Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS3390 Operating System Concepts

First Semester , 2024/2025

Project 1

The Name : Reem Salloum

ID Number : 1202160

Instructor : Mohammed Khalil

Section : 2

Date : 1/12/2024

Abstract

In this project, the goal was to analyze the frequency of words in a large dataset (text8.txt) and determine the top 10 most frequent words using three different approaches: naive (sequential processing), multiprocessing, and multithreading. Each approach was evaluated based on its execution time and efficiency on a multi-core machine.

The naive approach processed the dataset sequentially without any parallelism, whereas multiprocessing utilized multiple processes to divide the dataset into chunks and process them concurrently. Similarly, multithreading divided the dataset into chunks but used threads instead of processes. The performance of each approach was measured in terms of execution time, and the results were analyzed using Amdahl's Law to understand the speedup achieved and determine the optimal parallelization configuration.

This report provides a detailed comparison of these approaches, discusses their implementation, evaluates their performance, and concludes with recommendations for the most efficient approach based on the environment and dataset characteristics.

Table of contents

Table of figures

List of tables

1. Environment Description :

1) Computer Specification :

- Processor : 11th Gen Intel Core i7 (Cores : 4) .
- Speed : 2.80 GHz .
- Memory (RAM) : 8 GB .

2) Operating system (RAM) : Windows 10 , 64-bit

3) Programming language : Java , using OpenJDK 17

4) IDE tool : Eclipse IDE

5) Virtual machine : not used .

2) Implementation details :

2.1 Achievement of the multiprocessing and multithreading :

Naive Approach:

Description: This approach processed the entire dataset sequentially in a single thread.

Advantages: Simple implementation.

Disadvantages: Inefficient for large datasets as it does not utilize multi-core processors.

1. Multiprocessing

To implement multiprocessing, the following techniques and APIs were used:

- File Splitting:
    - The dataset was divided into manageable chunks using the splitFileIntoChunks method.
    - Each chunk represented a portion of the dataset to be processed independently.
- Process Management:

    - API Used: ExecutorService with a FixedThreadPool configuration.
    - Functionality:
        - Created a pool of processes equal to the specified number (numProcesses).
        - Submitted each chunk as a separate task to the pool using the submit method.
        - Each process counted word frequencies for its assigned chunk in parallel.
- Merging Results:
    - Data Structure Used: ConcurrentHashMap.
    - Functionality:
        - Each process produced a partial word frequency map.
        - These partial results were merged into a single map using the mergeFrequencies method.

2. Multithreading

The multithreading implementation used similar logic but leveraged threads instead of processes for parallel execution:

- Thread Management:
    - API Used: ExecutorService with FixedThreadPool.
    - Functionality:
        - Created a pool of threads instead of processes.
        - Each thread processed one chunk of the dataset.
- Shared Memory:
    - Threads shared memory space, making the ConcurrentHashMap ideal for storing and merging word frequency counts efficiently.
    - This reduced overhead compared to multiprocessing, which requires separate memory spaces.

1) File splitting : BufferedReader, splitFileIntoChunks , Efficiently read and split the dataset into chunks for parallel processing.

2) Task Submission : ExecutorService, submit , Distribute tasks (chunk processing) among processes or threads.

3) Synchronization : ConcurrentHashMap , Safely merge partial word frequency results from multiple threads/processes.

4) Word Counting : HashMap, countWords , Count word occurrences in each chunk.

5) Result Merging : mergeFrequencies , Combine results from all threads or processes into a single map.

6) Sorting : List.sort() , Identify the top 10 most frequent words.

SO :

1) File Splitting:

- Dataset is divided into numProcesses or numThreads chunks using the splitFileIntoChunks method.
- Each chunk is processed independently in parallel.

2) Parallel Processing:

- Multiprocessing: Tasks are distributed across separate processes.
- Multithreading: Tasks are distributed across threads in a shared memory environment.

3) Word Frequency Counting:

- Each thread or process counts the frequency of words in its assigned chunk using countWords.

