Optimizing Large-Scale Data Processing on Multicore Systems

Project Information

Course: SISMD

Institution: ISEP – Instituto Superior de Engenharia do Porto

Academic Year: 2024/2025

Authors: José Soares (1190782), Alexandre Monteiro (1230164) and Francisco Borges (1201477)

Introduction

As the world becomes even more data-centric, processing high levels of information becomes increasingly a critical problem. With commodity multicore processors now widely available, we must transcend conventional sequential methods and study concurrent and parallel methods to maximize current hardware capabilities.

This work is about optimizing the processing of an extensive XML data dump from the English Wikipedia to calculate the frequency of words on thousands of pages. Instead of implementing simply one solution, this research investigates and compares various programming models from the simple sequential one to different multithreaded ones in order to better comprehend their influence on performance, resource efficiency and scalability.

By joining practical implementation with performance analysis, the project delivers not just a technical solution but valuable knowledge about trade-offs and benefits of concurrent programming for real-world applications.

Objectives

The main goal of this project wasn't just to count how many times words appear in a massive Wikipedia data file but it was to understand how different approaches to parallel and concurrent programming can make that process faster and more efficient.

We started with a simple sequential solution, just to set a baseline. From there, the challenge was to explore and implement multiple ways of speeding things up by taking advantage of multicore systems. Each method from manually managing threads to using thread pools, Fork/Join and even CompletableFutures gave us new insights into what works best in which context and why.

Beyond writing code, this project was really about learning how to think in parallel, to break down problems in a way that computers with multiple cores can actually benefit from. It was also about comparing results, tuning performance and understanding the trade-offs between simplicity, control and scalability.

Implementation Approaches

To really understand how concurrency affects performance, we didn't just stick to one solution. Starting from a basic sequential version, we gradually introduced different levels of parallelism, each with its own style of managing tasks and threads.

Some approaches gave us full control (and more responsibility), while others handled the hard parts for us. In this section, we'll walk through what we tried, how each method works and what we learned from building and testing them.

1. Common

Before implementing any of the specific solutions, sequential or concurrent, we created an essential set of utility classes in the common package. These classes were already developed in the StartCode.zip, that was the initial code developed by the teacher.

The goal here to make the code more readable and to be easier to build each solution independently.

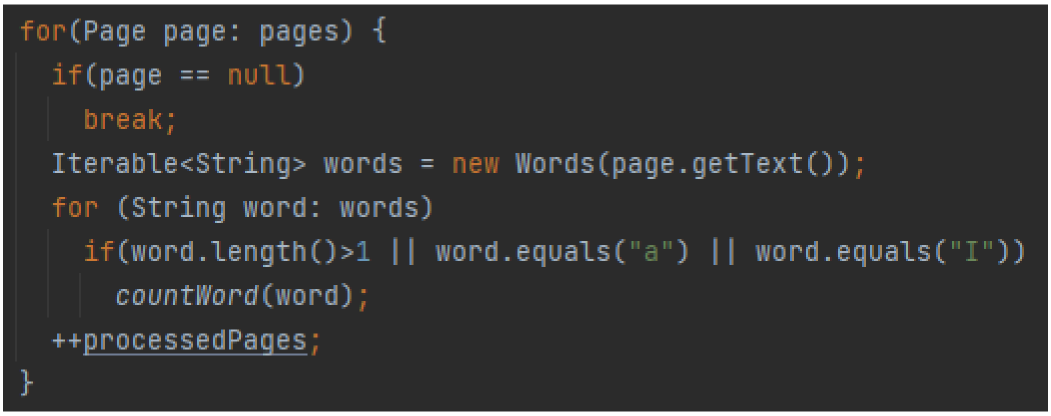
2. Sequential Solution

This version is our baseline. it does everything in one thread, from start to finish. The program reads each page from the XML dump, extracts the text, splits it into words and updates a frequency map.

There's no parallelism, no synchronization and no sharing of data between threads. That simplicity makes it easy to implement and debug, but it also means performance is limited, especially as the size of the dataset grows.

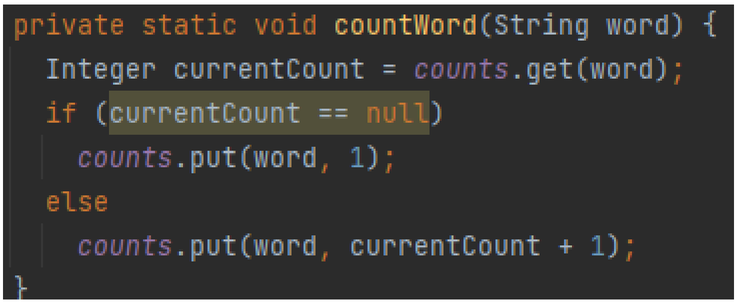
This version is essential because it sets a reference point to understand what we will be the main differences to the others solutions.

In terms of code, the logic is direct. We use the Pages class to read one Wikipedia page at a time and for each page we use Words to extract individual words. Then we filter out like short tokens and count the valid words using a simple HashMap.



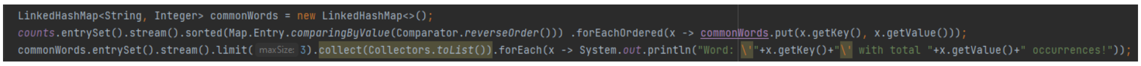
```
for(Page page: pages) {
  if(page == null)
    break;
  Iterable<String> words = new Words(page.getText());
  for (String word: words)
    if(word.length()>1 || word.equals("a") || word.equals("I"))
      countWord(word);
  ++processedPages;
}
```

For each page we extract the text, split it into words using the Words iterable and filter out very short tokens (except "a" and "I", which are valid words). Then we update the count in the map. Everything runs sequentially in a single thread.



```
private static void countWord(String word) {
  Integer currentCount = counts.get(word);
  if (currentCount == null)
    counts.put(word, 1);
  else
    counts.put(word, currentCount + 1);
}
```

This helper method either adds a new word to the map or increments its count.



```
LinkedHashMap<String, Integer> commonWords = new LinkedHashMap<>();
counts.entrySet().stream().sorted(Map.Entry.comparingByValue(Comparator.reverseOrder())) .forEachOrdered(x -> commonWords.put(x.getKey(), x.getValue()));
commonWords.entrySet().stream().limit( maxSize: 3).collect(Collectors.toList()).forEach(x -> System.out.println("Word: \'"+x.getKey()+"\' with total "+x.getValue()+" occurrences!"));
```

After processing, we sort the word counts in descending order using Java Streams and print the top 3 most frequent words. It’s a neat use of functional-style operations for presentation.

