Optimizing Large-Scale Data Processing on Multicore Systems

Inês Rodrigues Nepomuceno Alves
1242183
M1C

Sistemas Multinúcleo e Distribuídos
Mestrado em Engenharia Informática
Instituto Superior de Engenharia do Porto
2025

Table of Contents

Sistemas Multinúcleo e Distribuídos
Mestrado em Engenharia Informática
Instituto Superior de Engenharia do Porto
2025

Introduction

The goal of this project is to implement and analyze various approaches for efficiently processing large volumes of data on multicore systems. Specifically, the project aims to calculate the frequency of words in English texts by processing a large Wikipedia data dump. The performance of each implementation will be measured and analyzed to assess how different concurrency models influence execution time and resource utilization.

Objectives

- Implement multiple approaches to solve the problem of large-scale word frequency analysis;
- Identify the most efficient solution through performance comparison;
- Gain a deeper understanding of each concurrency model and explore techniques for tuning and optimization.

Implementation Approaches

The project explores five different implementation strategies for processing large-scale text data. One of these is a sequential approach, serving as a baseline for performance comparison. The remaining four leverage multicore processing to evaluate the benefits of parallel and concurrent execution.

All implementations were tested on two datasets: a smaller one with 27,368 pages and a larger one containing 89,618 pages. In each case, the focus was on identifying the top three most frequent words. Additionally, the criteria for determining valid words were modified from the original starter code, and this updated approach has been consistently applied across all implementations.

Below is an overview of the key aspects of each approach.

Sequential Solution

This approach is the original implementation provided by the professors. The key modification that sets this implementation apart from the one provided is the updated condition for determining whether a combination of characters qualifies as a valid word. Additionally, a few print statements were added primarily for human readability and benchmark execution. The logic for calculating the top N words was extracted into a method within a utility class (`Utils`), enabling its reuse across

other approaches. Another enhancement was defining the integer corresponding to the number of top words to analyze as a configurable variable.

Multithreaded Solution (Without Thread Pools)

This implementation adopts a multithreaded approach to process a large number of pages concurrently. It utilizes Java threads along with a `CountDownLatch` to coordinate the execution of multiple threads, eliminating the need for referencing threads explicitly.

The program counts the occurrences of words in the text of each page and stores the results in a thread-safe `ConcurrentHashMap`. The pages are processed in batches, with the number of batches determined by the available threads (`NUM_THREADS`), in this case, 12 threads. Each thread handles a subset of pages, counting word occurrences locally in a `HashMap`, and then merging the results into the global `ConcurrentHashMap`. To ensure proper synchronization, a `CountDownLatch` is used, which makes the program wait for all worker threads to complete before proceeding.

Please refer to Figure 1 and Figure 2 for further implementation details.

```
// Get List of pages and divide into batches according to the number of threads
List<Page>  pages = getListOfPages(pagesIterable);
int batchSize = (int) Math.ceil((double) pages.size() / NUM_THREADS);

// CountDownLatch to wait for all threads to finish without
// having to loop through the threads to join them
CountDownLatch latch = new CountDownLatch(NUM_THREADS);
List<Thread> threads = new ArrayList<>();
for (int i = 0; i < NUM_THREADS; i++) {

    // Get sublist of pages for each thread to process
    int startIdx = i * batchSize;
    int endIdx = Math.min(startIdx + batchSize, pages.size());
    List<Page> subList = pages.subList(startIdx, endIdx);

    Thread thread = new Thread(() -> {
            processPages(subList);
            latch.countDown();
        });
    thread.start();
}

// Wait for all threads to finish
latch.await();
```

Figure 1 - The logic for generating threads is based on dividing the work into batches, with the number of batches calculated according to the available threads.

```java
/*
 * Process a list of pages and count the occurrences of words in the text of each page.
 * The results are stored in a ConcurrentHashMap.
 */
private static void processPages(List<Page> pages) {  1 usage   Alvesl1-ppb
    for (Page page : pages) {
        if (page == null)
            break;
        Iterable<String> words = new Words(page.getText());
        Map<String, Integer> localCounts = new HashMap<>();
        for (String word : words){
            word = word.toLowerCase();
            if (isValidWord(word)){
                localCounts.merge(word,  value: 1, Integer::sum);
            }
        }

        // Merge local counts into global counts
        localCounts.forEach((word, count) ->
                counts.merge(word, count, Integer::sum)
        );

    }
}
```

Figure 2 - Wods are counted locally and them merged with into the global `ConcurrentHashMap`.