4) Result Merging:

- Partial results are merged into a ConcurrentHashMap to ensure thread safety and correctness.

3) Amdahl's Law Analysis :

Amdahl's Law provides a theoretical upper bound on the speedup achievable by parallelizing a portion of a program. It is used to predict the maximum performance improvement that can be obtained by parallelizing a program, given the portion of the code that is serial (i.e., not parallelizable).

The formula for **Amdahl's Law** is:

Speedup = 1 / (S + (P/N))

Where:

- S is the **serial portion** of the code (the fraction that cannot be parallelized).
- P is the **parallelizable portion** of the code (the fraction that can be parallelized).
- N is the **number of processors or threads**.

Serial Portion of the Code

- The serial portion includes:
  - **File reading** and **splitting** (this occurs sequentially).
  - **Merging word frequencies** (this is not fully parallelizable as it requires combining partial results).
  - **Sorting the results** (this step cannot be fully parallelized as the final sorting is a global operation).

The **serial portion** of the code to be around **30%** of the total execution time, as these tasks cannot be effectively parallelized.

Parallel Portion of the Code

- The parallelizable portion includes:
  - **Counting word frequencies** (each chunk of text is processed independently, so this is parallelizable).
  - **Merging partial results** (although merging can be parallelized to some extent, the merging itself is limited by the need to update shared data structures in a thread-safe manner).

Therefore, the **parallel portion** (P) is about **70%** of the code.

Maximum Speedup According to the Number of Cores

Let's calculate the maximum theoretical speedup using **Amdahl's Law**. We assume a machine with **4 cores**.

Speedup = $1 / (S + (P/N)) = 1 / (0.3 + (0.7/4)) \approx 2.1$

Thus, **the maximum theoretical speedup** on a 4-core system is approximately **2.1**×.

Optimal Number of Child Processes or Threads

The optimal number of processes or threads is typically determined by:

- The number of available logical cores.
- The overhead of managing threads or processes.
- The amount of parallel work available.

Based on the following:

- **Amdahl's Law** suggests a diminishing return on speedup after a certain point as the serial portion limits the maximum achievable speedup.
- In practice, adding more processes or threads beyond the number of available logical cores may lead to overhead (context switching, thread/process management, etc.) without significant performance improvement.

In my case, the system has **8 logical cores** (due to hyper-threading), but based on experimental results:

- **2 threads** or **2 processes** provided the best performance.
    - More than **2 threads or processes** did not yield a substantial decrease in execution time.
    - The overhead of managing too many processes or threads outweighed the benefits of parallelization beyond this point.

Therefore, **2 processes or threads** is the optimal configuration, providing the best balance between parallel execution and system resource utilization.

4) Performance Comparison :

4.1 A table that compares the performance of 3 approaches :

Approach	Run 1(ms)	Run 2(ms)	Run 3(ms)	Avg (ms)	Speed up
Naïve	**4110**	**3962**	**11014**	**6355**	**1x**
Multiprocessing	**3911**	**4059**	**4014**	**3994**	**1.59x**
Multithreading	**3684**	**2973**	**3583**	**3413**	**1.86x**

Table 1 : Compares the performance of the 3 approaches

- **Avg (ms) >> average of execution time (ms) .**

Notes:

- **Naive Approach**: Processes the entire dataset sequentially, which results in the highest execution time.

- **Multiprocessing**: Uses multiple processes to divide the workload, resulting in a noticeable speedup compared to the naive approach (1.59×).
- **Multithreading**: Achieved the fastest execution time, with a **1.86× speedup** over the naive approach, due to the efficiency of shared memory and reduced overhead compared to multiprocessing .

5)Observations and Discussion :

5.1 comment on the differences in performance :

Naive Approach:

- **Performance**: The **naive approach** processes the entire dataset sequentially, without any parallelism. It achieved the **highest execution time** (average of **6355 ms**).
- **Reason for Slower Performance**:
  - Since the naive approach does not utilize multiple cores, it cannot take advantage of the parallelism available on modern multi-core systems.
  - All tasks, including reading the file, counting word frequencies, and sorting the results, are done in sequence, leading to slower execution for large datasets.

