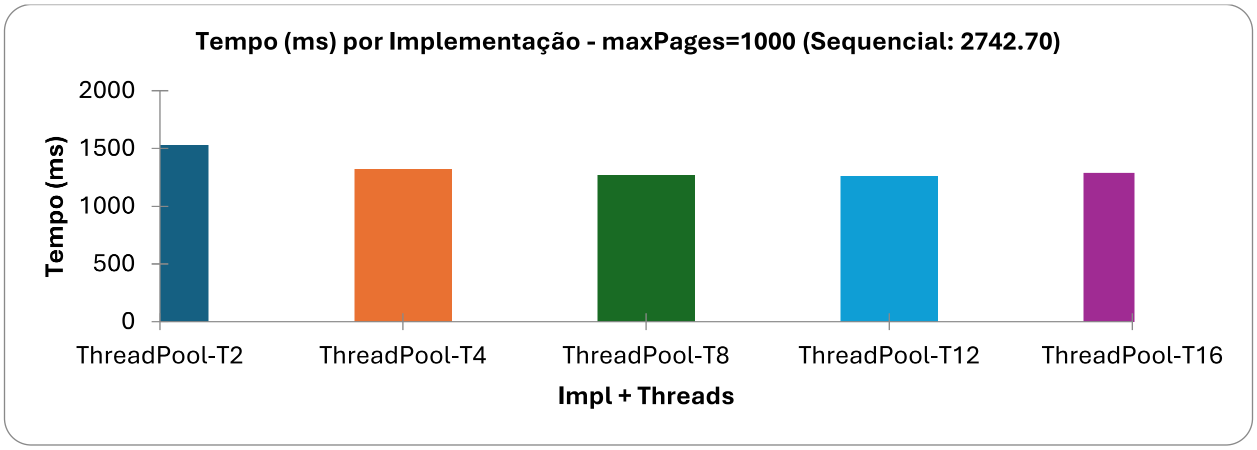


**Figure 5 - Multithreaded Solution (With Thread Pools ) results with 1000 max pages**

## 10,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 10000 | 1 | 23033.59 | 144.12 | 43 | 559 | 119.28 |
| ThreadPool | 10000 | 2 | 12015.2 | 167.75 | 31 | 646 | 132.61 |
| ThreadPool | 10000 | 4 | 7321.33 | 0.51 | 26 | 427 | 111.89 |
| ThreadPool | 10000 | 8 | 5987.21 | 0.7 | 21 | 428 | 86.11 |
| ThreadPool | 10000 | 12 | 6048.84 | 0.62 | 21 | 430 | 60.85 |
| ThreadPool | 10000 | 16 | 6361.22 | 0.64 | 21 | 438 | 47.66 |