3. Multithreaded Solution (Manual Threads)

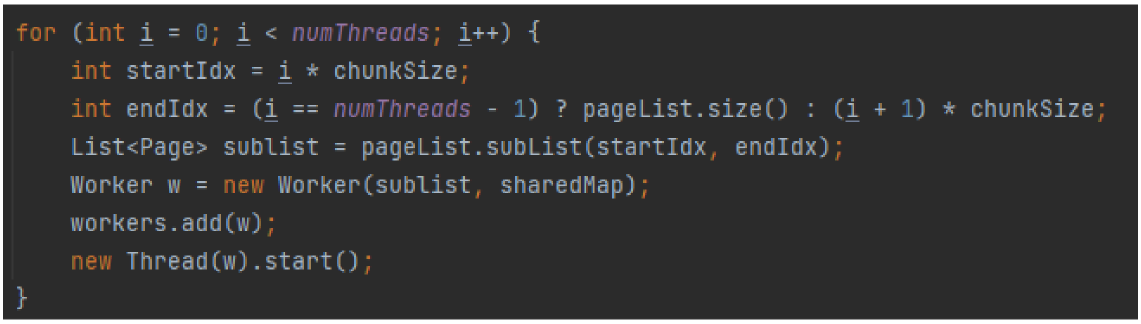
This was our first step into parallelism. Instead of doing everything in one thread, we split the workload across several threads, each responsible for processing part of the data.

We managed the threads manually wihthout thread pools, just Thread objects. Each thread processed a bit of pages and updated a shared word count map. To keep things safe, we used a ConcurrentHashMap.

To parallelize the work, we created a Worker class that implements Runnable. Each worker is responsible for processing a part of the Wikipedia pages. It loops through the pages it receives, extracts the words, filters them and updates the shared frequency map.

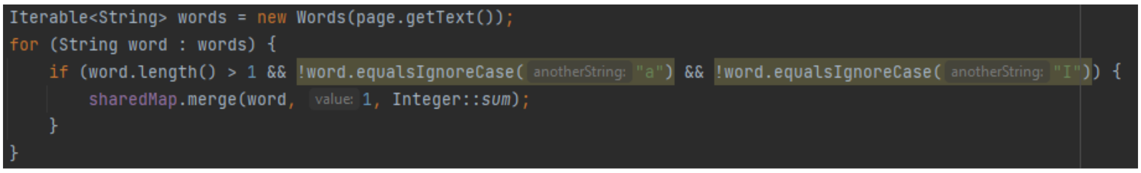
What makes this setup parallel is how we use multiple Worker instances, each running in its own thread.

In the main method, after loading all the pages, we divide the list into equal parts and assign each part to a new Worker. Then, we create a new Thread and start it:



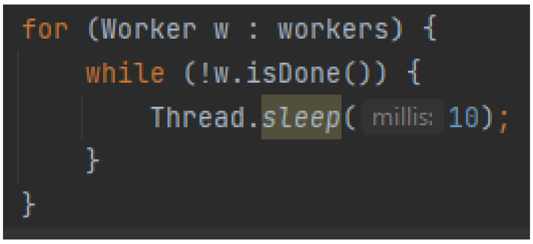
```
for (int i = 0; i < numThreads; i++) {
    int startIdx = i * chunkSize;
    int endIdx = (i == numThreads - 1) ? pageList.size() : (i + 1) * chunkSize;
    List<Page> sublist = pageList.subList(startIdx, endIdx);
    Worker w = new Worker(sublist, sharedMap);
    workers.add(w);
    new Thread(w).start();
}
```

By doing this for every chunk, we launch multiple threads in parallel . each one running its own worker and updating the shared map. Inside the Worker, we used ConcurrentHashMap.merge() to safely update word counts without needing explicit locks:



```
Iterable<String> words = new Words(page.getText());
for (String word : words) {
    if (word.length() > 1 && !word.equalsIgnoreCase( anotherString: "a") && !word.equalsIgnoreCase( anotherString: "I")) {
        sharedMap.merge(word,  value: 1, Integer::sum);
    }
}
```

Finally, each worker sets a done flag to true when it finishes. In the main thread, we simply wait for all workers to complete by checking this flag in a loop:



```
for (Worker w : workers) {
    while (!w.isDone()) {
        Thread.sleep( millis: 10);
    }
}
```

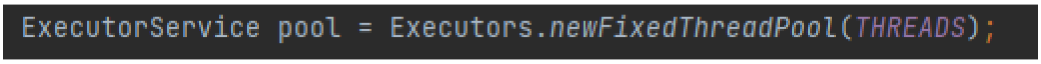
4. Multithreaded Solution (Thread Pools)

After manually managing threads we decided to simplify things by using a thread pool. With an ExecutorService, we can reuse a fixed number of threads instead of creating new ones for each task. This reduces overhead and makes the program easier to manage and scale.

We split the pages into small batches (10 pages per task) and submitted each batch to the pool. This batching helps balance the work across threads and keeps all cores busy.

To safely count words across threads we used a ConcurrentHashMap combined with AtomicInteger. This allows multiple threads to increment word counts at the same time without needing explicit synchronization.

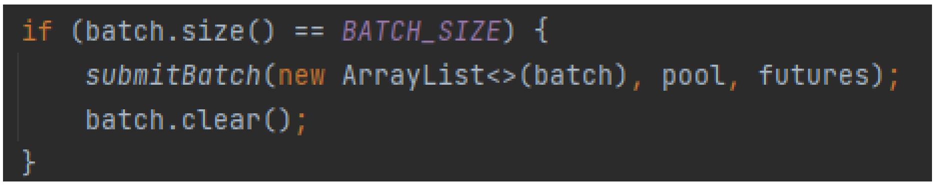
We start by defining the number of threads in the pool using the number of available processors:



```
ExecutorService pool = Executors.newFixedThreadPool(THREADS);
```

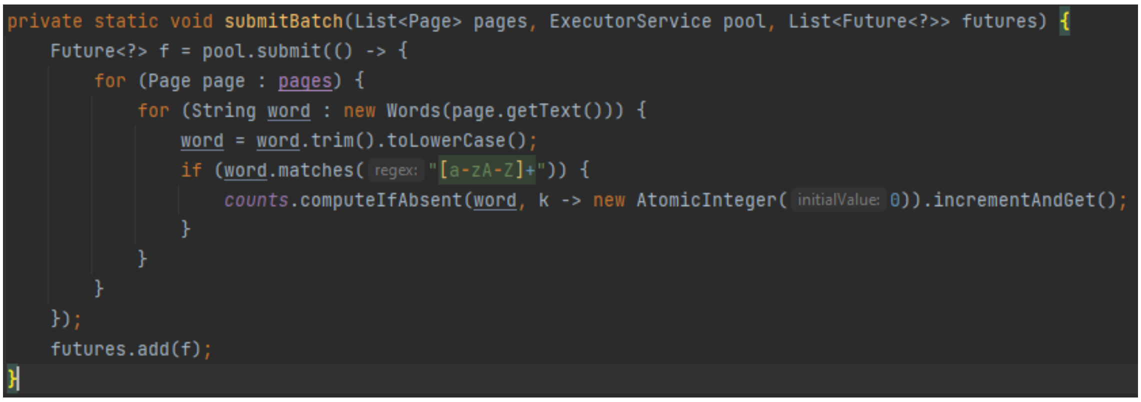
This way the system decides how many threads make sense based on the machine's CPU keeping things efficient.