Multiprocessing Approach:

- **Performance**: The **multiprocessing approach** showed a **significant improvement** over the naive approach, with an average execution time of **3994 ms** (a **1.59× speedup**).
- **Reason for Speedup**:
  - By splitting the dataset into multiple chunks and processing them in parallel across multiple processes, this approach utilized more CPU resources.
  - While processes run independently with separate memory spaces, there is still some overhead in managing multiple processes, which slightly limits the speedup.
  - The speedup observed (1.59×) is a result of parallelism in processing the chunks, but it is not as high as what was achieved with multithreading.

Multithreading Approach:

- **Performance**: The **multithreading approach** achieved the **fastest execution time** with an average of **3413 ms** (a **1.86× speedup** over the naive approach).
- **Reason for Speedup**:

    - **Shared Memory**: Unlike multiprocessing, threads share the same memory space, which significantly reduces the overhead of inter-process communication and memory duplication.
    - **Lower Overhead**: The multithreading approach benefits from lower overhead since threads do not need to create and manage separate memory spaces. This leads to more efficient parallel processing, particularly when the work involves frequent memory access or modifications.
    - **Improved Parallelism**: With 2 threads, the multithreading approach was able to fully utilize the system's available logical cores (8 logical cores), achieving better performance than multiprocessing.

Comparison and Insights:

- **Multiprocessing vs. Multithreading**:
    - **Multithreading** outperforms **multiprocessing** in this case, as threads share the same memory space, which reduces the overhead associated with managing multiple processes.
    - **Multiprocessing** involves additional complexity due to the need for separate memory spaces for each process, leading to more overhead when merging results and managing communication between processes.
    - For this particular dataset and the available number of cores (4 physical cores, 8 logical cores), **multithreading** was the most efficient solution.
- **Diminishing Returns**:
    - The **speedup achieved by multiprocessing and multithreading** is not linear because of the **serial portion** of the code (such as file reading, result merging, and sorting). Amdahl's Law indicates that the maximum speedup is limited by the serial portion of the code.
    - **Optimal number of threads/processes**: After testing, **2 threads or processes** provided the best performance, as increasing the number of threads/processes beyond this point introduced overhead without yielding significant additional benefits.