**Table 6 - Multithreaded Solution (With Thread Pools ) results with 10000 max pages**

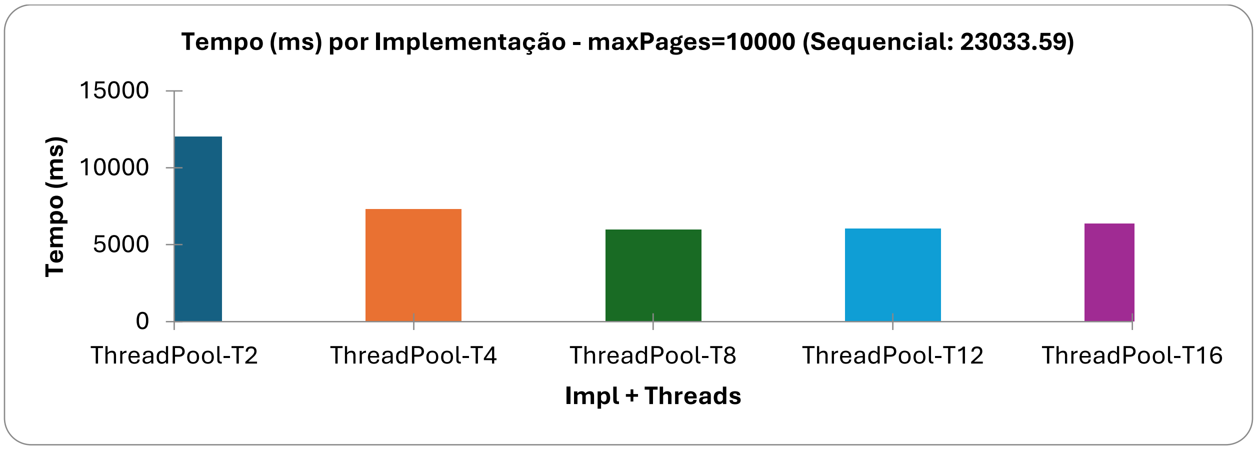


**Figure 6 - Multithreaded Solution (With Thread Pools ) results with 10000 max pages**

## 25,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 25000 | 1 | 59382.37 | 190.41 | 79 | 1140 | 140.43 |
| ThreadPool | 25000 | 2 | 28428.04 | 220.85 | 37 | 1260 | 140.09 |
| ThreadPool | 25000 | 4 | 18275.38 | 0.01 | 32 | 953 | 125.06 |
| ThreadPool | 25000 | 8 | 15595.7 | 0.44 | 33 | 955 | 106.84 |
| ThreadPool | 25000 | 12 | 15710.73 | 0.37 | 39 | 965 | 76.09 |
| ThreadPool | 25000 | 16 | 15275.2 | 0.26 | 38 | 949 | 58.71 |

**Table 7 - Multithreaded Solution (With Thread Pools ) results with 25000 max pages**

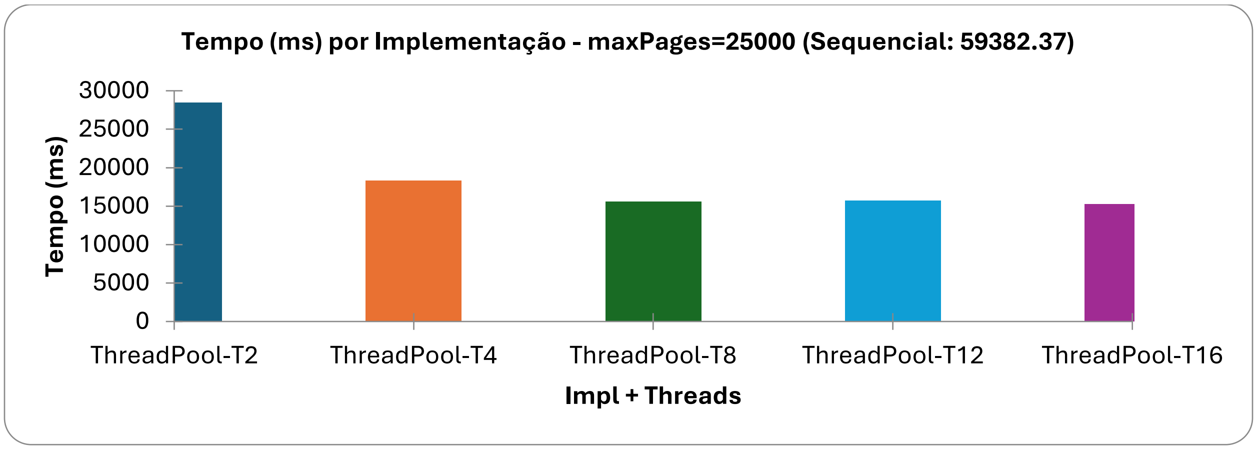


**Figure 7 - Multithreaded Solution (With Thread Pools ) results with 25000 max pages**

## 50,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 50000 | 1 | 65838.01 | 31.88 | 21 | 792 | 107.48 |
| ThreadPool | 50000 | 2 | 32148.55 | 36.59 | 38 | 1103 | 137.72 |
| ThreadPool | 50000 | 4 | 20342.38 | 0 | 37 | 1054 | 126.14 |
| ThreadPool | 50000 | 8 | 17824.88 | 0 | 39 | 1095 | 110.96 |
| ThreadPool | 50000 | 12 | 17796.77 | 0 | 41 | 1097 | 81.14 |
| ThreadPool | 50000 | 16 | 17514.95 | 0 | 42 | 1103 | 60.33 |

