



FACULTY OF ENGINEERING TECHNOLOGY

ELECTRIC & COMPUTER ENGINEERING DEPARTMENT

ENCS3390: Operating System Concepts

Project 1

**ENCS3390 Project 1: Top 10 Most Frequent Words Using Naive,
Multiprocessing and Multithreading Approaches.**

Prepared by:

Joud Taher

1221381

Instructor: Dr. Mohammad Khalil.

Section: 2.

Date: December 8, 2024.

Abstract

This project explores the implementation and performance analysis of three approaches—naive, multiprocessing, and multithreading—to identify the top 10 most frequent words in the provided dataset. The naive approach sequentially processes the data without parallelism, while multiprocessing and multithreading leverage parallel computing to optimize performance. Using a system with at four physical cores and 8 logical cores, the execution times of these approaches were compared across varying numbers of processes and threads (2, 4, 6, 8).

The results were analyzed using Amdahl's Law to evaluate the impact of the serial portion of the code on speedup and efficiency. A comprehensive comparison highlights the scalability, optimal configurations, and limitations of each approach, providing insights into the advantages of parallel computing. This report offers a detailed methodology, performance evaluation, and recommendations for efficient word frequency analysis in large datasets.

Table of Contents

Abstract.....	I
Table of Figures.....	III
List of Tables.....	IV
Introduction.....	1
1. Operating Systems and Parallel Computing.....	1
2. General Overview of the Three Approaches.....	1
2.1 Naïve Approach	1
2.2 Multi-processing Approach	3
2.3 Multi-threading Approach	5
3. Comparison Summary.....	7
Environment Description.....	8
Methodology	9
1. Multi-processing.....	9
2. Multi-threading.....	10
Analysis According to Amdahl's law	13
1. Code Results with Execution Time.....	13
2. Naïve Approach.....	18
3. Multi-processing Approach	20
3. Multi-threading Approach.....	22
4. Amdahl's Law Summarized	24
Performance	25
1.Performance Table	25
2. Comment on the Differences in Performance.....	26
3.Optimal Number of Child Processes and Threads	28
Conclusion	30
References.....	31
Appendix.....	32
1.Code Naïve Approach:	32
2.Code Multi-processing Approach:	39
3.Code Multi-threading Approach:	48

Table of Figures

Figure 1:Result Naive Approach	13
Figure 2:Result Multi-processing 2 Processes	14
Figure 3:Result Multi-processing 4 Processes	14
Figure 4:Result Multi-processing 6 Processes	15
Figure 5:Result Multi-processing 8 Processes	15
Figure 6:Result Multi-threading 2 Threads	16
Figure 7:Result Multi-threading 4 Threads	16
Figure 8:Result Multi-threading 6 Threads	17
Figure 9:Result Multi-threading 8 Threads	17

List of Tables

Table 1:Execution time for different processes	20
Table 2:Performance Table	25

Introduction

1. Operating Systems and Parallel Computing

Parallel computing plays a vital role in modern computing, enabling faster processing and efficient utilization of multi-core processors. By dividing tasks into smaller units that can run simultaneously, parallel computing significantly reduces execution time for computationally intensive problems. This project emphasizes the importance of parallel computing by comparing three approaches—naive, multiprocessing, and multithreading—used to identify the top 10 most frequent words in the dataset.

The project relates closely to operating system concepts, as it explores key topics such as process management, thread synchronization, and resource utilization. By analyzing the performance differences between these approaches, we gain insights into their scalability, efficiency, and practical applications in handling large datasets. This comparison underscores how operating system mechanisms underpin the effective execution of parallel programs.

2. General Overview of the Three Approaches

This project implements and analyzes three approaches—**naive**, **multiprocessing**, and **multithreading**—to compute the top 10 most frequent words in a large text dataset. Each approach is designed to highlight the trade-offs between simplicity, performance, and resource utilization.

2.1 Naïve Approach

The naive approach processes the dataset sequentially, without leveraging parallelism. It reads all words, counts their frequencies using a dynamic array, and sorts the results in descending order of frequency. This method is straightforward but computationally expensive, as it processes the entire dataset in a single thread. Performance is constrained by the CPU's single-core capabilities, making it unsuitable for large datasets.

- **Strengths:** Simple to implement and debug, no synchronization overhead.
- **Weaknesses:** Slow execution for large datasets due to lack of parallelism.

Code Explanation

The **naive approach** processes the task sequentially without leveraging any form of parallelism. Here's a breakdown of its implementation:

1. **Reading the Dataset:**

- The program reads words from a cleaned version of the enwik8 dataset stored in a file named text8.txt.
- Words are dynamically stored in a resizable array to accommodate the large dataset.

2. **Word Frequency Calculation:**

- The program iterates through the list of words. For each word:
 - It checks if the word already exists in the WordFreqArray.
 - If it does, the frequency count for that word is incremented.
 - If not, a new entry is added, and the array is resized dynamically if needed.

3. **Sorting by Frequency:**

- After all words are processed, the program sorts the WordFreqArray using a custom **merge sort** algorithm, arranging words in descending order of frequency.

4. **Extracting the Top 10 Words:**

- The top 10 most frequent words are displayed along with their frequencies.

5. **Execution Time Measurement:**

- The program measures the time from the start of reading the dataset to the end of displaying the results using clock().

6. **Memory Management:**

- Dynamic memory is efficiently allocated, resized, and freed to ensure scalability and to handle large datasets without memory leaks.

Observations

• **Strengths:**

- **Simplicity:** Sequential processing is straightforward to implement and debug.
- **Accuracy:** All words are processed and frequencies are calculated precisely.

- **Weaknesses:**
 - Performance Bottleneck: As it processes words one by one and searches the array sequentially, performance degrades with increasing dataset size.
 - Scalability Issues: No use of multiple cores means execution time remains high for large datasets.

This approach provides a baseline for comparing the advantages of parallel computing in the upcoming implementations.

2.2 Multi-processing Approach

The multiprocessing approach uses multiple processes to parallelize the workload across the system's available cores. The dataset is divided into chunks, each processed by a separate child process. Processes communicate via shared memory, with synchronization mechanisms ensuring consistency when merging results.