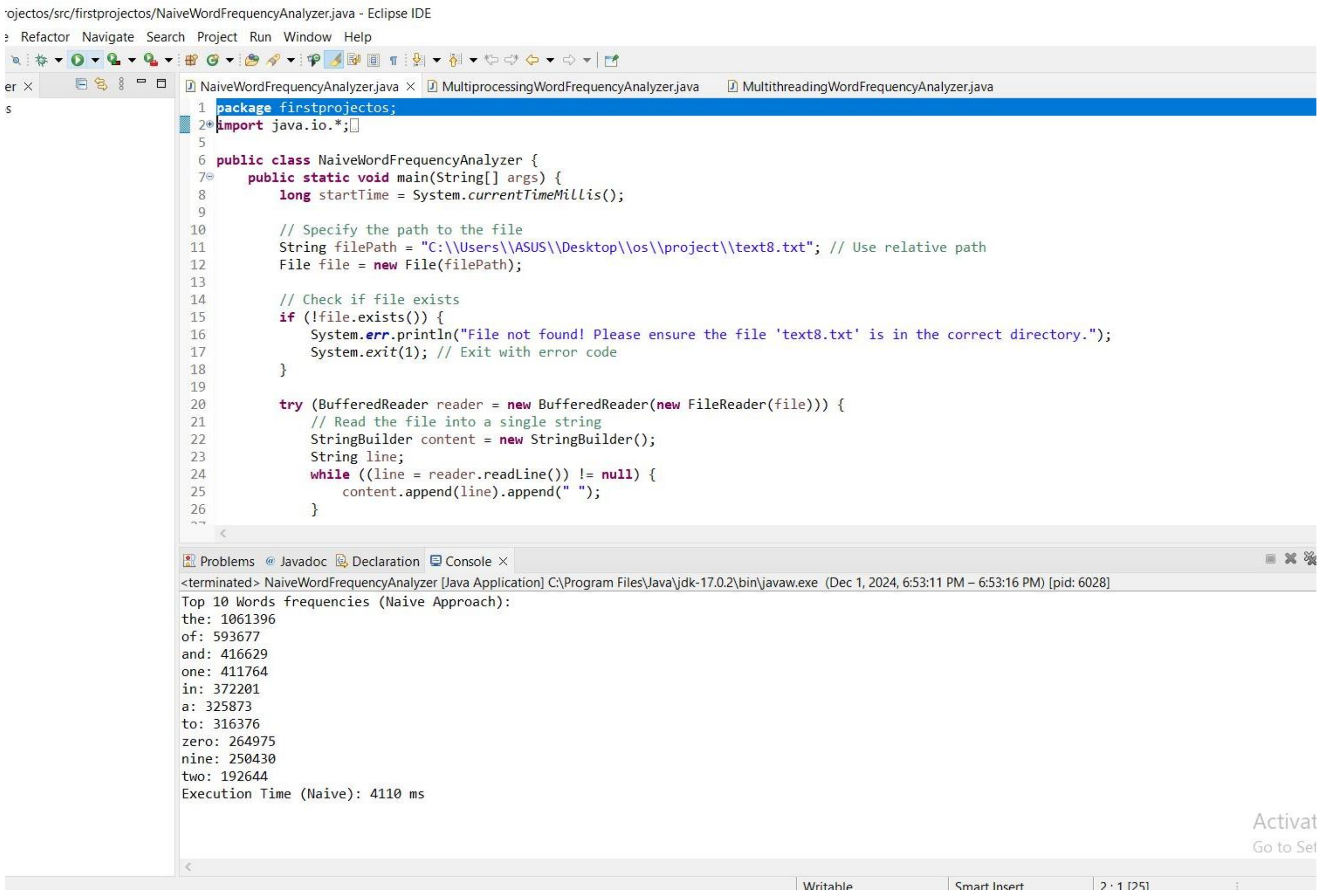
Summary:

- The **naive approach** is inefficient for large datasets due to its sequential nature.
- **Multiprocessing** improves performance by parallelizing the workload, but the overhead of managing separate processes limits the speedup.

- **Multithreading** provides the best performance in this case, as threads share memory and have lower overhead compared to processes, achieving a **1.86× speedup**.

Thus, **multithreading** is the optimal approach for this dataset, considering both speed and resource utilization.

6) Codes :