**Table 8 - Multithreaded Solution (With Thread Pools ) results with 50000 max pages**

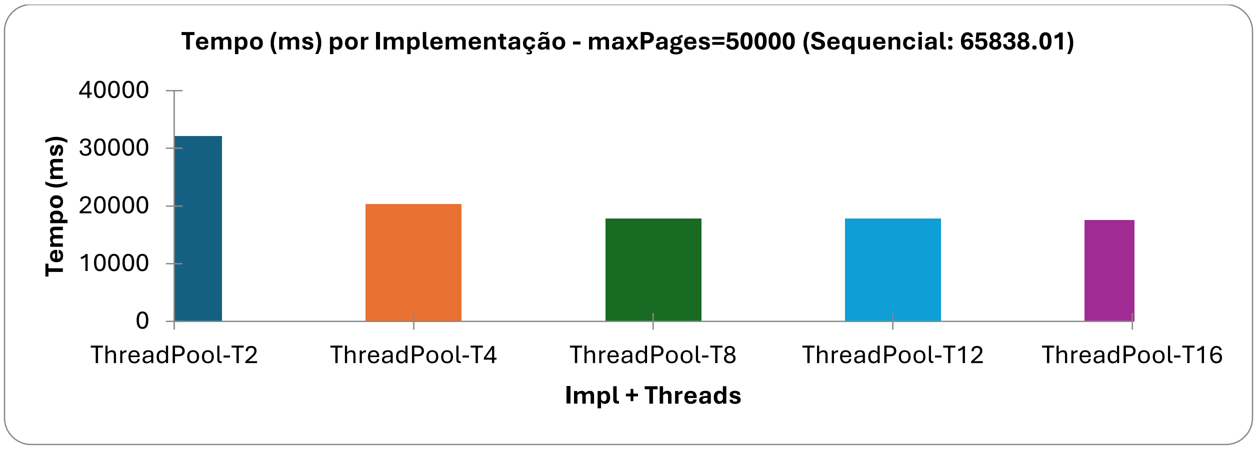


**Figure 8 - Multithreaded Solution (With Thread Pools ) results with 50000 max pages**

### CPU usage chart (12 threads – 50,000 pages)

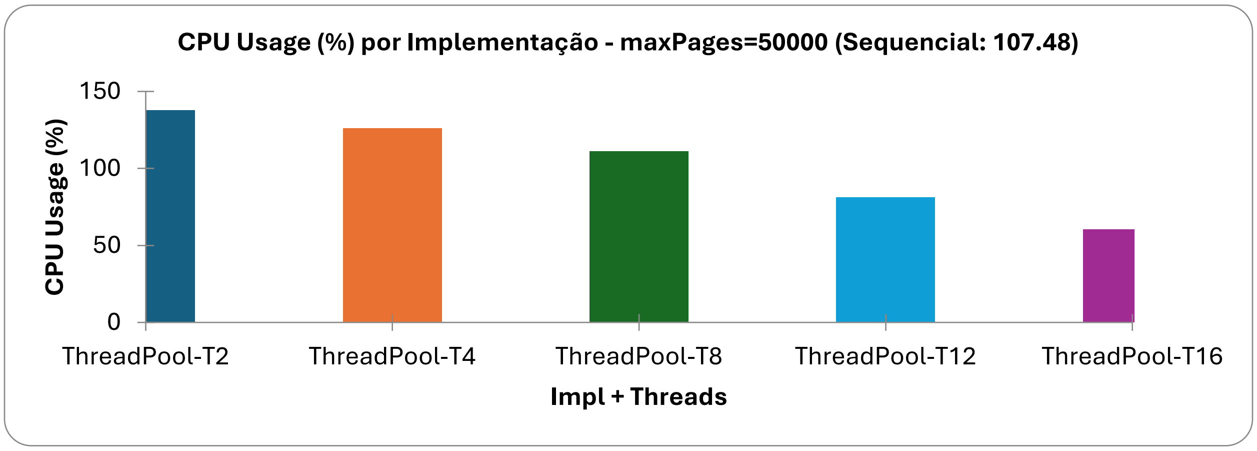


**Figure 9 - Multithreaded Solution (With Thread Pools – 12 threads ) CPU usage with 50000 max pages**

## Fork/Join Solution

### 1000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 1000 | 1 | 3489.75 | 32.96 | 15 | 134 | 197.35 |
| ForkJoin | 1000 | 2 | 1755.26 | 35.86 | 10 | 116 | 117.18 |
| ForkJoin | 1000 | 4 | 894.69 | 0.16 | 7 | 74 | 67.92 |
| ForkJoin | 1000 | 8 | 712.5 | 0.32 | 7 | 80 | 50.18 |
| ForkJoin | 1000 | 12 | 685.92 | 0.16 | 7 | 76 | 42.73 |
| ForkJoin | 1000 | 16 | 671.25 | 0.16 | 7 | 75 | 37.56 |