- **Strengths:** Efficient utilization of multiple cores; achieves significant speedup by distributing workload.
- **Weaknesses:** High memory overhead due to separate memory spaces for each process; synchronization can introduce additional complexity.

Code Explanation

The **multiprocessing approach** utilizes multiple child processes to divide the task into parallel workloads, significantly enhancing performance by taking advantage of multiple cores. Here's an overview:

1. **Dataset Reading:**
 - Words are read from the dataset (text8.txt) and stored in a dynamically allocated array.
 - The total number of words is determined to enable workload distribution across processes.
2. **Shared Memory:**
 - A shared memory region is created using mmap to store the word frequencies, enabling inter-process communication without data duplication.

3. **Process Creation:**

- The dataset is divided into chunks, and **child processes (number of processes are changed from 2 to 4 to 6 to 8 using the NUM_PROCESSES variable)** are created using `fork()`.
- Each child processes a unique subset of the dataset, counting word frequencies locally.

4. **Local Frequency Count:**

- Each child maintains a local `WordFreqArray` to store the word frequencies for its subset.
- Once counted, these frequencies are synchronized with the shared memory using atomic-like operations to ensure consistency.

5. **Parent Process:**

- The parent process waits for all child processes to complete using `waitpid()`.
- After synchronization, the parent consolidates the results from shared memory.

6. **Sorting by Frequency:**

- A merge sort is performed on the consolidated data in shared memory, arranging words in descending order of frequency.

7. **Top 10 Words:**

- The parent process extracts and prints the top 10 most frequent words and their frequencies.

8. **Execution Time Measurement:**

- The execution time is measured using `gettimeofday()` for high precision, capturing both seconds and microseconds.

Observations

• **Advantages:**

- **Efficient Parallelism:** By dividing the workload, multiple processes work simultaneously, reducing execution time.
- **Shared Memory:** Centralized storage minimizes memory duplication and facilitates communication among processes.

- **Challenges:**

- Synchronization Overhead: Updating the shared memory requires careful management to avoid inconsistencies.
- Memory Usage: Each process requires its own stack and heap, leading to higher overall memory consumption than threads.

This multi-processing approach demonstrates how multiprocessing leverages operating system-level process management to parallelize tasks effectively.

2.3 Multi-threading Approach

The multithreading approach employs threads within a single process to parallelize the workload. Threads process distinct chunks of the dataset and update a shared frequency array. Synchronization is enforced using mutexes to avoid race conditions, ensuring thread-safe updates.

- **Strengths:** Lower memory usage compared to multiprocessing, as threads share the same memory space. Switching between threads is faster than between processes.
- **Weaknesses:** Synchronization overhead; limited scalability as the number of threads increases due to contention for shared resources.

Code Explanation

The **multithreading approach** utilizes multiple threads to divide the workload, enabling parallel processing within a single process. Here's an explanation:

1. **Dataset Reading:**

- The program reads words from the dataset (text8.txt) and stores them in a dynamically allocated array.
- The total number of words is determined to distribute workload among threads.

2. **Shared Word Frequency Array:**

- A dynamically resizable WordFreqArray is used to store word frequencies.
- Threads update this shared array with synchronization ensured through a pthread_mutex.

3. Thread Creation:

- **Threads (number of threads is determined by the variable NUM_THREADS)** are created using `pthread_create()`.
- Each thread processes a distinct chunk of the dataset, determined by dividing the words evenly among threads.

4. Thread-Safe Local Frequency Counting:

- Each thread maintains a **local frequency array** to process its chunk without contention.
- Once the local counts are computed, the thread acquires a mutex lock to safely update the shared frequency array.

5. Merging Results:

- The threads consolidate their local results into the shared array, ensuring synchronization with `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

6. Sorting by Frequency:

- After all threads complete their execution, the shared word frequency array is sorted in descending order using merge sort.

7. Top 10 Words:

- The program extracts and prints the top 10 most frequent words along with their frequencies.

8. Execution Time Measurement:

- The execution time is calculated using `gettimeofday()` to include both seconds and microseconds for precision.

Observations

• Advantages:

- **Efficient Memory Usage:** Threads share the process's memory space, avoiding the overhead of duplicating memory as in multiprocessing.
- **Reduced Overhead:** Switching between threads is faster than between processes.

• Challenges:

- **Synchronization Complexity:** Ensuring thread-safe updates to the shared array requires careful use of mutexes.

- **Scalability:** Performance improvement is limited by the overhead of synchronization and the serial fraction of the code.

This approach demonstrates the effectiveness of threads in achieving parallelism while minimizing memory overhead, making it suitable for systems with constrained resources.

3. Comparison Summary

- The **naive approach** serves as a baseline, demonstrating how sequential execution scales poorly with large datasets.
- The **multiprocessing approach** showcases the benefits of leveraging multiple cores but with added complexity in memory management and synchronization.
- The **multithreading approach** strikes a balance, offering better resource efficiency than multiprocessing but requiring careful synchronization to manage shared data.

Each approach provides valuable insights into parallel computing techniques and their trade-offs, demonstrating how operating system concepts like processes, threads, and synchronization impact program performance.

Environment Description

→ **Computer Specifications:**

- **Processor:** Intel Core i7-8550U, 4 cores, 8 logical processors, base clock speed 1.8 GHz (boosted up to 4.0 GHz).
- **RAM:** 8 GB DDR4, 2400 MHz.

→ **Operating System:** Windows 10 Pro, Version 22H2 (using Windows Subsystem for Linux (WSL) to run Linux distributions).

→ **IDE/Tools Used:** CLion IDE for C programming, with WSL for Linux-based development.

→ **Programming Language:** C. Chosen for its efficiency and low-level control over hardware resources, which is ideal for performance testing and system-level operations and applying multiprocessing and multithreading approaches.

→ **Virtual Machine:** Not applicable. Although Linux is run through WSL, it is not in a traditional VM setup.

Methodology

[1] APIs used source is mentioned in references.