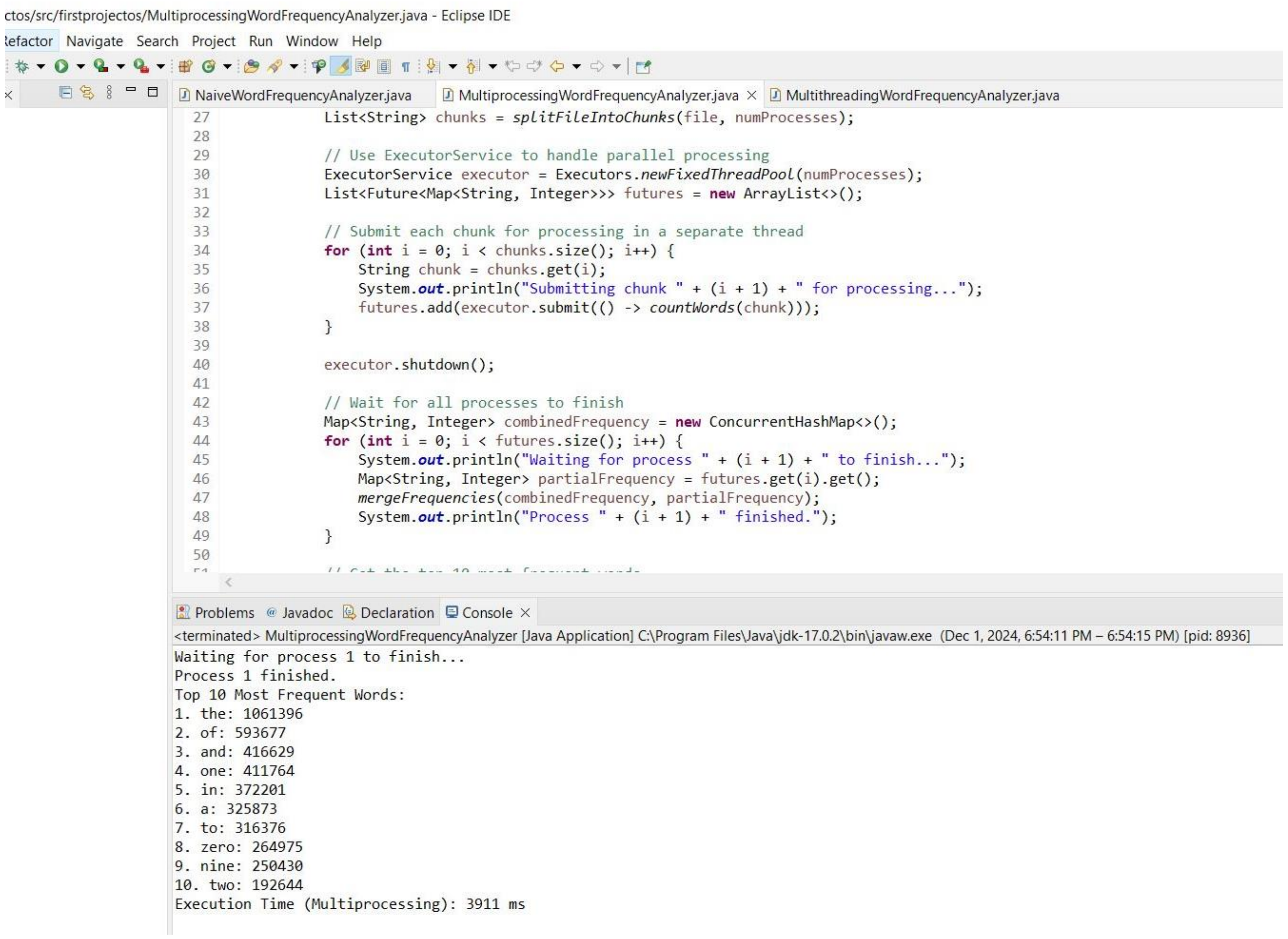


ojectos/src/firstprojectos/NaiveWordFrequencyAnalyzer.java - Eclipse IDE
Refactor Navigate Search Project Run Window Help
NaiveWordFrequencyAnalyzer.java
MultiprocessingWordFrequencyAnalyzer.java
MultithreadingWordFrequencyAnalyzer.java
Problems Javadoc Declaration Console
Writable Smart Insert 2 : 1 [25]

```java
package firstprojectos;
import java.io.*;

public class NaiveWordFrequencyAnalyzer {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        // Specify the path to the file
        String filePath = "C:\\Users\\ASUS\\Desktop\\os\\project\\text8.txt"; // Use relative path
        File file = new File(filePath);

        // Check if file exists
        if (!file.exists()) {
            System.err.println("File not found! Please ensure the file 'text8.txt' is in the correct directory.");
            System.exit(1); // Exit with error code
        }

        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            // Read the file into a single string
            StringBuilder content = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                content.append(line).append(" ");
            }
```

```
<terminated> NaiveWordFrequencyAnalyzer [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe (Dec 1, 2024, 6:53:11 PM – 6:53:16 PM) [pid: 6028]
Top 10 Words frequencies (Naive Approach):
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution Time (Naive): 4110 ms
```

