

**Faculty of Engineering and Technology**

**Department of Electrical and Computer Engineering**

**ENCS3390-Operating System Concepts**

**Project 1 Report**

**Prepared by:** Tasneem Shelleh 1220439

**Instructor:** Dr. Mohammed Khalil

**Section:** 2

**Date:** 8/12/2024

# **Abstract**

This report explores how Java's multithreading and multiprocessing features can be used to find the top 10 most frequent words in a dataset. It explains the implementation process using the threading and multiprocessing libraries, examines how they perform under different conditions, and uses Amdahl’s law to analyze the results. The goal is to understand the potential speed improvements and how to make the best use of system resources.

**Table of Contents**

[**Abstract** II](#_Toc184585959)

[**List of Tables** III](#_Toc184585960)

[**List of figures** III](#_Toc184585961)

[**Introduction** 1](#_Toc184585962)

[**The environment of the work** 1](#_Toc184585963)

[**Implementation** 2](#_Toc184585964)

[**1.** **Naïve approach** 2](#_Toc184585965)

[**2.** **Multiprocessing approach** 2](#_Toc184585966)

[**3.** **Multithreading approach** 2](#_Toc184585967)

[**Analysis and discussion** 3](#_Toc184585968)

[ **Discussion:** 3](#_Toc184585969)

[ **The optimal number of child processes or threads** 4](#_Toc184585970)

[**The output of the code** 4](#_Toc184585971)

[**A table that compares the performance of the 3 approaches.** 4](#_Toc184585972)

[**The differences in performance of the 3 approaches** 5](#_Toc184585973)

[**Conclusion** 6](#_Toc184585974)

# **List of Tables**

[Table 1:comparing between the 3 approaches 4](#_Toc184585975)

# **List of figures**

[Figure 1: The top 10 most frequent words in the dataset 4](#_Toc184585976)

# **Introduction**

This project focuses on improving how we find the top 10 most frequent words in a dataset using Java's concurrency tools, like threading and multiprocessing. We compare these methods to a simple single-threaded approach to see which one performs better and delivers the most significant speedup.

# **The environment of the work**

This project was developed on a computer powered by an Intel Core i7 processor with 4 cores and a base speed of 2.30 GHz, supported by 8 GB of RAM to handle the workload smoothly. The operating system was Windows 10, providing a reliable and familiar development environment. Java was the chosen programming language because of its excellent support for concurrency and multithreading. The IDE tool that used is IntelliJ IDEA, which made coding, and debugging much easier. The code ran directly on the host machine without using a virtual machine, allowing us to fully utilize the hardware for efficient testing and execution.

# **Implementation**

## **Naïve approach**

The MultithreadingApproachThread class implements a word frequency counting solution using Java's Callable interface, designed for multithreaded processing. It processes an array of words by splitting it into smaller chunks splitWordsToChunks around 20 chunks, and calculating word frequencies for each chunk using a helper method getFrequencyList, that use two loops to calculate it. The results are combined and merged into a TreeMap<String, Integer>, which maintains sorted words and their counts. The words are then sorted by frequency, and the top 10 most frequent ones are displayed.

## **Multiprocessing approach**

The ProcessesApproach class reads a text file and calculates word frequencies using a certain number of parallel processes. It splits the content into words and uses the method ProcessBuilder to create new processes, then use the method ProcessManager to distribute the work on them. The method process.waitFor() will wait for the child process to finish its work. Then merge and store the results in a TreeMap. Afterward, it combines the word counts. The words are then sorted by frequency, and the top 10 most frequent ones are displayed.

## **Multithreading approach**

The MultithreadingApproach class calculates the top 10 most frequent words in a text file using multithreading, with each thread managed by the MultithreadingApproachThread class. It reads the file, splits its content into words, and divides the word array into chunks, assigning each chunk to one of 8 threads using an ExecutorService, which is newFixedThreadPool(numberOfThreads), after that by implementing executer.submit(new MultithreadingApproachThread(arrayParts[i])) a new thread will be created. The MultithreadingApproachThread class processes each chunk, computes word frequencies, and returns the results as a TreeMap. The method future.get() will wait for the new thread to finish its work.

The main program collects and merges these results, combining duplicate word counts into a unified list. The frequencies are then sorted in descending order, and the top 10 most frequent words are displayed.

