# 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.

## 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.

## Sequential Solution

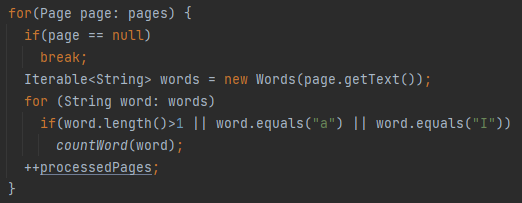
This version is our baseline. it does everything in one thread, from start to finish. The program reads each page from the XML dump, extracts the text, splits it into words and updates a frequency map.

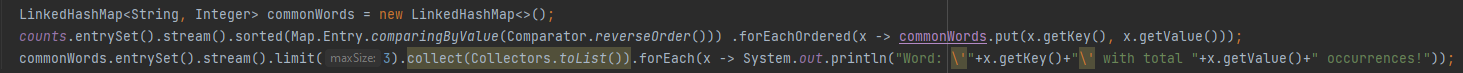
There’s no parallelism, no synchronization and no sharing of data between threads. That simplicity makes it easy to implement and debug, but it also means performance is limited, especially as the size of the dataset grows.

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

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.

A screen shot of a computer code

AI-generated content may be incorrect.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.

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

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.

## 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.

A screen shot of a computer code

AI-generated content may be incorrect.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:

A screen shot of a computer

AI-generated content may be incorrect.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:

A screen shot of a computer program

AI-generated content may be incorrect.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:

## 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:



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:

A computer screen shot of a black screen

AI-generated content may be incorrect.

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

A computer screen shot of a program code

AI-generated content may be incorrect.

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.

## 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:

A computer screen shot of a program code

AI-generated content may be incorrect.

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:

A computer screen with text

AI-generated content may be incorrect.

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

A black background with white text

AI-generated content may be incorrect.

## CompletableFuture-Based Solution

# Garbage Collector Tuning

# Concurrency and Synchronization

# Performance Analysis

# Conclusions

# References