**Table 9 – Fork/Join solution results with 1000 max pages**

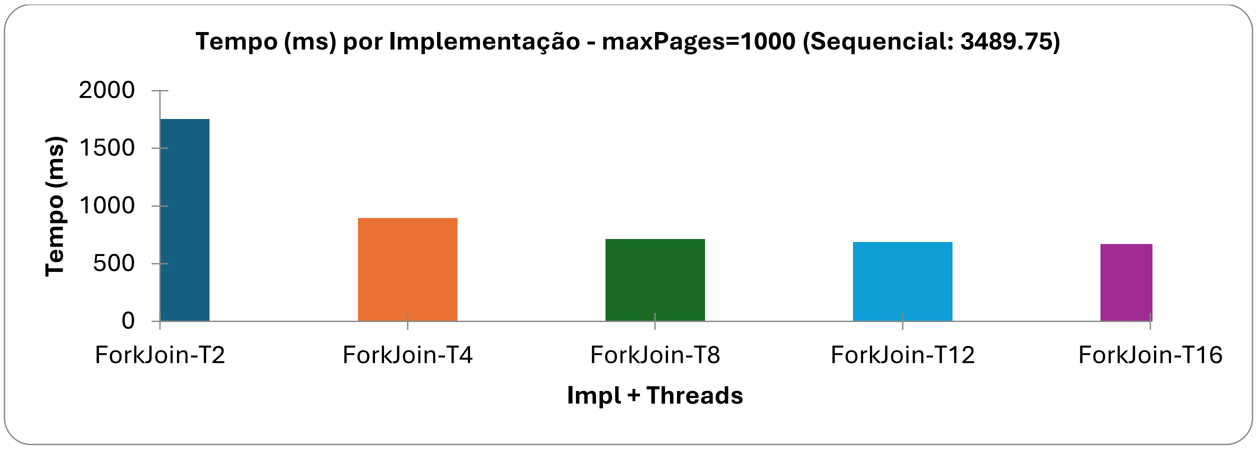


**Figure 10 – Fork/Join solution results with 1000 max pages**

## 10,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 10000 | 1 | 24211.03 | 143.98 | 44 | 570 | 120.1 |
| ForkJoin | 10000 | 2 | 14113.83 | 167.64 | 36 | 718 | 115.51 |
| ForkJoin | 10000 | 4 | 8897.21 | 0.03 | 21 | 412 | 84.15 |
| ForkJoin | 10000 | 8 | 7791.78 | 0.36 | 22 | 461 | 63.91 |
| ForkJoin | 10000 | 12 | 7178.15 | 0 | 21 | 459 | 54.3 |
| ForkJoin | 10000 | 16 | 7353.59 | 0.03 | 22 | 437 | 44.6 |

**Table 10 – Fork/Join solution results with 10000 max pages**

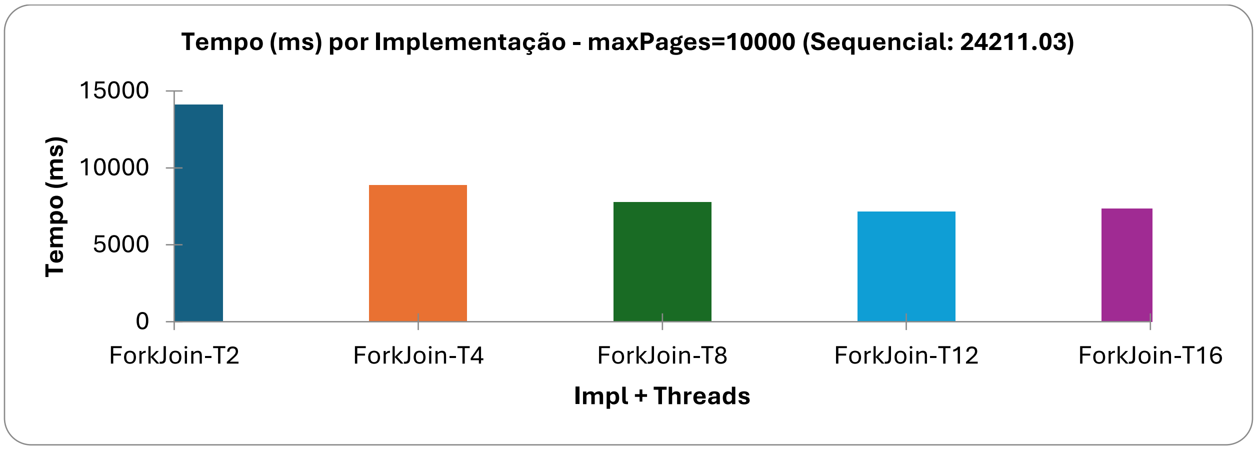


**Figure 11 – Fork/Join solution results with 10000 max pages**

## 25,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 25000 | 1 | 62032.69 | 190.77 | 57 | 1123 | 125.39 |
| ForkJoin | 25000 | 2 | 35851.49 | 221.91 | 50 | 1526 | 119.82 |
| ForkJoin | 25000 | 4 | 24055.56 | 0 | 43 | 948 | 96.25 |
| ForkJoin | 25000 | 8 | 21040.15 | 0.03 | 44 | 954 | 74.8 |
| ForkJoin | 25000 | 12 | 19615.72 | 0 | 42 | 953 | 62.79 |
| ForkJoin | 25000 | 16 | 19516.35 | 0 | 44 | 1014 | 48.65 |