Instead of giving each thread a huge chunk of pages, we split the work into **small batches** (10 pages at a time). Each batch becomes a task submitted to the pool:



```
if (batch.size() == BATCH_SIZE) {
    submitBatch(new ArrayList<>(batch), pool, futures);
    batch.clear();
}
```

Each task is handled by a lambda function inside submitBatch. It loops through the pages and counts words using a shared map:



```
private static void submitBatch(List<Page> pages, ExecutorService pool, List<Future<?>> futures) {
    Future<?> f = pool.submit(() -> {
        for (Page page : pages) {
            for (String word : new Words(page.getText())) {
                word = word.trim().toLowerCase();
                if (word.matches( regex: "[a-zA-Z]+")) {
                    counts.computeIfAbsent(word, k -> new AtomicInteger( initialValue: 0)).incrementAndGet();
                }
            }
        }
    });
    futures.add(f);
}
```

Here we use ConcurrentHashMap combined with AtomicInteger to safely handle concurrent updates without needing locks.

Once all batches are submitted, we wait for each task to finish using their Future. Finally, the pool is shut down and we print the top 10 most frequent words.

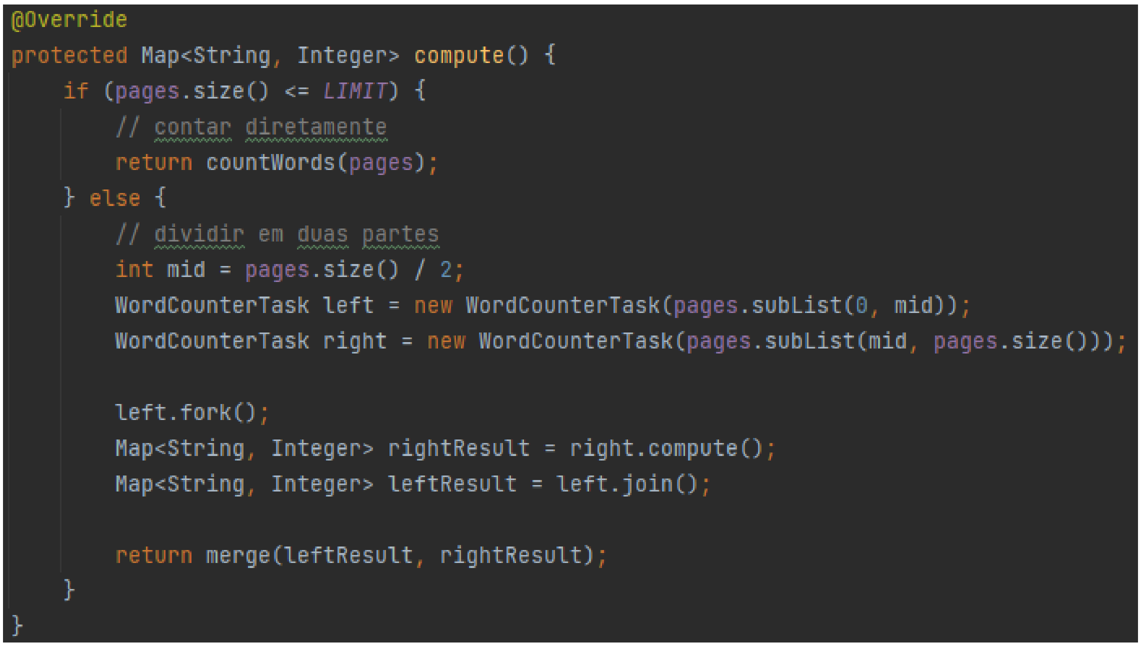
5. Fork/Join Framework Solution

In this version we tried a different parallel strategy by using Java's Fork/Join framework which is especially good for problems that can be broken into smaller parts.

The idea is to recursively split the workload in our case, a list of Wikipedia pages into smaller and smaller chunks. Each chunk is processed in parallel and the results are combined at the end.

The main goal here was simple. If the list of pages is small enough, we process it directly. If not, we split it in half and process each half in parallel then merge the results.

That logic is in the WordCounterTask, which extends RecursiveTask. This class is designed exactly for this kind of "divide and conquer" problem. The logic can be seen here:



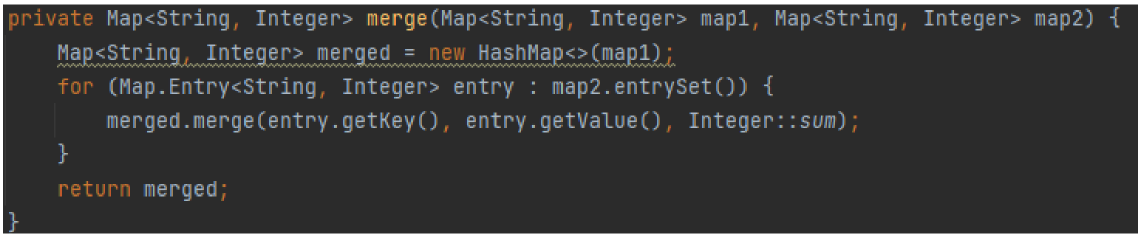
```java
@Override
protected Map<String, Integer> compute() {
    if (pages.size() <= LIMIT) {
        // contar diretamente
        return countWords(pages);
    } else {
        // dividir em duas partes
        int mid = pages.size() / 2;
        WordCounterTask left = new WordCounterTask(pages.subList(0, mid));
        WordCounterTask right = new WordCounterTask(pages.subList(mid, pages.size()));

        left.fork();
        Map<String, Integer> rightResult = right.compute();
        Map<String, Integer> leftResult = left.join();

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

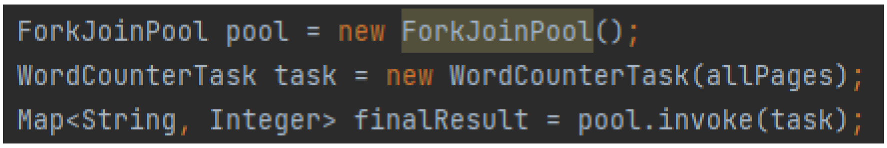
But when the list is bigger then we fork the left side (will run in parallel) and immediately compute the right side. Then we just wait for the left side to finish (with the join) and merge the results of the right and left side.

This merge is done with the merge function:



```java
private Map<String, Integer> merge(Map<String, Integer> map1, Map<String, Integer> map2) {
    Map<String, Integer> merged = new HashMap<>(map1);
    for (Map.Entry<String, Integer> entry : map2.entrySet()) {
        merged.merge(entry.getKey(), entry.getValue(), Integer::sum);
    }
    return merged;
}
```

In the main, we first collect all the pages and then start the process with a ForkJoinPool:



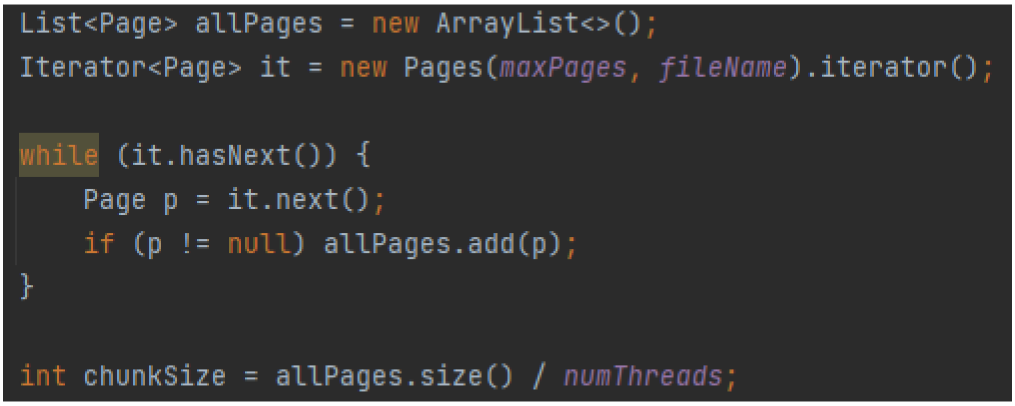
```
ForkJoinPool pool = new ForkJoinPool();
WordCounterTask task = new WordCounterTask(allPages);
Map<String, Integer> finalResult = pool.invoke(task);
```

6. CompletableFuture-Based Solution

For this version, we explored a more modern and high-level way to write asynchronous code using Java's CompletableFuture. Instead of managing threads directly or using recursive tasks, we focused on defining what should happen and let the framework handle when and how it runs.

The main idea was to treat each batch of pages as a task, submit them asynchronously and then combine all the results once they're done. this approach gave us cleaner code and let us take advantage of parallelism without worrying about thread creation or manual joins.

We started by reading all the pages into memory and splitting them into equal-sized chunks one for each thread:

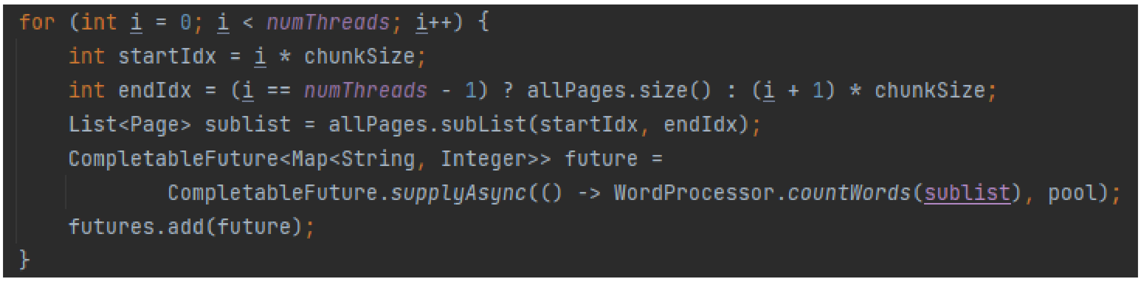


```
List<Page> allPages = new ArrayList<>();
Iterator<Page> it = new Pages(maxPages, fileName).iterator();

while (it.hasNext()) {
    Page p = it.next();
    if (p != null) allPages.add(p);
}

int chunkSize = allPages.size() / numThreads;
```

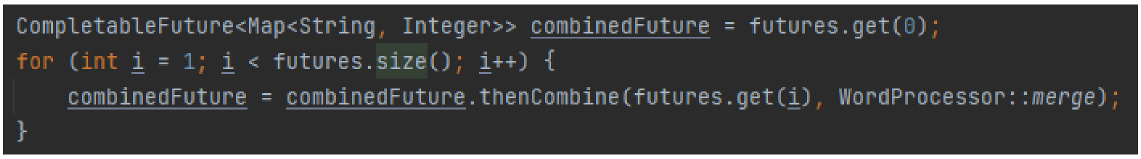
Then, for each chunk, we created an asynchronous task using:



```
for (int i = 0; i < numThreads; i++) {
    int startIdx = i * chunkSize;
    int endIdx = (i == numThreads - 1) ? allPages.size() : (i + 1) * chunkSize;
    List<Page> sublist = allPages.subList(startIdx, endIdx);
    CompletableFuture<Map<String, Integer>> future =
            CompletableFuture.supplyAsync(() -> WordProcessor.countWords(sublist), pool);
    futures.add(future);
}
```

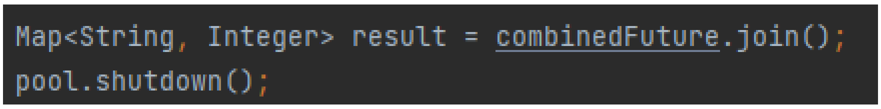
Each one runs in parallel and counts the words in its chunk using the WordProcessor.countWords method. That method is the same as in previous versions. it loops through each page, extracts words, filters them and updates a local map.

Finally we need to merge all the partial results. Instead of waiting manually for each task, we chained them using thenCombine:



```java
CompletableFuture<Map<String, Integer>> combinedFuture = futures.get(0);
for (int i = 1; i < futures.size(); i++) {
    combinedFuture = combinedFuture.thenCombine(futures.get(i), WordProcessor::merge);
}
```

This creates a kind of pipeline each future waits for the one before it and combines its result using the merge function. That way, at the end, we have one big map with the final counts.



```java
Map<String, Integer> result = combinedFuture.join();
pool.shutdown();
```

This blocks until all the tasks are done and gives us the full result. We shut down the pool and print the top words like in the other versions.

Garbage Collector Tuning

While optimizing for parallelism was the main goal of this project, we also explored how Java's Garbage Collector (GC) affects performance, especially under heavy memory load. Processing large XML dumps and storing thousands of pages and word counts in memory puts real pressure on the JVM's memory management so GC was really helpfull.

We tested different garbage collectors by launching the program with various JVM flags. We used two different configurations:

7. G1GC (Garbage-First GC): Enabled with -XX:+UseG1GC. This is the default collector in most recent Java versions. It's designed for predictable pause times and a balanced trade-off between throughput and latency;
8. Parallel GC: Enabled with -XX:+UseParallelGC. This collector focuses on maximizing throughput by using multiple threads for stop-the-world garbage collection. It's simple and efficient in scenarios with short-lived objects.

In addition, we configured the heap size in all tests using: -Xms2g -Xmx4g. This gave the JVM 4GB of memory max and 2 GB min to work with, which helped ensure consistency across test runs and avoided frequent resizing of the heap.

The detailed effects of each GC on performance (execution time, memory pressure and GC overhead) are discussed later in the Performance Analysis section.

Below are examples of the commands we used to run the different configurations:

Compile specific solution and common folder