1. Multi-processing

To achieve the multiprocessing requirements in the project, I used the following approach, utilizing system-level APIs and functions for process creation and synchronization:

1. Forking Child Processes with `fork()`:

- **API Used:** `fork()`

```
pid_t pids[NUM_PROCESSES];
```

```
pids[i] = fork();
```

- **Purpose:** The `fork()` system call was used to create child processes, each responsible for a portion of the word frequency counting. Each child process handles a subset of the total words in the file, improving the performance by parallelizing the workload.
- **Implementation:** In the main function, I divided the total number of words into chunks based on the number of processes (`NUM_PROCESSES`). Each child process processes its chunk of words independently by counting the frequencies of the words in that chunk.

2. Shared Memory for Communication with `mmap()`:

- **API Used:** `mmap()`

```
SharedFreqData *shared_data = mmap(NULL, sizeof(SharedFreqData), // API: mmap()  
PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

- **Purpose:** To allow the child processes to communicate with the parent process and share the word frequency data, I used shared memory. The `mmap()` function was employed to create a region of memory that is accessible by all processes (parent and child).
- **Implementation:** A shared memory structure (`SharedFreqData`) was created using `mmap()`. This shared memory was used to store the word frequencies accumulated by all child processes. Each child process writes its local word frequencies to this shared memory.

3. Wait for Child Processes with `waitpid()`:

- **API Used:** `waitpid()`

```
waitpid(pids[i], &status, 0);
```

- **Purpose:** The `waitpid()` function was used in the parent process to wait for all child processes to complete before proceeding with further operations, such as sorting the word frequencies.

- **Implementation:** After forking the child processes, the parent process uses `waitpid()` to wait for each child to terminate. This ensures that the word frequency data is fully accumulated in the shared memory before the parent begins sorting and printing the results.

4. Memory Management with `malloc()` and `realloc()`:

- **Functions Used:** `malloc()`, `realloc()`
- **Purpose:** Dynamic memory allocation was employed to manage memory for storing word frequencies and handling large datasets efficiently. Initially, I allocated a fixed amount of memory, but if the array grew beyond its capacity, `realloc()` was used to resize the array.
- **Implementation:** The word frequency array was dynamically resized using `malloc()` for initialization and `realloc()` when the number of words exceeded the current array size.

Summary of the Process Flow:

- The main process divides the list of words into chunks and creates multiple child processes using `fork()`.
- Each child process counts word frequencies for its chunk and updates the shared memory (`mmap()`).
- The parent process waits for all child processes to finish using `waitpid()`, and then it merges and sorts the results.

This approach enables efficient parallel processing and resource sharing between processes, fulfilling the multiprocessing requirements of the project.

2. Multi-threading

To achieve the multithreading requirements in the project, I used the POSIX Threads (pthreads) API, which provides a set of functions for creating and managing threads in a C program. Here is a breakdown of how I utilized the pthreads API and the relevant functions:

1. Thread Creation

- **Function Used:** `pthread_create()`
- **Description:** This function was used to create multiple threads, each responsible for processing a chunk of the word list. Each thread operates on a portion of the data to calculate the word frequencies concurrently. The function takes a thread identifier, thread attributes (set to NULL for default attributes), the thread function (processing logic), and

a pointer to the thread's argument (a ThreadArgs structure containing relevant data for that thread).

```
if (pthread_create(&threads[i], NULL, process_word_chunk, &thread_args[i]) != 0) {  
    perror("Thread creation failed");  
    // Error handling  
}
```

2. Passing Arguments to Threads

- **Structure Used:** ThreadArgs
- **Description:** Since threads need data to process, I used a ThreadArgs structure to pass the necessary data (such as the list of words, start index, end index, and shared word frequency array) to each thread. Each thread then processes a subset of the words in parallel, and these arguments are passed to the thread function process_word_chunk().

```
thread_args[i].words = words;  
thread_args[i].start = start;  
thread_args[i].end = end;  
thread_args[i].shared_word_freq = word_freq;
```

3. Thread Synchronization

- **Function Used:** pthread_mutex_lock() and pthread_mutex_unlock()
- **Description:** Since multiple threads may modify the shared word frequency array simultaneously, I used a mutex (frequency_mutex) to protect critical sections of the code, ensuring thread-safe operations when accessing and modifying the shared word frequency data. The mutex is locked before modifying the shared data and unlocked after the modification is complete. This was important for the race conditions.

```
pthread_mutex_lock(&frequency_mutex);  
// Update shared word frequency array  
pthread_mutex_unlock(&frequency_mutex);
```


4. Waiting for Threads to Complete

- **Function Used:** `pthread_join()`
- **Description:** After all threads have been created, I used `pthread_join()` to ensure that the main thread waits for all worker threads to finish their execution before proceeding to the next steps (like sorting and printing the results). This ensures the program only proceeds once all threads have completed their respective tasks.