**Table 11 – Fork/Join solution results with 25000 max pages**

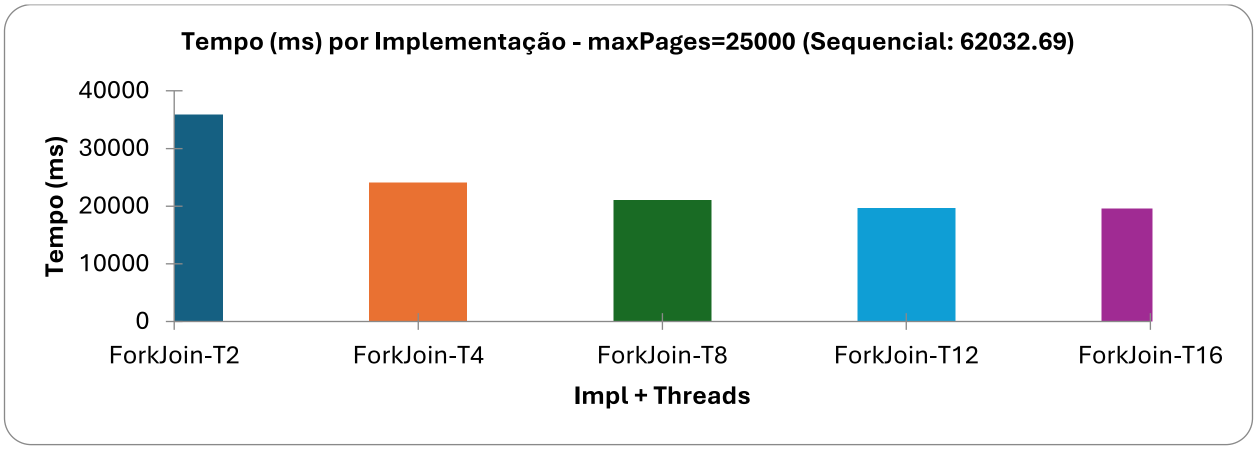


**Figure 12 – Fork/Join solution results with 25000 max pages**

## 50,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 50000 | 1 | 72262.06 | 32.12 | 21 | 906 | 108.03 |
| ForkJoin | 50000 | 2 | 49120.42 | 36.59 | 62 | 1310 | 122.7 |
| ForkJoin | 50000 | 4 | 28887.21 | 0 | 58 | 1063 | 99.69 |
| ForkJoin | 50000 | 8 | 23259.79 | 0.03 | 58 | 1092 | 78.59 |
| ForkJoin | 50000 | 12 | 22853.4 | 0.03 | 59 | 1090 | 64.9 |
| ForkJoin | 50000 | 16 | 23337.61 | 0 | 56 | 1167 | 47.86 |