```
javac -d out src/common/*.java src/sequential/*.java
```

Run with default GC

```
java -Xms2g -Xmx4g -cp out sequential.WordCount > logs/sequential/machine1/default-output.log
```

Run with G1GC and logging

```
java -Xms2g -Xmx4g -XX:+UseG1GC -Xlog:gc*:file=logs/sequential/machine1/gc-g1.log -cp out sequential.WordCount
```

Run with Parallel GC and logging

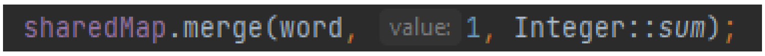
```
java -Xms2g -Xmx4g -XX:+UseParallelGC -Xlog:gc*:file=logs/sequential/machine1/gc-parallel.log -cp out sequential.WordCount
```

Concurrency and Synchronization

One of the biggest challenges we faced throughout the project wasn't just making things run in parallel but it was making sure they ran safely.

When we started using multiple threads, we had to think carefully about shared data. especially the structure that holds the word counts. If two threads tried to update the same word at the same time, we could easily end up with inconsistent or wrong results.

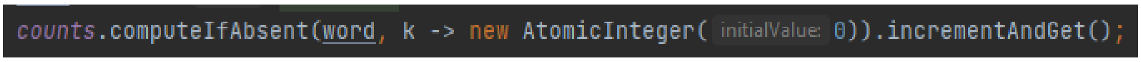
In the version with **manual threads**, we tackled this by using a ConcurrentHashMap. This gave us thread-safe access without needing to manually lock anything. To update the map, we used:



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

This method safely handles concurrent updates by combining the values atomically if the key already exists. Without it, we would've needed locks or risked inconsistent data.

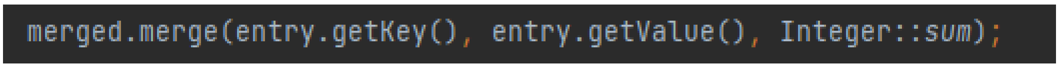
With **thread pools**, the situation was similar, but we added an extra layer of safety by combining ConcurrentHashMap with AtomicInteger. This allowed us to increment values directly, without replacing the whole entry:



```
counts.computeIfAbsent(word, k -> new AtomicInteger( initialValue: 0)).incrementAndGet();
```

This usage gave us great performance and avoided the need for any manual synchronization.

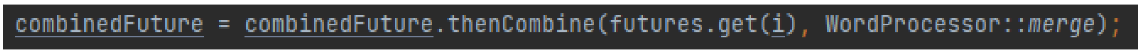
Fork/Join was a bit of a relief: each task worked with its own isolated data, using a normal HashMap. Since there was no shared access while counting, there were no concurrency issues inside the tasks. We only combined results at the end using a simple merge function:



```
merged.merge(entry.getKey(), entry.getValue(), Integer::sum);
```

This design avoided shared state entirely, which made things simpler and safer.

In the C**ompletableFuture** solution, we followed a similar idea to Fork/Join. each async task processed its own chunk of pages and built its own local map. Then, when all tasks completed, we combined their results using:



```
combinedFuture = combinedFuture.thenCombine(futures.get(i), WordProcessor::merge);
```

Again, by keeping the counting local and only synchronizing at the end, we avoided most of the usual concurrency problems.

In summary, we used different strategies depending on the model:

- Thread-safe collections (ConcurrentHashMap, AtomicInteger) when threads shared data;
- Isolation of state in Fork/Join and CompletableFuture to avoid the need for synchronization entirely.

This demonstrated us that writing concurrent code isn't just about using threads. it's about knowing when and how data is accessed and choosing the right tools to keep it safe.

Performance Analysis

To understand how each implementation performed under different conditions, we tested all solutions across three different machines, each belonging to a member of our group. This included variations in operating systems, hardware and available CPU cores (for example, one of the machines was a MacBook).

In addition to comparing the different concurrency models, we also measured how different Garbage Collector configurations impacted performance. Our goal was not only to identify the fastest solution, but also to understand how execution time and resource usage changed depending on the environment and GC strategy.

All runs were executed using the same dataset with consistent parameters across machines to keep the comparisons fair.

Machine 1 (1190782) – Legion i7-9750H 2.60GHz, 16,0 GB RAM and 768Gb SSD

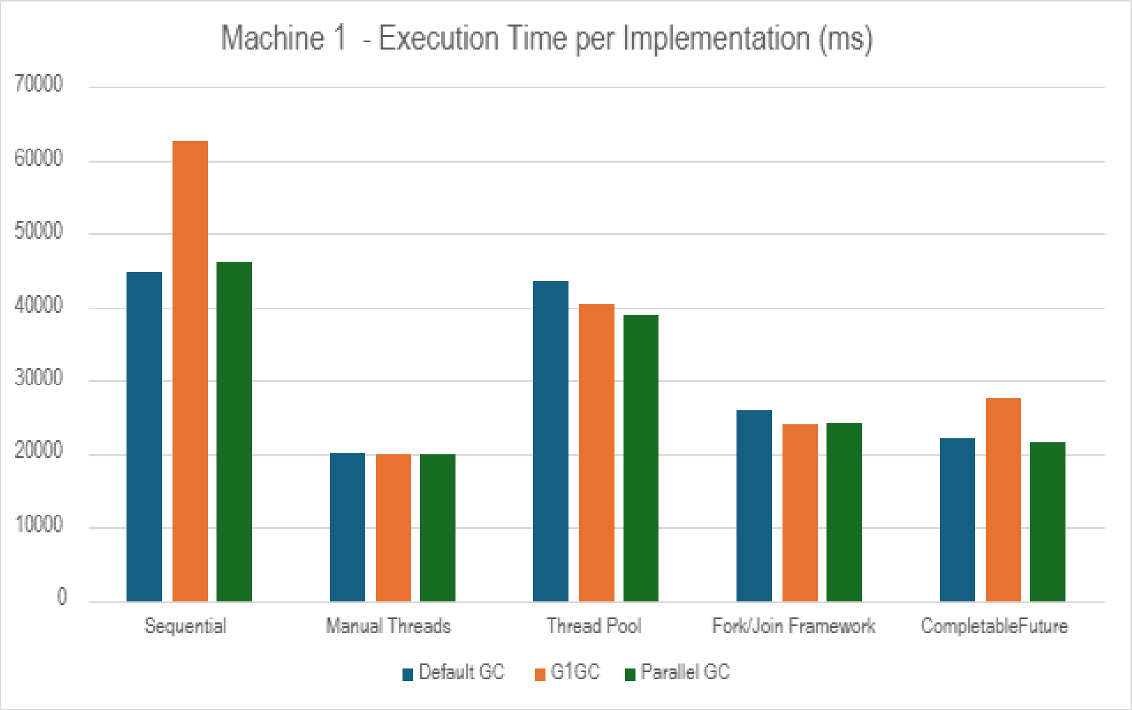
Machine 2 (1230164) – Custom Build AMD Ryzen 7 7800 x3d, 32,0 GB RAM and 1Tb SSD

Machine 3 (1201477) – MacBook Pro 13 2020 i51038NG7 2.30GHz, 8.0 GB Ram and 512 Gb

Scalability Considerations

Although our testing was against a fixed dataset, it is useful to look at how each would scale as the workload increased. The sequential version would perform bad with large inputs or low-CPU machines as it gains nothing from multicore processors. The parallel versions and Thread Pool and Fork/Join in particular are expected to scale much better, making better use of available cores and distributing the workload more. This would make a real difference in the real world with larger data or on a multi-processor machines and increase the performance difference even more from the sequential version.

Machine 1



Looking at the execution times on this machine, we can clearly see how both the choice of implementation and Garbage Collector (GC) had a big impact on performance:

- **Sequential** had the worst performance overall, especially with G1GC where it reached 62,645 ms, much higher than with the default GC (44,688 ms) or Parallel GC (46,218 ms). This shows how GC overhead can affect even a simple single thread program;
- **Manual Threads** performed consistently well across all GC settings, with times around 20,000 ms. This version gained a lot from parallelism without too much GC pressure, probably because each thread handled its own batch independently. Although we can see that the sequential is higher than any of the others GC's;
- **Thread Pool** showed good performance too, especially with Parallel GC (39,086 ms). The default was the worst, as expected and then the G1GC showed nice improvements being the Parralel the best one;
- **Fork/Join** run with solid and stable results. It handled the work efficiently and both G1GC and Parallel GC helped keep times low (around 24,000–24,500 ms);
- **CompletableFuture** had slightly more variation. With Parallel GC it performed very well (21,635 ms), but G1GC caused a significant slowdown

(27,728 ms). This may be due to more frequent object creation and less predictable memory behavior.

The garbage collection logs gave us helpful info into how memory was managed behind the scenes for each solution:

- For the **Sequential implementation**, G1GC triggered 116 GC events with a total pause time of 1.5 seconds, while Parallel GC had slightly less events (102) and less total pause time (0.93 seconds). this confirms that G1GC added more overhead explaining its longer execution time;
- With **Manual Threads**, both collectors behaved similarly, but Parallel GC ended up with a slightly higher total pause time (1.29s) and pause time per event. Even though, the total runtime was still fast;
- **Thread Pool** had really different number of GC events: G1GC triggered a massive 804 events and Parallel GC even more (1,150). the pause time was much lower, especially with Parallel GC (only 1.26 ms per event). This shows that while more frequent, the pauses were very short and didn't hurt performance so much;
- In the **Fork/Join solution**, G1GC caused longer pauses (18 ms), while Parallel GC had more fewer events and with only 7.6 ms per pause;
- **CompletableFuture** also showed many GC events (especially with G1GC), but overall the total pause time was lower than before (about 1.3s). Still, the impact of G1GC was visible in the higher execution time compared to Parallel GC.

When we compare the **Default GC** across implementations we see a big difference between the Sequential approach and the rest. While Sequential took 45 seconds, all the parallel versions completed in half that time or less. Manual Threads and CompletableFuture performed especially well, showing that adding concurrency already brings huge performance gains.

When we compare G1GC and Parallel GC it's easy to see that Parallel GC was generally the better option. It gave the best results or stayed really close to the best in almost every implementation. G1GC only showed better results in the Manual Threads version and even there the difference wasn't significant. In most of the other cases especially Sequential and CompletableFuture G1GC just added more pauses and slowed things down. it's a more complex collector, but in our tests it didn't really help and sometimes made things worse. Parallel GC was simpler, more stable was the most reliable choice overall.

The following table summarizes the number of Garbage Collection (GC) events and the total accumulated pause time (in milliseconds) for each implementation using G1GC and ParallelGC collectors. These metrics were extracted from the execution logs on Machine 1 and shows us the impact of memory management on performance.

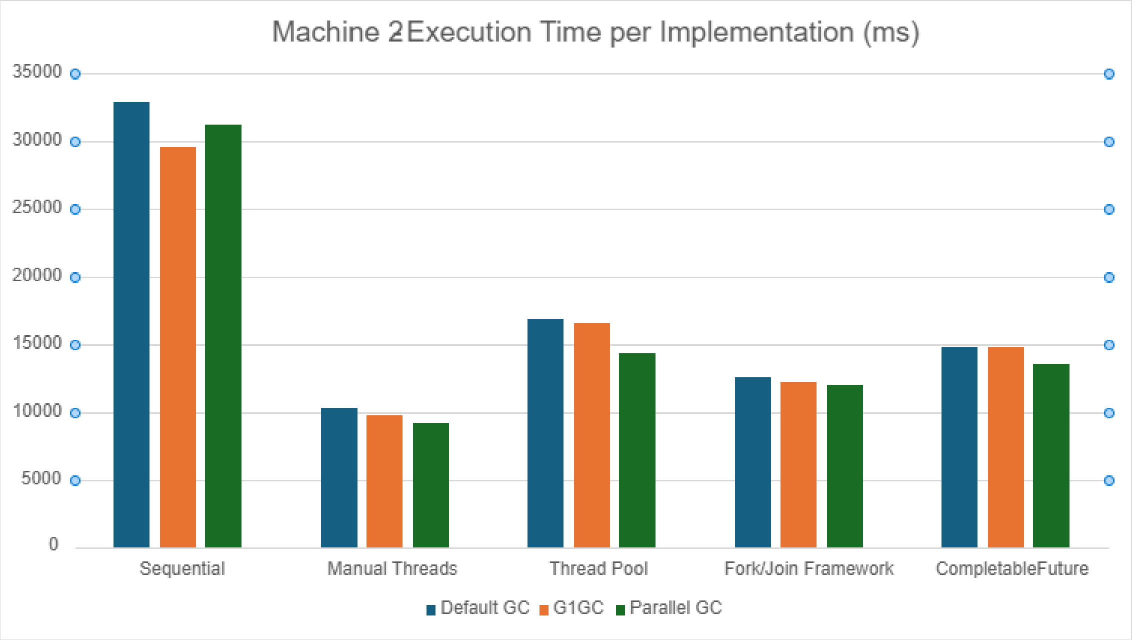
Implementation	GC Type	GC Events	Total Pause Time (ms)
completablefuture	G1GC	136	5364.04
	ParallelGC	86	496.98
forkjoin	G1GC	188	20830.03
	ParallelGC	126	728.84
sequential	G1GC	118	2710.83
	ParallelGC	102	0
thread_manual	G1GC	134	4804.26