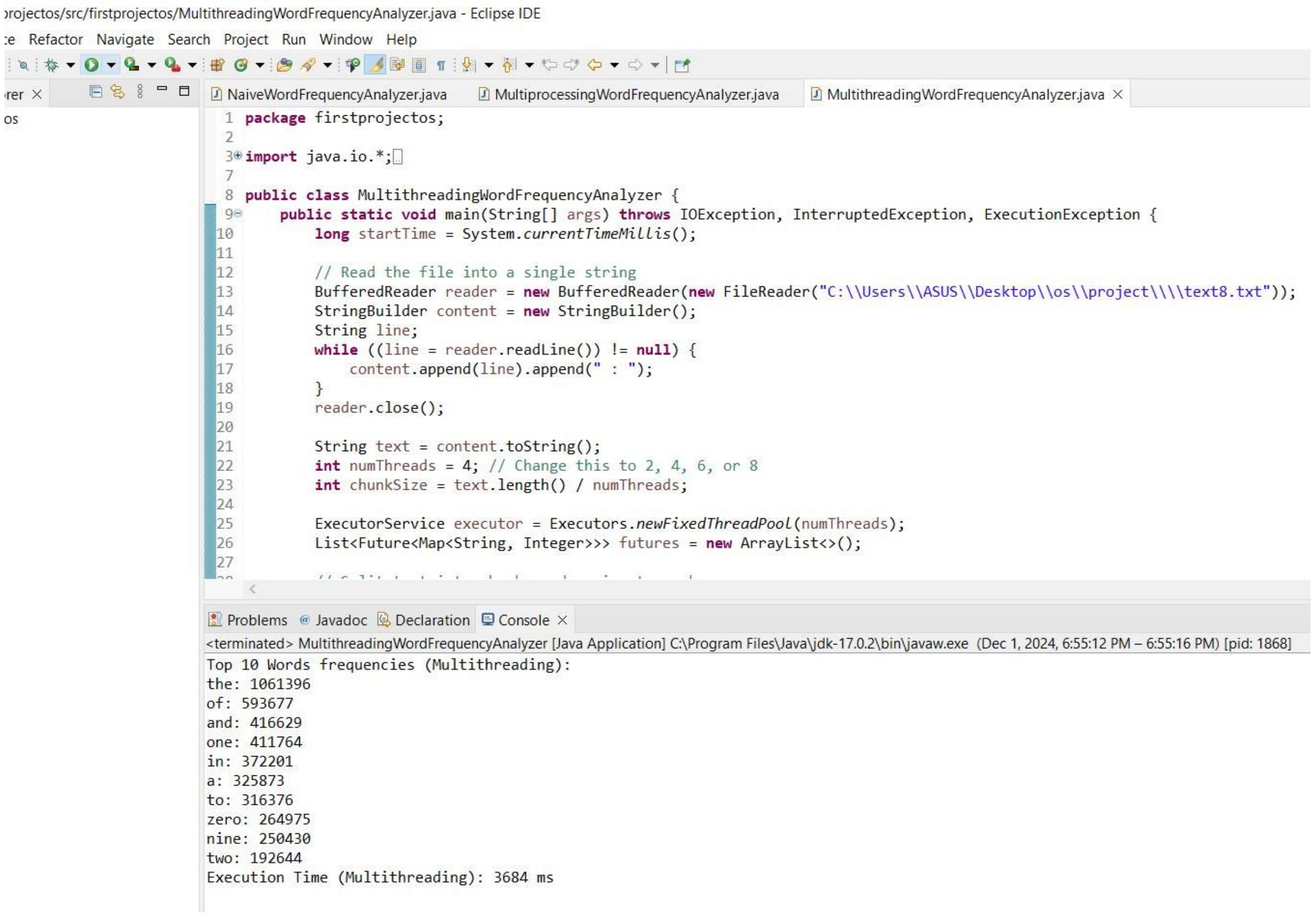
Figure 1 : first execution time of naïve approach