**Table 12 – Fork/Join solution results with 50000 max pages**

### CPU usage chart (12 threads – 50000 pages)

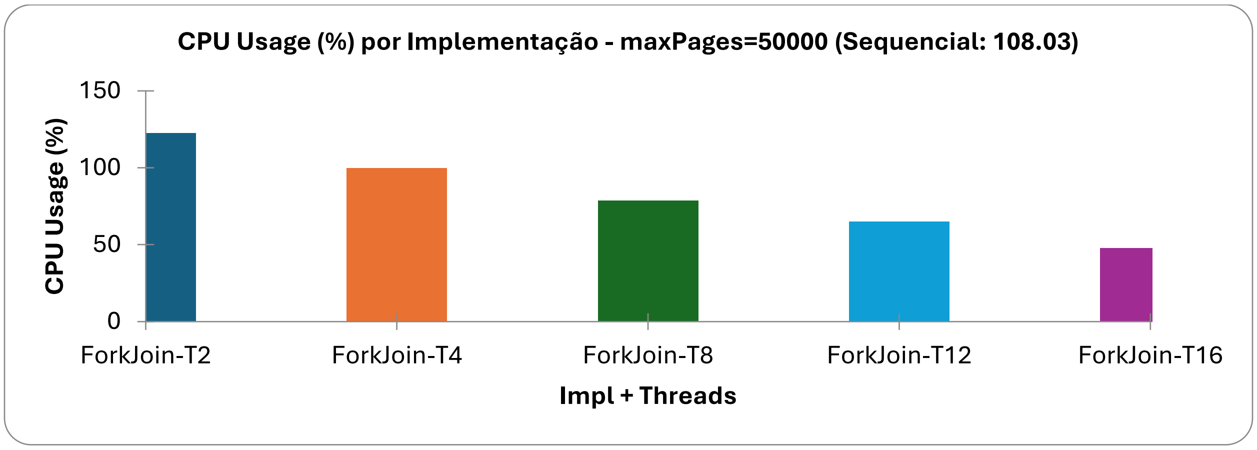


**Figure 13 – Fork/Join solution results with 50000 max pages**

## Completable Futures Solution

### 1000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 1000 | 1 | 2325.75 | 33.39 | 15 | 105 | 227.07 |
| CompletableFuturesBasedSolutio | 1000 | 2 | 1138.79 | 35.83 | 10 | 108 | 168 |
| CompletableFuturesBasedSolutio | 1000 | 4 | 479.95 | 0.15 | 6 | 63 | 97.77 |
| CompletableFuturesBasedSolutio | 1000 | 8 | 427.5 | 0.15 | 6 | 64 | 59.09 |
| CompletableFuturesBasedSolutio | 1000 | 12 | 435.73 | 0.14 | 6 | 63 | 38.88 |
| CompletableFuturesBasedSolutio | 1000 | 16 | 439.71 | 0.21 | 7 | 62 | 30.92 |

**Table 13 – Completable Futures solution results with 1000 max pages**

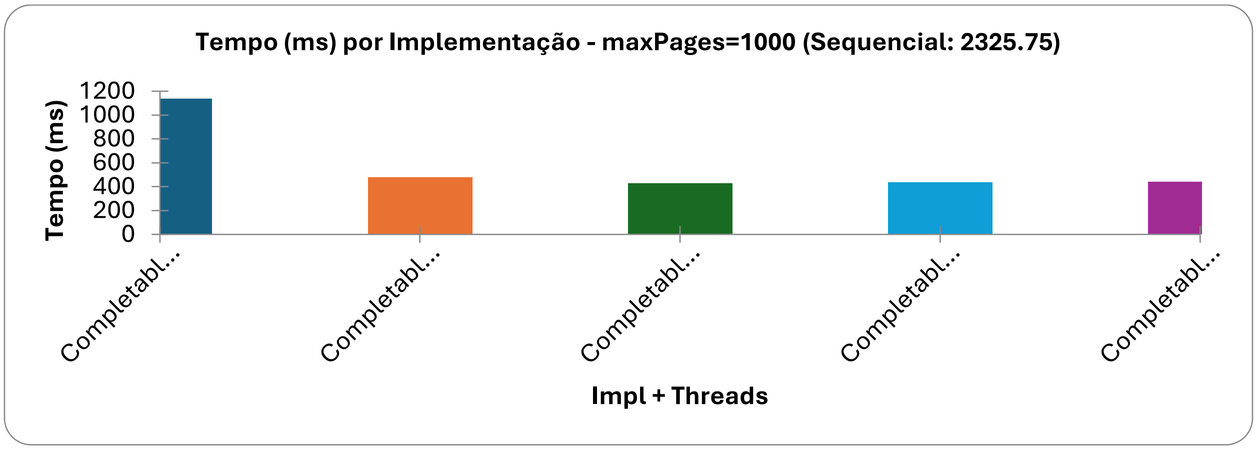


**Figure 14 – Completable Futures solution results with 1000 max pages**

## 10,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 10000 | 1 | 17949.45 | 143.88 | 36 | 516 | 121.73 |
| CompletableFuturesBasedSolutio | 10000 | 2 | 8439.17 | 167.31 | 31 | 678 | 157.76 |
| CompletableFuturesBasedSolutio | 10000 | 4 | 5026.59 | 0.38 | 14 | 397 | 116.36 |
| CompletableFuturesBasedSolutio | 10000 | 8 | 4078.09 | 0.05 | 14 | 269 | 82.96 |
| CompletableFuturesBasedSolutio | 10000 | 12 | 4316.03 | 0.36 | 14 | 265 | 53.97 |
| CompletableFuturesBasedSolutio | 10000 | 16 | 4642.57 | 0.38 | 13 | 269 | 40.6 |