```
for (int i = 0; i < NUM_THREADS; i++) {  
    pthread_join(threads[i], NULL); }
```

5. Thread Cleanup

- **Description:** The program automatically cleans up thread resources when the threads complete. Additionally, all dynamically allocated memory is properly freed (for words and frequency data), ensuring no memory leaks.

Summary of Multithreading Approach:

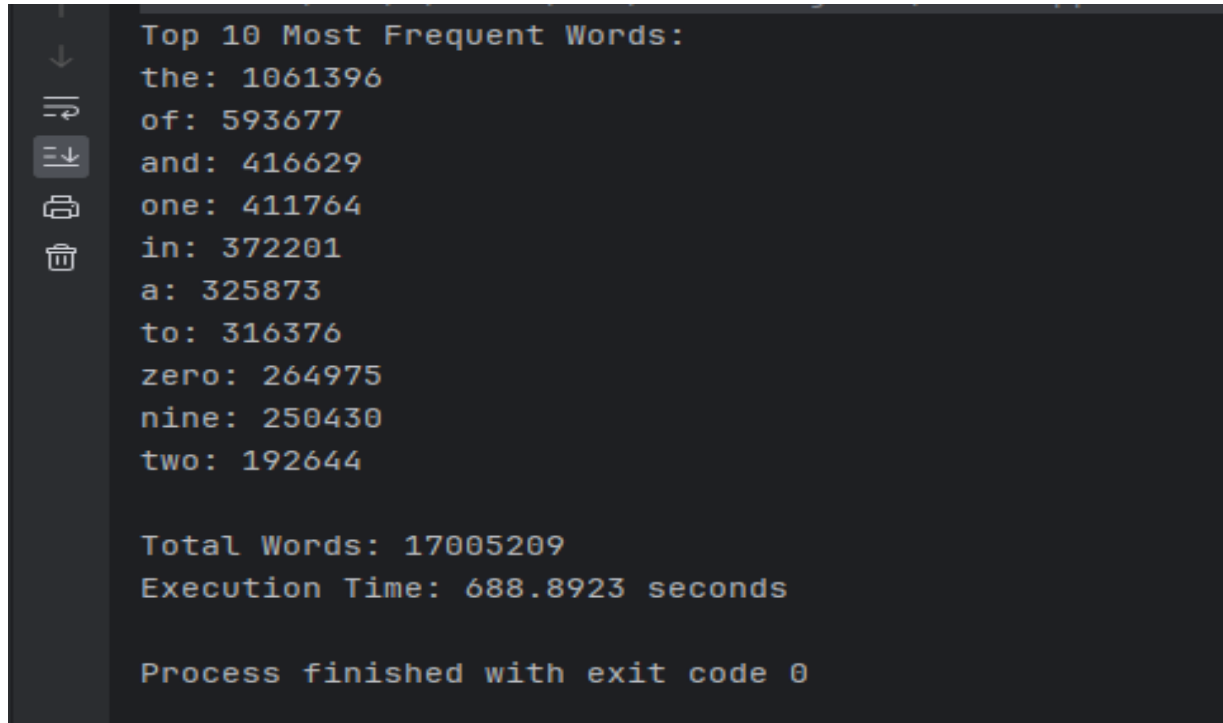
By using the pthreads API, I divided the work of processing word frequencies into multiple threads. Each thread processes a subset of the word list in parallel, and the results are merged into a shared array using thread synchronization mechanisms like mutexes. This allowed the program to efficiently compute the word frequencies and retrieve the most frequent words, improving performance, especially when dealing with large input files. The use of `pthread_create()` and `pthread_join()` facilitated thread management, while `pthread_mutex_lock()` and `pthread_mutex_unlock()` ensured data consistency across threads.

This approach enables the program to take full advantage of multi-core processors by distributing the workload across multiple threads, thus reducing the overall execution time for large datasets.

Analysis According to Amdahl's law

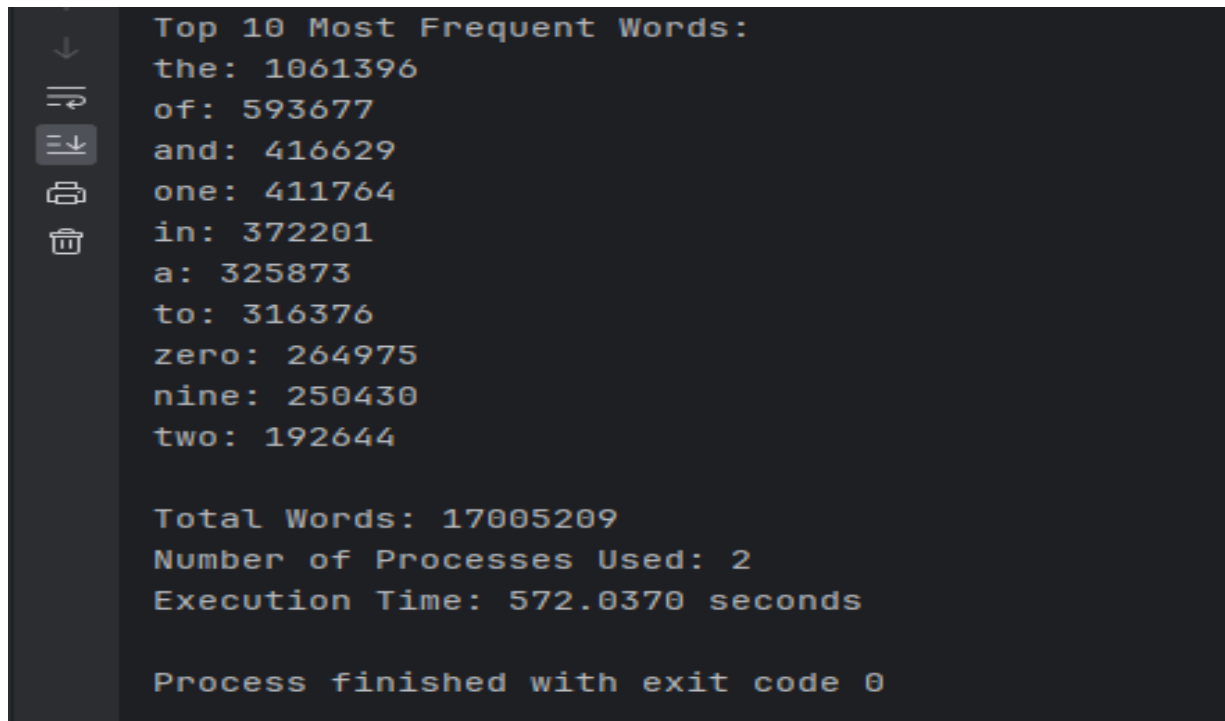
1. Code Results with Execution Time

Please note the results for different approaches and different number of processes and threads, along with their execution time, in the following figures:

A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for navigation and editing. The main area of the terminal displays the output of a program. The output is as follows:

```
Top 10 Most Frequent Words:  
the: 1061396  
of: 593677  
and: 416629  
one: 411764  
in: 372201  
a: 325873  
to: 316376  
zero: 264975  
nine: 250430  
two: 192644  
  