Multithreaded Solution (With Thread Pools)

This implementation adopts a multithreaded approach to process a large number of pages concurrently using a thread pool. It leverages Java's `ExecutorService` to manage threads efficiently, and a `ConcurrentHashMap` is used to store word counts in a thread-safe manner.

The program processes a list of pages by dividing them into batches of a fixed size, which was determined empirically and may not guarantee full correctness. A fixed-size thread pool (with 12 threads) is used to execute tasks concurrently, where each task handles the processing of a batch of pages. To ensure proper synchronization, a `CountDownLatch` is employed, ensuring that the main thread waits for all tasks to complete before proceeding. Each batch is processed locally in a `HashMap`, and the results are then merged into a global `ConcurrentHashMap` to ensure thread safety, similar to how it was done in the thread-based approach without the use of a thread pool (Figure 3).

Fork/Join Framework Solution

This implementation uses a parallelized approach to process a large number of pages by leveraging Java's `ForkJoinPool`. It counts word occurrences in the text of each page and stores the results in a thread-safe `ConcurrentHashMap`.

The list of pages is processed by dividing the workload into smaller tasks, based on a fixed threshold value, which was determined empirically. While this threshold provides reasonable performance, it may not represent the optimal value for all workloads. The `ForkJoinPool` framework manages the threads and recursively splits tasks into subtasks, enabling efficient parallel processing. Each task handles a subset of pages, counts word occurrences locally in a `HashMap`, and merges the results into a global `ConcurrentHashMap` to maintain thread safety, as previous approaches.

CompletableFuture-Based Solution

This implementation uses a parallelized approach to process a large number of pages by leveraging Java's `CompletableFuture`. It counts word occurrences in each page's text and stores the results in a thread-safe `ConcurrentHashMap`.

The list of pages is divided into batches of a fixed size, which was determined empirically and may not guarantee full correctness. Each batch is processed asynchronously using `CompletableFuture.runAsync`, allowing non-blocking parallel execution.

Each batch counts word occurrences locally in a `HashMap`, and the results are merged into a global `ConcurrentHashMap` to maintain thread safety, as previous approaches.

Garbage Collector Tuning

Due to time constraints, the garbage collector analysis was limited to testing a few collectors known to perform well with parallel workloads, without any fine-tuning of their parameters. Specifically, the evaluation included G1GC (the default), Z Garbage Collector (ZGC), Parallel GC, and Shenandoah GC.

Concurrency and Synchronization

In this project, and particularly across most parallel implementations, `ConcurrentHashMap` and atomic variables were used. This choice was made not only to ensure consistency in the results by avoiding race conditions, but also to simplify the design by eliminating the need to return and iterate over constructs like `Futures`.

The performance impact of using these structures was not explicitly benchmarked, for example, by comparing implementations with and without them in terms of CPU or memory usage. However, all

parallel approaches produced results consistent with the baseline outputs of the sequential implementation across both datasets, indicating correctness.

Performance Analysis

First, the results for each individual implementation will be presented, showcasing performance under different garbage collectors without any fine-tuning. This will be followed by a comparative analysis of all approaches using both datasets. Finally, a broader evaluation will be provided in the conclusions section, highlighting overall trends and insights across all implementations.