	ParallelGC	92	1089.42
thread_pool	G1GC	976	30802.03
	ParallelGC	1158	534.26

The GC analysis shows clear differences depending on the gc and the level of parallelism. As expected, G1GC caused more frequent and longer pauses, especially in thread_pool, where it reached over 30 seconds of total pause time. Even though it's designed to reduce pause times, the overall impact can still be high with heavy workloads.

On the other hand, ParallelGC had much shorter total pauses, even in parallel versions. For example, thread_pool had more than 1,100 events but only 500 ms of pause time, making it more suitable for performance focused on applications. In the sequential version with ParallelGC no GC pause time was recorded, which shows lower memory pressure.

This confirms that the choice of GC has a strong impact on performance and in our case ParallelGC was more efficient for parallel implementations.

Machine 2



On this machine, we saw once again how both the implementation and the GC configuration play an important role in performance.

- **Sequential** was clearly the slowest, taking over 30,000 ms in all configurations. Even though G1GC improved performance compared to the default, the difference was minimal and Parallel GC gave the same results.
- **Manual Threads** performed really well, with the best time of 9,300 ms using Parallel GC. the G1GC and default GC were slower, but still below the 11,000 ms.
- **Thread Pool** showed clear improvements when switching GCs. While the default took 17,000 ms, G1GC reduced it a bit and Parallel GC was 14,000 ms.
- **Fork/Join** had a consistent performance, finishing around 12,000 and 13,000 ms for all GC setups. It seems this implementation balanced the work efficiently enough that GC choice didn't change that much.
- **CompletableFuture** in Parallel GC was better than G1GC by about 2,000 ms. Sequential was the worst as expected.

The garbage collection logs gave us helpful info into how memory was managed behind the scenes for each solution:

- For **Sequential**, both GCs behaved pretty similarly. G1GC ran 70 events and Parallel GC ran 56, with total pause times being around 800 ms for both. The average pause was a bit longer with Parallel GC, but that didn't make difference.
- In **Manual Threads**, the difference was much more noticeable. G1GC ran 320 GC events, while Parallel GC just 82. Both finished quickly overall.
- **Thread Pool** was different. G1GC ran 432 collections and Parallel GC 606 but the important to notice was that Parallel GC had just over 1 millisecond.
- **In Fork/Join** G1GC caused longer total pause time (1.7 seconds) than Parallel GC (1.3 seconds), even though Parallel GC had less events. Both performed well, but Parallel GC was better.
- **CompletableFuture** had in G1GC ran 342 events but the pauses were short around 3 ms each. Parallel GC had less events (82), but with longer pauses.

Looking just at the **Default GC**, we noticed that performance stayed acceptable for the parallel implementations, but it was clearly not optimized especially in Thread Pool and CompletableFuture. It worked best in Manual Threads which needed the least from the GC.

Comparing **G1GC and Parallel GC**, Parallel GC showed that was the most consistent. It gave the best result in almost every implementation. G1GC did well in some cases, like Manual Threads, but in others (Fork/Join and CompletableFuture) it caused longer GC times and didn't have faster execution. Parallel GC kept pause times low which helped all parallel models finish sooner.

The following table lists the number of Garbage Collection (GC) events as well as the total pause time for each G1GC and ParallelGC collector. These figures were derived from execution logs on Machine 2 and indicate how the management of memory affects performance.

Implementation	GC Type	GC Events	Total Pause Time (ms)
completablefuture	G1GC	930	11916.84
	ParallelGC	86	284.75
forkjoin	G1GC	900	19296.15
	ParallelGC	100	2886.26
sequential	G1GC	70	777.7
	ParallelGC	56	0
thread_manual	G1GC	848	8360.86
	ParallelGC	86	260.25
thread_pool	G1GC	432	1879.3
	ParallelGC	610	172.72

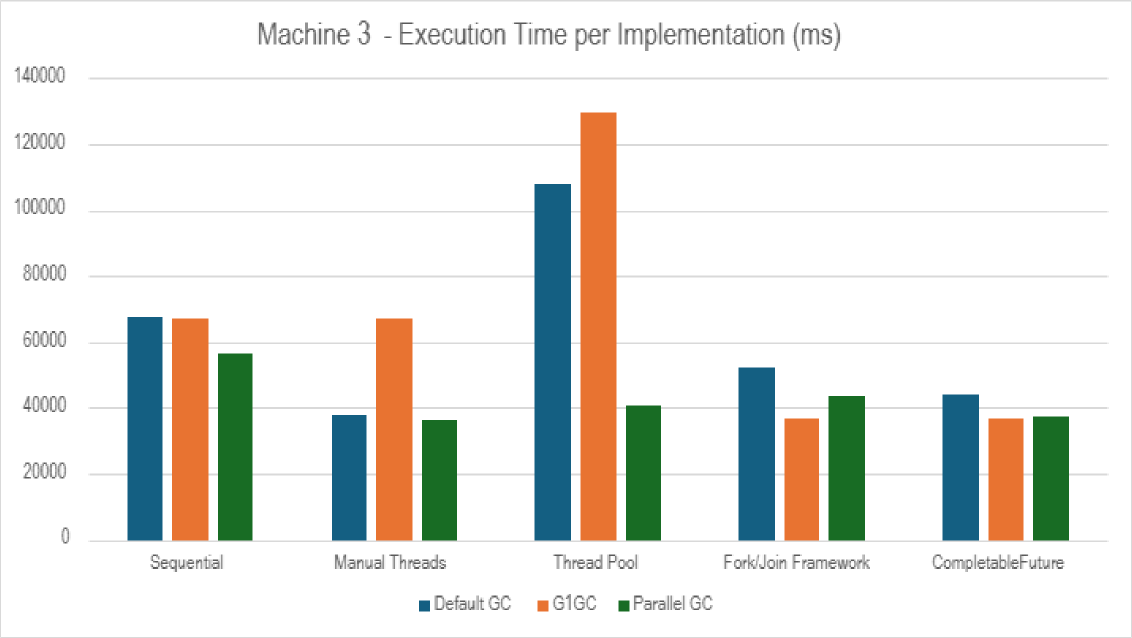
GC statistics on Machine 2 also show G1GC consistently generated more GC events and larger pause times on all deployments. An example is completablefuture with 930 events and close to 12 seconds of pause time compared to more than 19 seconds by forkjoin, showing the effect of excessive parallelism under memory pressure.

ParallelGC also recorded shorter pauses. Relatively high number of events thread_pool with 610 events the aggregate pause time was still low, supporting its applicability to throughput workloads.

The sequential build with ParallelGC had zero pause time, indicating that it manages light usage of memory well.

The results validate the inference that ParallelGC is more efficient on parallel workloads, delivering improved performance by lessening the total time taken by the garbage collection.

Machine 3



Machine 3 showed a very different behavior compared to the others probably due to having fewer CPU cores and less RAM.

- **Sequential** performed consistently across the board taking around 60,000 to 70,000 ms depending on the GC used. Parallel GC had the best result here (just above 50,000 ms) and G1GC offered no improvement.
- **Manual Threads** worked good with Parallel GC (around 36,000 ms), but G1GC caused a decrease (over 65,000 ms). The default GC was in between.
- **Thread Pool** was the most inconsistent. The default GC took around 110,000 ms and G1GC went over 130,000 ms. In contrast, Parallel GC performed much better finishing in about 40,000 ms.
- **Fork/Join** and **CompletableFuture** were the most stable. Both stayed under 60,000 ms for all GC, with Parallel GC again haviing having the best time (just above 40,000 ms in each).

The garbage collection logs gave us helpful info into how memory was managed behind the scenes for each solution:

- **Sequential** had few GC events 33 with G1GC and 27 with Parallel GC. But the pause time with Parallel GC was significantly lower (37 ms and 72 ms), which helped lower total runtime.
- **Manual Threads** showed higher GC overhead with G1GC (67 events, 125 ms avg. pause), while Parallel GC was a bit less (49 events, 135 ms avg.), but still resulted in faster execution.

- **Thread Pool** was different. G1GC ran 200 GC events with a pause time of 64 ms, setting to over 12 seconds in GC pause time. Parallel GC ran more events (289), but with very fast pauses (9 ms), only 2.5 seconds.