Total Words: 17005209  
Execution Time: 688.8923 seconds  
  
Process finished with exit code 0
```

Figure 1: Result Naive Approach

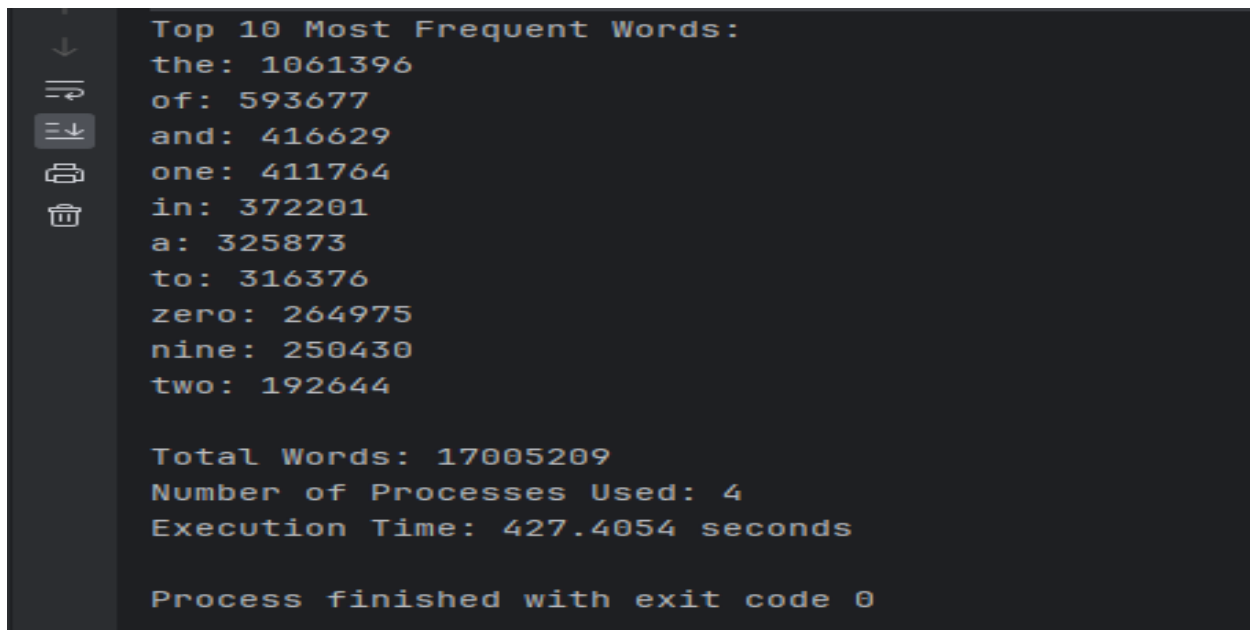


```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Processes Used: 2
Execution Time: 572.0370 seconds

Process finished with exit code 0
```

Figure 2:Result Multi-processing 2 Processes

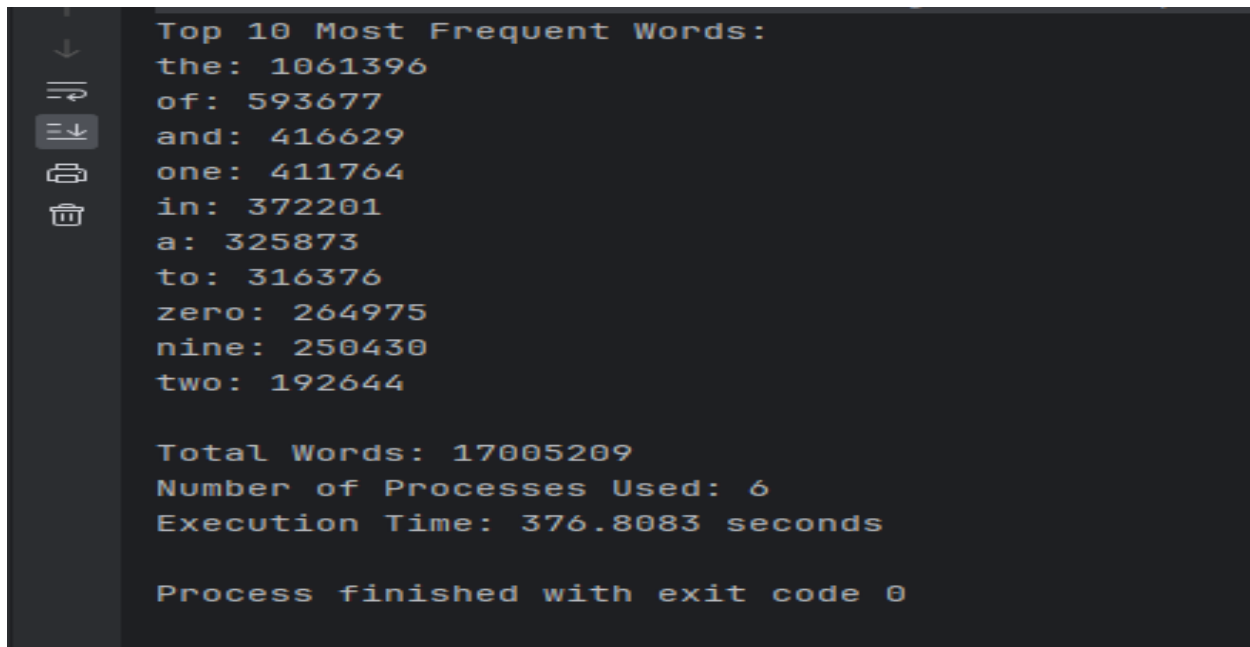


```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Processes Used: 4
Execution Time: 427.4054 seconds