**Table 14 – Completable Futures solution results with 10000 max pages**

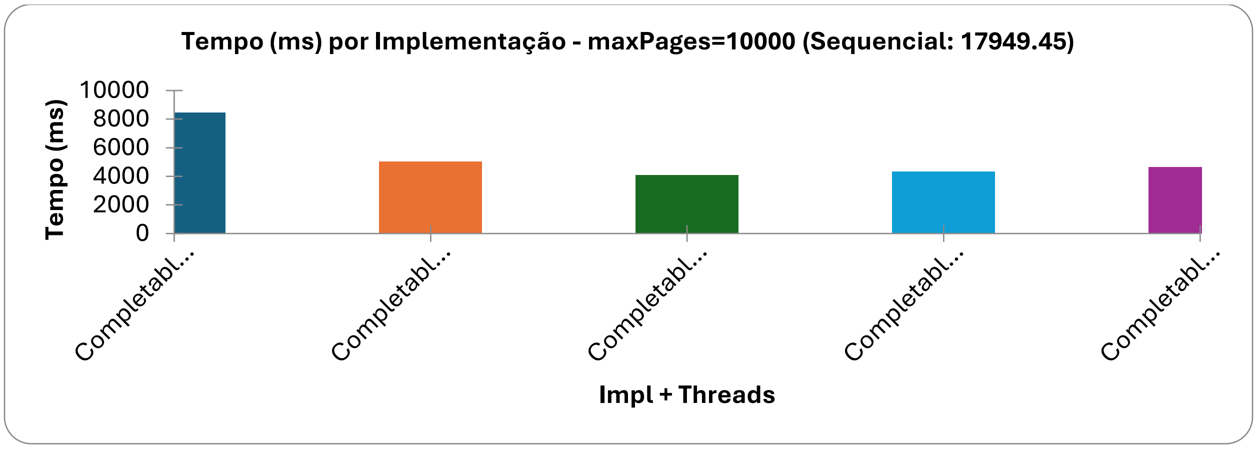


**Figure 15 – Completable Futures solution results with 10000 max pages**

## 25,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 25000 | 1 | 47804.16 | 190.59 | 49 | 1087 | 128.73 |
| CompletableFuturesBasedSolutio | 25000 | 2 | 21173.99 | 221.66 | 34 | 1274 | 147.43 |
| CompletableFuturesBasedSolutio | 25000 | 4 | 13031.66 | 0.01 | 26 | 884 | 122.69 |
| CompletableFuturesBasedSolutio | 25000 | 8 | 13647.61 | 0.5 | 16 | 632 | 82.82 |
| CompletableFuturesBasedSolutio | 25000 | 12 | 12848.03 | 0.5 | 16 | 637 | 53.96 |
| CompletableFuturesBasedSolutio | 25000 | 16 | 13109.7 | 0.53 | 16 | 651 | 41.24 |

**Table 15 – Completable Futures solution results with 25000 max pages**

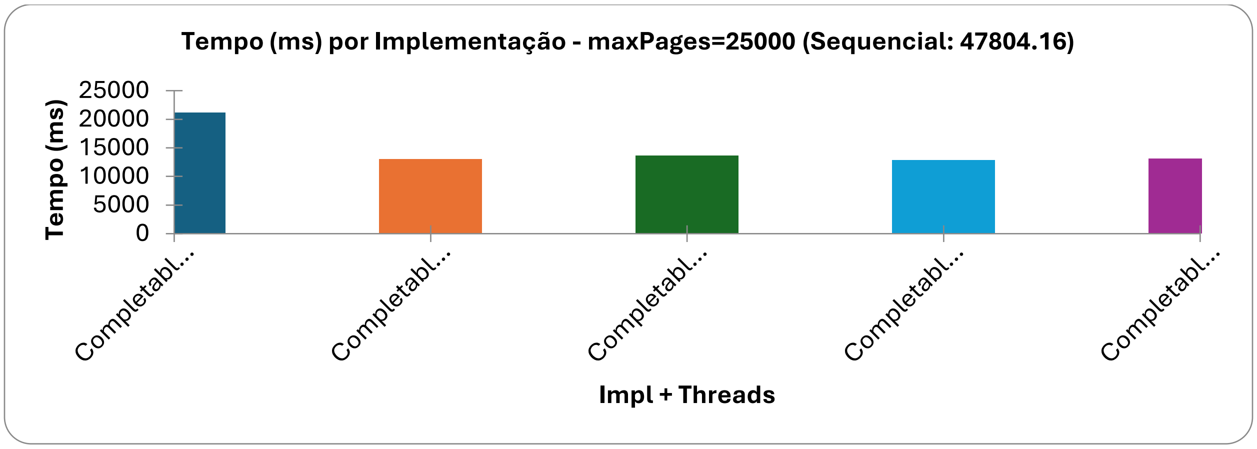


**Figure 16 – Completable Futures solution results with 25000 max pages**

## 50,000 pages report

| Implementação | maxPages | Threads | Tempo (ms) | Memória (MB) | GC Count | GC Time (ms) | CPU Usage (%) |
|---|---|---|---|---|---|---|---|
| Sequencial | 50000 | 1 | 57297.88 | 32.13 | 24 | 932 | 113.82 |
| CompletableFuturesBasedSolutio | 50000 | 2 | 23763.9 | 36.61 | 39 | 1068 | 148.7 |
| CompletableFuturesBasedSolutio | 50000 | 4 | 15741.98 | 0.03 | 32 | 1022 | 133.78 |
| CompletableFuturesBasedSolutio | 50000 | 8 | 12830.52 | 0 | 22 | 675 | 91.85 |
| CompletableFuturesBasedSolutio | 50000 | 12 | 12339.62 | 0 | 19 | 602 | 54.48 |
| CompletableFuturesBasedSolutio | 50000 | 16 | 12633.7 | 0 | 19 | 602 | 41.06 |