Sequential Solution

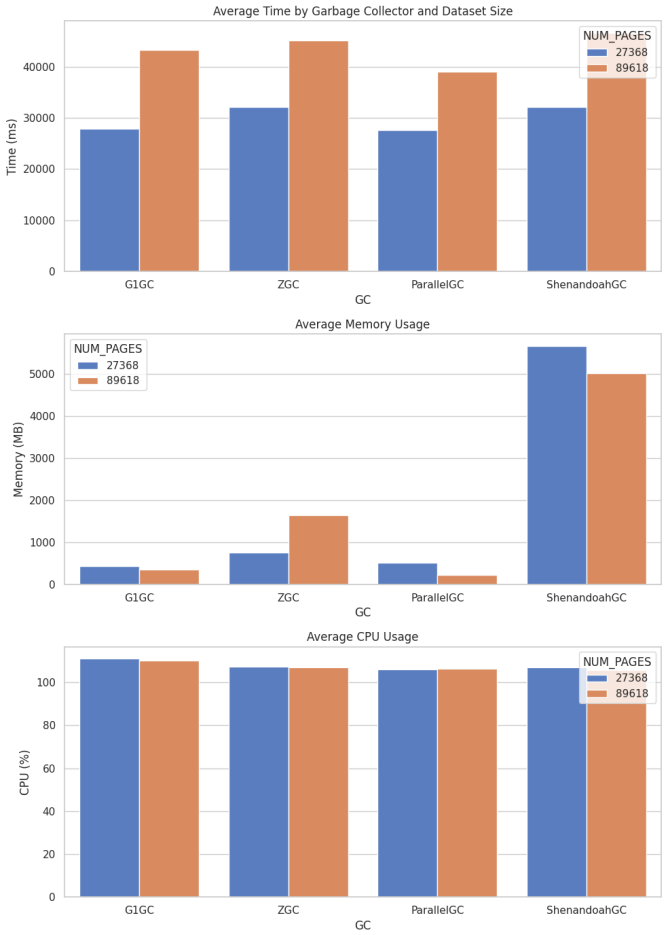

Figure 3 - Performance results for the sequential solution.

Multithreaded Solution (Without Thread Pools)

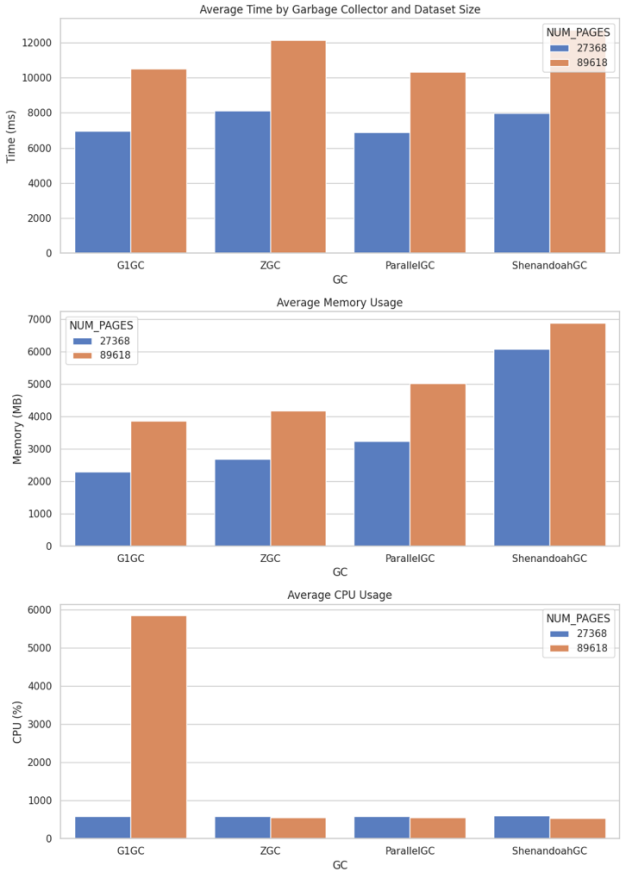

Figure 4- Performance results for the multithreaded solution (without thread pools).

Multithreaded Solution (With Thread Pools)