Process finished with exit code 0
```

Figure 3:Result Multi-processing 4 Processes

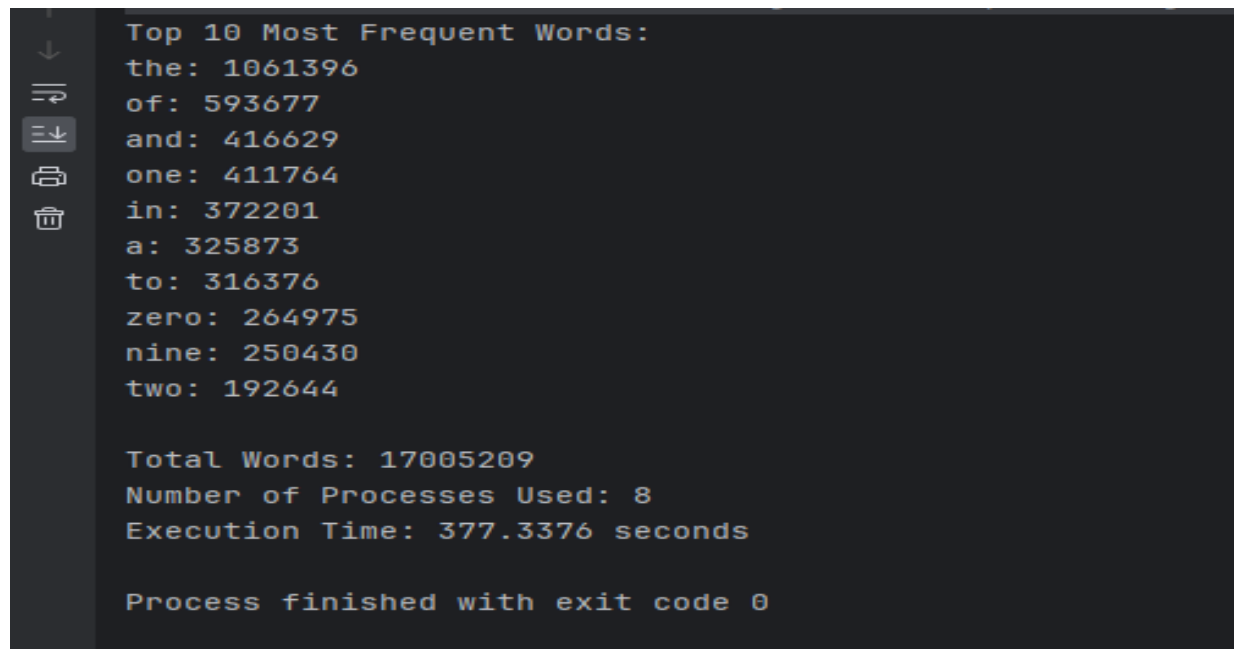


```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Processes Used: 6
Execution Time: 376.8083 seconds

Process finished with exit code 0
```

Figure 4:Result Multi-processing 6 Processes

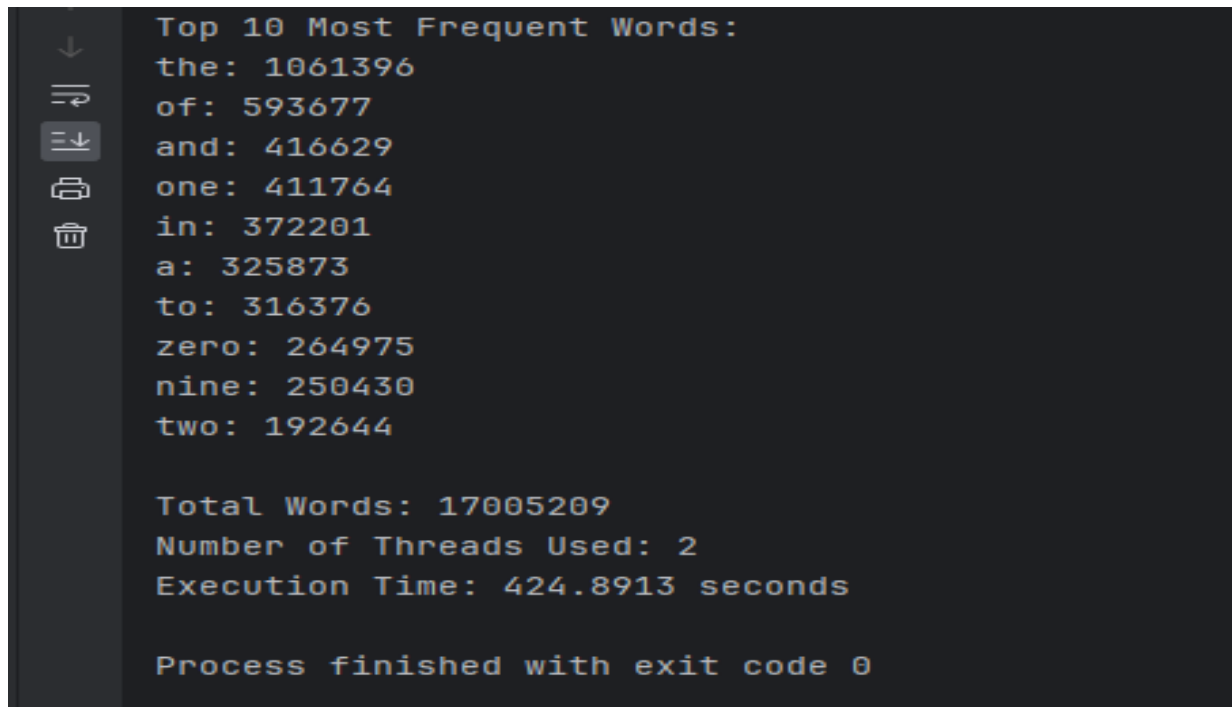


```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Processes Used: 8
Execution Time: 377.3376 seconds