**Table 16 – Completable Futures solution results with 50000 max pages**

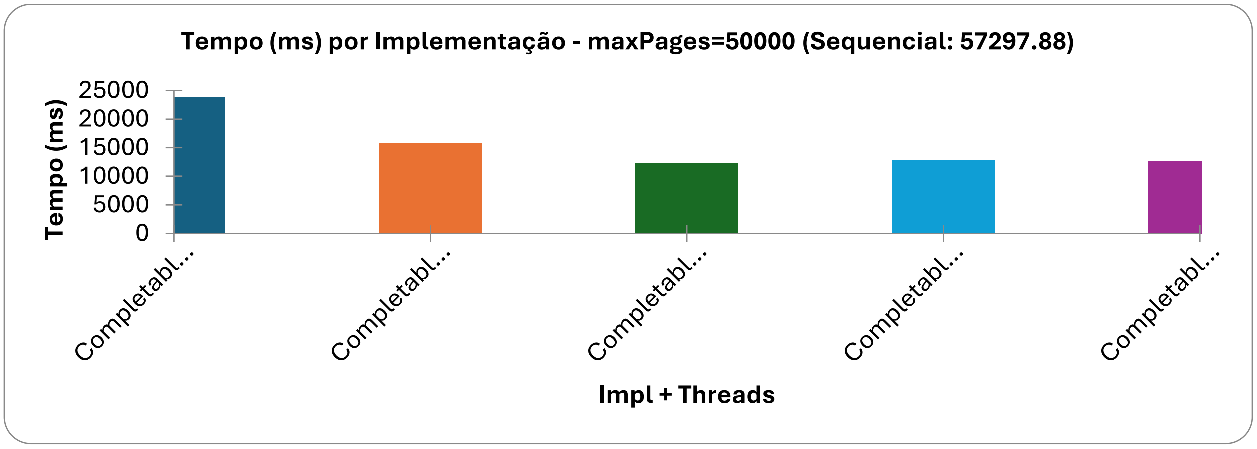


**Figure 17 – Completable Futures solution results with 50000 max pages**

## CPU usage chart (12 threads – 50000 pages)

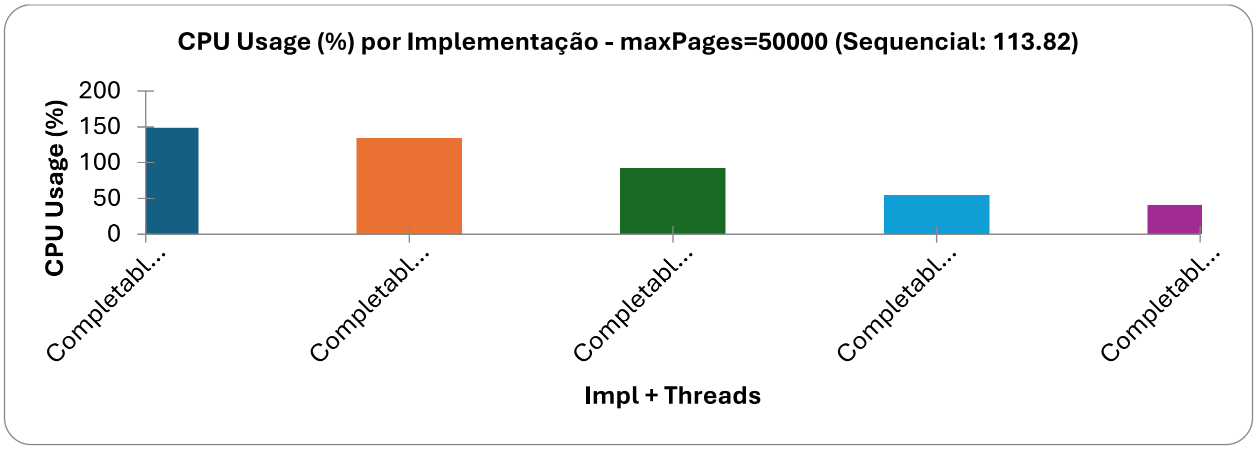


**Figure 18 – Completable Futures solution (12 threads) CPU usage with 50000 max pages**