# **Analysis and discussion**

Amdahl's Law helps predict the maximum possible speedup of a program when part of it is parallelized. The formula is: Speedup ,

where S is the serial part of the program, (P is the parallelizable portion), and N is the number of cores used.

After analyzing the performance, we found that about 86% of the code runs in parallel, with only 14% being serial. This shows that most of the processing is efficiently distributed across multiple threads or processes, making good use of parallelism. It highlights how well the code is optimized to handle tasks concurrently, resulting in faster and more efficient execution.

Then the maximum speedup, when the parallel portion is 86%, using 4 cores, is:

Speedup = 2.81

## **Discussion:**

When the maximum speedup of a code is 2.81, it means that parallelization has made the program about 2.81 times faster than when running on a single processor. While this is a good improvement, the speedup is still limited by the part of the code that can't be parallelized. The serial section of the program holds back the potential for even greater speedup. How much faster the program can run depends on how much of the code can be parallelized and how efficiently the parallel tasks are handled.

## **The optimal number of child processes or threads**

Since the computation isn't very complex and is mostly CPU-bound, using a number of threads that matches the number of available cores (like 4 cores) is usually the most efficient approach in Java. Adding more threads than there are cores likely won't improve performance due to the extra cost of context switching and potential issues like thread contention or synchronization. It's crucial in Java to match the number of threads to the available CPU resources to get the best performance without overloading the system.

# **The output of the code**

The top 10 most frequent words in the dataset:

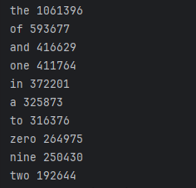


Figure : The top 10 most frequent words in the dataset

# **A table that compares the performance of the 3 approaches.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Approach | 1 threads/processes | 2 threads/processes | 4 threads/processes | 6 threads/processes | 8 threads/processes |
| Naïve approach | 540 sec = 9 min | - | - | - | - |
| Multiprocessing | - | 339sec = 5.62 min | 199 sec = 3.3 min | 162 sec = 2.7 min | 140sec = 2.3 min |
| Multithreading | - | 301sec = 5.01 min | 177sec = 2.95 min | 130sec = 2.16 min | 117sec = 1.95 min |

Table :comparing between the 3 approaches

# **The differences in performance of the 3 approaches**

**The Naïve approach**, which relies on a single thread or process, takes the longest at 540 seconds (9 minutes), underscoring the inefficiency of a single-threaded approach for tasks that can be parallelized.

**Multiprocessing** reduces execution time as more processes are added. With 2 processes, the time drops to 339 seconds (5.62 minutes), and with 4 processes, it decreases further to 199 seconds (3.3 minutes). By 8 processes, the time is 140 seconds (2.3 minutes), showing the benefits of distributing the workload across multiple CPU cores. This is especially effective for CPU-bound tasks, as each process runs on a separate core, improving efficiency.

**Multithreading** also reduces execution time, performing slightly better than **Multiprocessing**. At 2 threads, the time is 301 seconds (5.01 minutes), improving to 177 seconds (2.95 minutes) with 4 threads. With 6 threads, it drops to 130 seconds (2.16 minutes), and with 8 threads, it reaches 117 seconds (1.95 minutes), which it is the fastest of all configurations. Thread-level parallelism is more efficient than using multiple processes.

The main difference between **Multithreading** and **Multiprocessing** comes down to how they manage memory and handle context switching. **Multithreading** shares the same memory space across threads, making it more resource-efficient and reducing the overhead of context switching. In contrast, **Multiprocessing** assigns separate memory to each process, which helps avoid the issues caused by shared memory but increases overhead due to more frequent context switching and the need for communication between processes. So, both **Multiprocessing** and **Multithreading** outperform the **Naïve approach**, with **Multithreading** providing the best performance at higher thread counts. The results demonstrate the power of parallelization in improving efficiency.

# **Conclusion**

This project demonstrated how execution methods, such as naïve, multithreading and multiprocessing, can significantly speed up the process in identifying the top 10 most frequent words in a dataset. By comparing between these approaches, I saw clear performance improvements, with multithreading outperforming multiprocessing due to lower overhead and more efficient resource use. Amdahl’s law helped explain the limitations of speedup, emphasizing that the serial portion of the code ultimately limits the overall performance. In the end, both parallel approaches provided a substantial speed advantage over the naïve approach, with multithreading proving to be the most effective for this particular task.