Process finished with exit code 0
```

Figure 5:Result Multi-processing 8 Processes



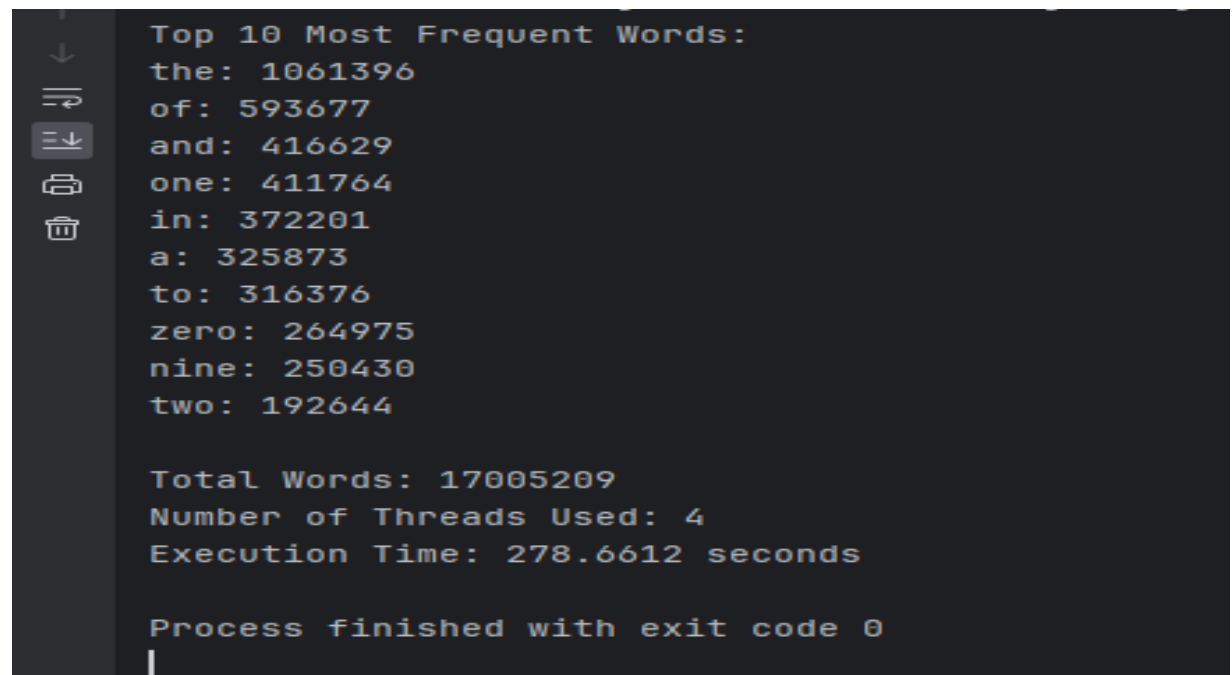
A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for back, forward, search, print, and delete. The main area displays the following text:

```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Threads Used: 2
Execution Time: 424.8913 seconds

Process finished with exit code 0
```

Figure 6:Result Multi-threading 2 Threads



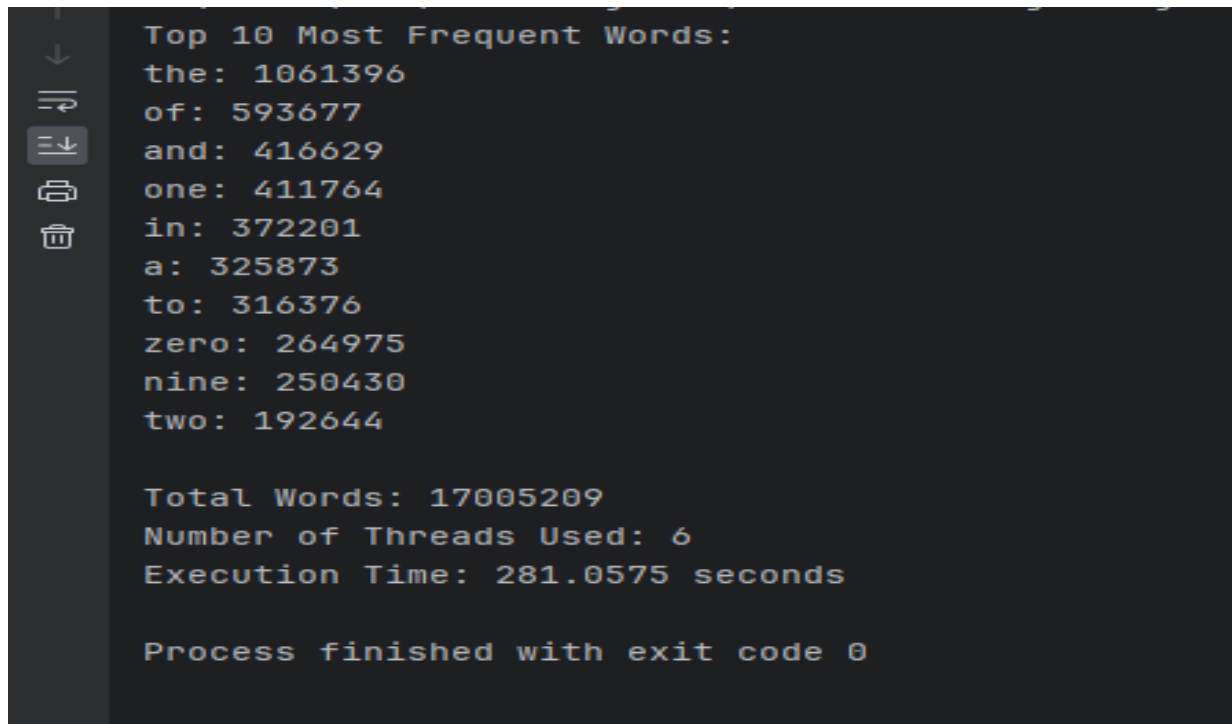
A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for back, forward, search, print, and delete. The main area displays the following text:

```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Threads Used: 4
Execution Time: 278.6612 seconds