ctos/src/firstprojectos/MultiprocessingWordFrequencyAnalyzer.java - Eclipse IDE
Refactor Navigate Search Project Run Window Help
NaiveWordFrequencyAnalyzer.java
MultiprocessingWordFrequencyAnalyzer.java
MultithreadingWordFrequencyAnalyzer.java
Problems
Javadoc
Declaration
Console

```java
            List<String> chunks = splitFileIntoChunks(file, numProcesses);

            // Use ExecutorService to handle parallel processing
            ExecutorService executor = Executors.newFixedThreadPool(numProcesses);
            List<Future<Map<String, Integer>>> futures = new ArrayList<>();

            // Submit each chunk for processing in a separate thread
            for (int i = 0; i < chunks.size(); i++) {
                String chunk = chunks.get(i);
                System.out.println("Submitting chunk " + (i + 1) + " for processing...");
                futures.add(executor.submit(() -> countWords(chunk)));
            }

            executor.shutdown();

            // Wait for all processes to finish
            Map<String, Integer> combinedFrequency = new ConcurrentHashMap<>();
            for (int i = 0; i < futures.size(); i++) {
                System.out.println("Waiting for process " + (i + 1) + " to finish...");
                Map<String, Integer> partialFrequency = futures.get(i).get();
                mergeFrequencies(combinedFrequency, partialFrequency);
                System.out.println("Process " + (i + 1) + " finished.");
            }
```

```
<terminated> MultiprocessingWordFrequencyAnalyzer [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe (Dec 1, 2024, 6:54:11 PM – 6:54:15 PM) [pid: 8936]
Waiting for process 1 to finish...
Process 1 finished.
Top 10 Most Frequent Words:
1. the: 1061396
2. of: 593677
3. and: 416629
4. one: 411764
5. in: 372201
6. a: 325873
7. to: 316376
8. zero: 264975
9. nine: 250430
10. two: 192644
Execution Time (Multiprocessing): 3911 ms
```

Figure 2 : first execution time of multiprocessing approach



```
package firstprojectos;

import java.io.*;

public class MultithreadingWordFrequencyAnalyzer {
    public static void main(String[] args) throws IOException, InterruptedException, ExecutionException {
        long startTime = System.currentTimeMillis();

        // Read the file into a single string
        BufferedReader reader = new BufferedReader(new FileReader("C:\\Users\\ASUS\\Desktop\\os\\project\\\\text8.txt"));
        StringBuilder content = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            content.append(line).append(" : ");
        }
        reader.close();

        String text = content.toString();
        int numThreads = 4; // Change this to 2, 4, 6, or 8
        int chunkSize = text.length() / numThreads;

        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        List<Future<Map<String, Integer>>> futures = new ArrayList<>();

```

```
<terminated> MultithreadingWordFrequencyAnalyzer [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe (Dec 1, 2024, 6:55:12 PM – 6:55:16 PM) [pid: 1868]
Top 10 Words frequencies (Multithreading):
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution Time (Multithreading): 3684 ms
```

Figure 3 : first execution time of multithreading approach

7) Description and notes about the three approaches :

1. Naive Approach

- **Code Description**: Processes the entire file sequentially in a single thread.
- **Strengths**:

- Simpler implementation.
- Low memory overhead.

- **Weaknesses**:

- slow for large files due to lack of parallelism.

- **Code Implementation**:

- Reads the entire file into memory.
- Counts word frequencies using a HashMap.
- Outputs the top 10 most frequent words.

2. Multiprocessing Approach

- **Code Description**: Splits the file into chunks and processes them in parallel using child processes.
- **Strengths**:
  - Effective use of multi-core CPUs.
  - Ideal for systems with multiple processors.
- **Weaknesses**:
  - Higher memory and resource overhead.
  - File splitting and inter-process communication add complexity.
- **Code Implementation**:
  - Splits the file into N chunks.
  - Uses a fixed thread pool (ExecutorService) to process chunks.
  - Combines the results of all child processes into a single word frequency map.
  - Outputs the top 10 words with their frequencies.