- In **Fork/Join**, G1GC had less events but longer pauses, while Parallel GC ran more events but shorter.
- **CompletableFuture** Parallel GC had less total pauses in time, even with higher time per pause.

Looking only at the Default GC, we saw high execution times in every implementation except Fork/Join and CompletableFuture. On this machine, the default behavior didn't help much when the data became more intense.

Comparing G1GC and Parallel GC, the bette: Parallel GC outperformed G1GC in all implementationss. G1GC had longer pauses, more total GC time and made performance worse than the default. Parallel GC handled memory faster and more efficiently, making it the most suitable option for a lower-power machine like this MacBook.

The following table summarizes the number of Garbage Collection (GC) events and the total accumulated pause time (in milliseconds) for each implementation using G1GC and ParallelGC collectors. These metrics were extracted from the execution logs on Machine 3 and show how memory management influenced performance on a lower-resource system.

Implementation	GC Type	GC Events	Total Pause Time (ms)
completablefuture	G1GC	71	3852,3
	ParallelGC	48	0
forkjoin	G1GC	87	11168,99
	ParallelGC	53	0
sequential	G1GC	33	67049
	ParallelGC	27	56561
thread_manual	G1GC	81	79212,07
	ParallelGC	52	36578
thread_pool	G1GC	202	133329,84
	ParallelGC	291	40538

The GC performance on Machine 3 shows collector discrepancy in performance under constrained hardware. Long pause times were generated by G1GC on all implementations, most significantly by thread_pool with more than 133 seconds of pause time and thread_manual with close to 80 seconds. These are indicative of the additional load on the memory and GC on low-end systems.

In contrast, ParallelGC had much better performance even as it generated appreciable pause times in certain scenarios. Times were 40 seconds in thread_pool and 36 seconds in thread_manual, even when using sequential with ParallelGC, over 56 seconds of pause was generated, suggesting that low RAM and CPU cores can exacerbate GC impact even on non-parallel workloads.

In brief, Machine 3 was even more affected by GC overhead, particularly G1GC. ParallelGC was still more efficient overall, but the data illustrates that hardware constraints make GC tuning even more influential even in less complicated implementations.

Conclusions

This project was a great opportunity to dive into the world of parallel programming in Java. Starting from a basic sequential solution and gradually introducing different levels of concurrency helped us understand exactly what changes when we move from one thread to many.

The Manual Threads implementation showed the complexity of managing concurrency manually. In contrast, Thread Pools simplified this process while introducing new considerations regarding task scheduling and load balancing. The Fork/Join framework introduced a recursive divide-and-conquer approach offering efficient parallelism with minimal coordination. Finally, the CompletableFuture-based solution demonstrated the potential of asynchronous programming and composition of concurrent tasks. In this project, it was also possible to see the significant influence of different Garbage Collector (GC) strategies on the solutions performance.

An analysis across different machines confirmed that hardware characteristics significantly influence results. Machine 2, with the highest specifications, achieved better execution times and handled both GC configurations efficiently. Machine 1 shoed more variability, particularly with G1GC, while Machine 3 with more lower resources showed that the choice of both implementation and GC strategy can lead to substantial performance differences.

Apart from technical implementation, this project helped us to understand how concurrency works in more real scneario practice, how memory is managed by the JVM and how GC usage can affect application behaviour. Some of the topics not so spoken, for example GC pause times or the overhead of certain thread management techniques, showed to be essential when evaluating real performance.

In conclusion, this project was challenging and very needed. It provided hands-on experience with key concepts such as thread coordination, task parallelism, asynchronous execution and memory better usage that are essential for developing efficient and scalable applications in modern multicore environments.

References

1. SISMD Moodle – *Lecture slides*. Instituto Superior de Engenharia do Porto (ISEP), 2024/2025.
   Available at: Disciplina: DEI - Sistemas Multinúcleo e Distribuídos - 2º Semestre 2024/2025 SISMD Moodle – *PLs*. Instituto Superior de Engenharia do Porto (ISEP), 2024/2025.
   Available at: Disciplina: DEI - Sistemas Multinúcleo e Distribuídos - 2º Semestre 2024/2025
2. SISMD Moodle – *TP Enunciados*. Instituto Superior de Engenharia do Porto (ISEP), 2024/2025.
   Available at: Disciplina: DEI - Sistemas Multinúcleo e Distribuídos - 2º Semestre 2024/2025
3. Oracle Docs – *Java Garbage Collection Tuning Guide*. Oracle Corporation.
   Available at:
   https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/
4. Baeldung – *A Guide to Java Garbage Collection*.
   Available at: https://www.baeldung.com/java-garbage-collection
5. Baeldung – *Introduction to Fork/Join Framework in Java*.
   Available at: https://www.baeldung.com/java-fork-join
6. GeeksforGeeks – *Java Thread Pool*.
   Available at: https://www.geeksforgeeks.org/thread-pools-java/
7. Stack Overflow – *When to choose SerialGC/ParallelGC over CMS/G1 in Java*.
   Available at: https://stackoverflow.com/questions/54615916/when-to-choose-serialgc-parallelgc-over-cms-g1-in-java
8. Medium – *Java 21: ZGC vs G1GC vs Parallel GC – Which One Is Best for High Throughput Apps?* by Shant Khayalian.
   Available at: https://medium.com/@ShantKhayalian/java-21-zgc-vs-g1gc-vs-parallel-gc-which-one-is-best-for-high-throughput-apps-6ac4f4d589d3
9. Medium – *Java Garbage Collectors Explained* by Anmol Sehgal.
   Available at: https://anmolsehgal.medium.com/java-garbage-collectors-610689a5b125
10. Stack Overflow – *What overhead does garbage collection logging add to the JVM?*.
    Available at: https://stackoverflow.com/questions/18359364/what-overhead-does-garbage-collection-logging-add-to-the-jvm
11. Medium – *Overhead Added by Garbage Collection Logging*.
    Available at: https://medium.com/@marketing_864/overhead-added-by-garbage-collection-logging-1456d599fbb9