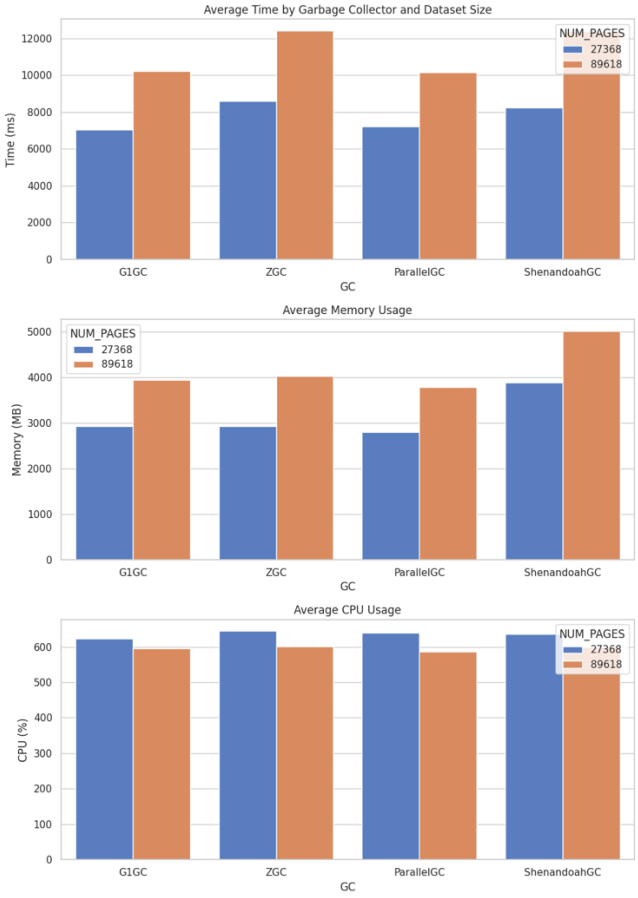

Figure 5 - Performance results for the multithreaded solution with thread pools.

Fork/Join Framework Solution

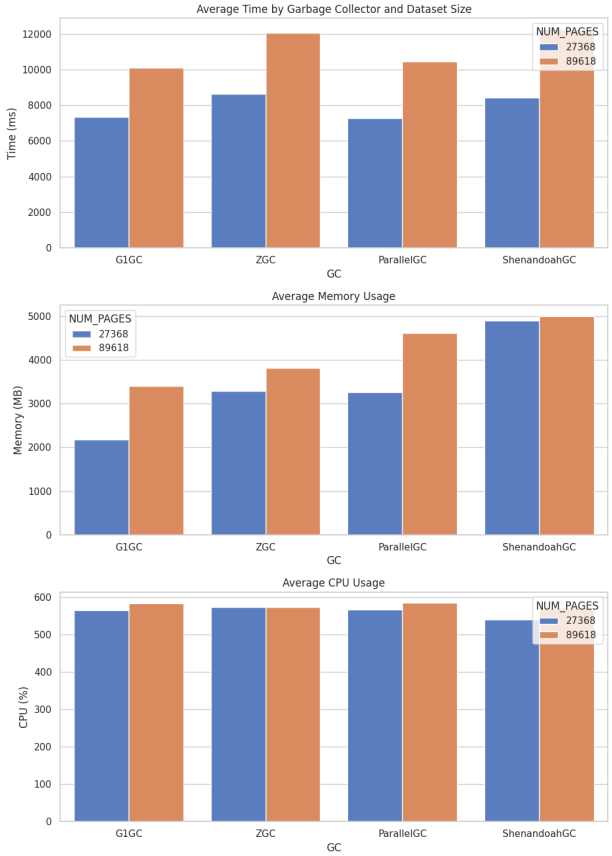

Figure 6 - Performance results for the Fork/Join Solution.