3. Multithreading Approach

- **Code Description**: Splits the file into chunks and processes them using threads within a single process.
- **Strengths**:
  - Lower overhead compared to multiprocessing.

- Can be faster for tasks with significant shared memory access.

- **Weaknesses**:
  - Limited by the Global Interpreter Lock (GIL) in some environments (not an issue in Java).
  - Thread management can be complex for very large numbers of threads.
- **Code Implementation**:
  - Splits the file into N chunks.
  - Uses a fixed thread pool (ExecutorService) to process chunks.
  - Combines results from threads into a shared word frequency map.
  - Outputs the top 10 words with their frequencies.

Comparison Notes

1. **Performance**:
   - **Naive Approach**: Execution time will generally be the longest due to sequential processing.
   - **Multiprocessing Approach**: Execution time reduces as the number of child processes increases (up to a limit).
   - **Multithreading Approach**: Similar performance to multiprocessing but with potentially lower overhead.
2. **Execution Time Measurement**:
   - The System.currentTimeMillis() method used in the code accurately measures the execution time of each approach.
3. **Scalability**:
   - **Naive Approach**: Poor scalability.
   - **Multiprocessing and Multithreading**: Scale better with file size and system resources but may experience diminishing returns as the number of processes/threads increases.
4. **Experimentation**:
   - By varying the number of processes or threads (e.g., 2, 4, 6, 8), you can observe how execution time changes.
   - Optimal performance is typically observed at the point where the workload is evenly distributed without excessive contention.

8) Conclusion :

The objective of this project was to assess the performance of three approaches for word frequency analysis: naive sequential processing, multiprocessing, and multithreading. These approaches were tested on a large dataset (text8.txt), and their performance was compared based on execution time, taking into account the benefits and limitations of parallelism.

Naive Approach:

The naive approach resulted in the longest execution time, processing the dataset sequentially without exploiting parallel computing capabilities. It recorded an average time of 6355 ms.

This approach highlighted the inefficiency of handling large datasets without parallelism, particularly in modern systems equipped with multiple cores.

Multiprocessing:

The multiprocessing approach showed an improvement over the naive method, with an average execution time of 3994 ms, representing a 1.59× speedup.

However, the use of separate processes introduced additional overhead, particularly in merging results and managing memory spaces, which limited the extent of performance improvement.

Multithreading:

The multithreading approach achieved the best performance, with an average execution time of 3413 ms, offering a 1.86× speedup over the naive approach.

Multithreading made more efficient use of the available system resources, especially since threads share memory, which reduces the overhead compared to processes that require separate memory spaces.

Amdahl’s Law and Speedup:

Amdahl’s Law shows that the maximum achievable speedup is constrained by the serial portion of the code. Given that the serial portion (such as file reading, merging results, and sorting) accounted for about 30% of the execution time, the theoretical maximum speedup for a 4-core machine was calculated to be around 2.1×.

The results were consistent with this theory, with the multithreading approach achieving the highest speedup close to this limit.

Optimal Approach and Recommendations:

Based on the findings, the multithreading approach proved to be the most efficient for this dataset, achieving the best balance between speed and resource utilization. The 2 threads configuration provided optimal performance, as adding more threads or processes resulted in diminishing returns.

Multiprocessing also provided significant performance gains but was less efficient due to the higher overhead of managing multiple processes.

For larger datasets or more CPU-intensive tasks, it would be beneficial to explore multithreading or multiprocessing, depending on the nature of the task (shared vs. independent data).

In conclusion, this project demonstrates how parallelism can dramatically improve the performance of data processing tasks. By selecting the right approach and tuning the number of threads or processes, significant time savings can be achieved, especially on multi-core systems.

The multithreading approach was ultimately the most effective for this project, showcasing the potential of shared memory parallelism for efficiently handling large datasets.