CompletableFuture-Based Solution

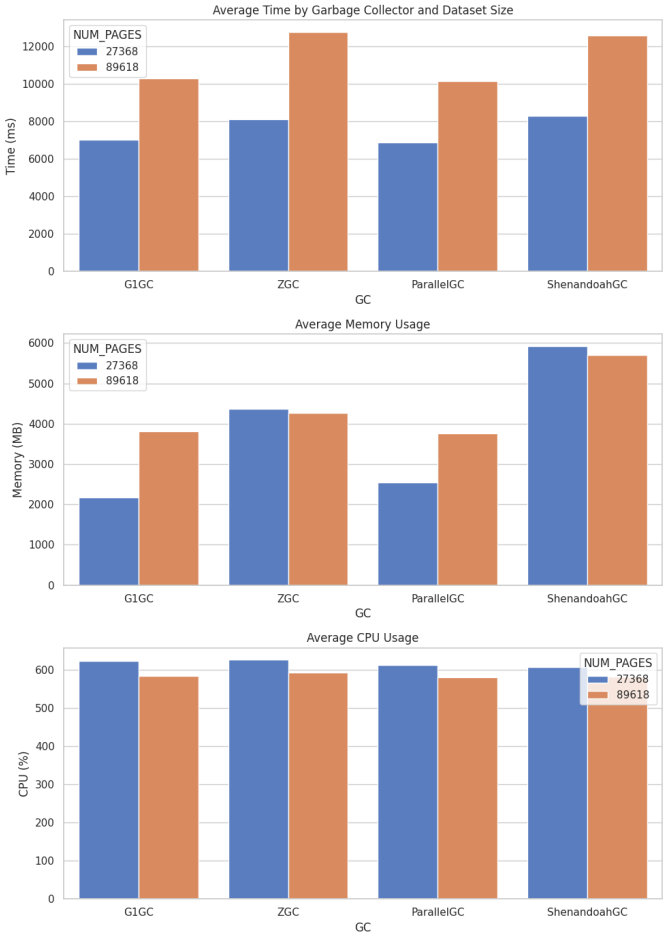

Figure 7 - Performance results for the Completable Future based solution.

General Analysis

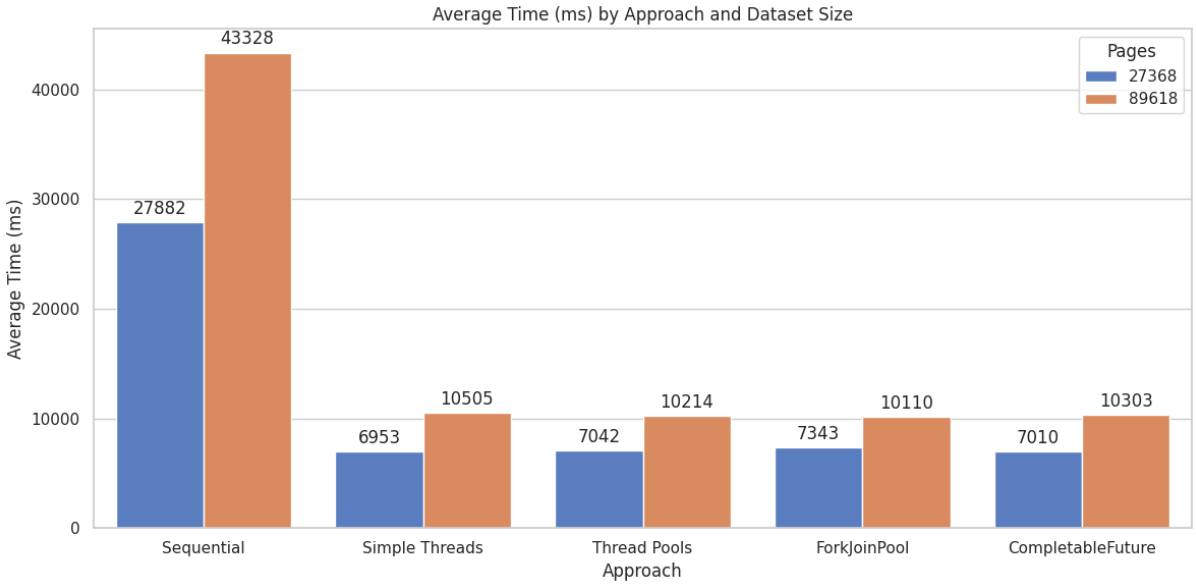

Figure 8 - Comparative time of execution for all approaches, using both datasets.

Conclusions

All multithreaded solutions show similar elapsed times, which suggests that either the implementations may have issues, or the problem is too simple to benefit from more complex algorithms. These more advanced algorithms likely introduce some overhead, which could explain why their execution times are comparable to simpler approaches. Another noteworthy observation is CPU usage, which is significantly higher across all multithreaded solutions, with the exception of the simple threads approach and the sequential implementation. Regarding memory usage, the sequential approach has the lowest memory consumption, except when using the Shenandoah garbage collector. Overall, all multithreaded solutions performed similarly for the given problem.

Future Improvements

One potential improvement is to better organize the code for increased readability and maintainability. By refactoring the implementation into smaller, modular components or classes, the project would become easier to manage as it grows in complexity. This approach would also simplify debugging and testing, particularly as more features or optimizations are added over time.

Another area for improvement is fine-tuning the garbage collectors used in the project.