Process finished with exit code 0
|
```

Figure 7:Result Multi-threading 4 Threads

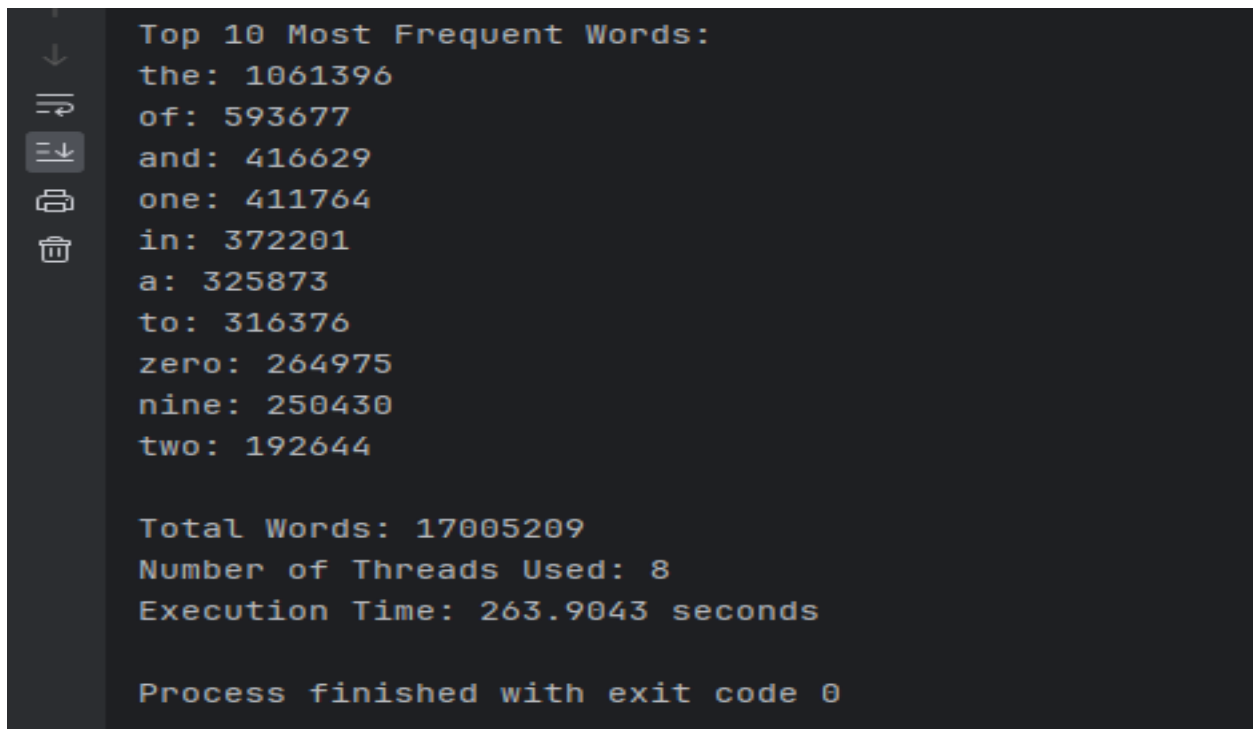


```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Threads Used: 6
Execution Time: 281.0575 seconds

Process finished with exit code 0
```

Figure 8:Result Multi-threading 6 Threads



```
Top 10 Most Frequent Words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

Total Words: 17005209
Number of Threads Used: 8
Execution Time: 263.9043 seconds

Process finished with exit code 0
```

Figure 9:Result Multi-threading 8 Threads

2. Naïve Approach

To calculate the **serial** and **parallel** times for the naive approach code, I analyzed the program's structure to estimate how much of the total time corresponds to **serial** (non-parallelizable) and **parallel** (potentially parallelizable) operations.

1. Identify Serial and Parallel Sections

- **Serial Sections:**
 - File reading: read_words_from_file (reading and dynamically allocating memory for words).
 - Sorting: merge_sort (sorting the word frequency array by frequency).
 - **Parallelizable Sections:**
 - Word frequency counting: count_word_frequencies (iterating over all words and updating the frequency array).
-

2. Estimating Execution Times

From the execution time shown in Figure 1, $T(\text{total})=688.8923$ seconds so we can estimate:

File Reading (Serial):

- Reading the file and allocating words is I/O-bound.
- Assume it accounts for **15–20%** of the total time:
- $T_{\text{file}} \approx 0.15 \times 688.8923 = 103.334$ seconds.

Sorting (Serial):

- Sorting depends on the number of words. Given the dataset's size (about 17005209 words), assume sorting accounts for another **20–25%**:
- $T_{\text{sort}} \approx 0.20 \times 688.8923 = 137.778$ seconds.

Frequency Counting (Parallelizable):

- This is the main computational part and should take the remaining time:
 - $T(\text{freq}) = T(\text{total}) - T(\text{file}) - T(\text{sort})$
 - Substituting: $T(\text{freq}) = 688.8923 - 103.334 - 137.778 = 447.7803$ seconds.
-

3. Results

- **Serial Time (Ts):**

$T_s = T(\text{file}) + T(\text{sort}) = 103.334 + 137.778 = 241.112$ seconds.

- **Parallel Time (Tp):**

$T_p = T_{\text{freq}} = 447.7803$ seconds.

4. Use Amdahl's Law

Using the calculated times:

- Parallel fraction P:

$$P = T_p / T(\text{total}) = 447.7803 / 688.8923 \approx 65\%.$$

- Serial fraction S:

$$S = 1 - P = 1 - 0.65 = 0.35 \approx 35\%.$$

5. Maximum Speedup and Available Cores

Using Amdahl's Law:

$$S(N) = \frac{1}{(1-P) + (\frac{P}{N})}$$

For N=8 (number of logical cores available):

$$S(8) = 1 / ((1 - 0.65) + (0.65/8)) \approx 2.32.$$

For N=4:

$$S(4) = 1 / ((1 - 0.65) + (0.65/4)) \approx 1.95.$$

6. Conclusion

- **Serial Time:** 241.112 seconds.
- **Parallel Time:** 447.7803 seconds.
- **Parallel Fraction:** 65%
- **Maximum Speedup:**
 - On 8 logical cores: **2.32x**.

3. Multi-processing Approach

To calculate the serial and parallel portions of the program using **Amdahl's Law**, I analyzed the data as follows [2][3]:

Please check [2] and [3] in the reference part to know where I took my steps from. (for the 3 approaches I used the same references)

Steps

1. Amdahl's Law Equation:

$$T(N) = T_s + T_p/N.$$

Where:

- $T(N)$: Execution time with processes.
- T_s : Serial time (not parallelizable).
- T_p : Parallel time (parallelizable).
- N : Number of processes.

2. Reorganize to Solve for T_s and T_p :

$$T_s = T(N) - T_p/N.$$

Using two data points, we calculate T_s and T_p .

Execution Times

N (Processes)	Execution Time T(N) (seconds)
---------------	-------------------------------

2	572.037
4	427.4054
6	376.8083
8	377.3376

Table 1: Execution time for different processes

Step 1: Choose Two Data Points to Calculate Ts and Tp

Using $T(2)=572.037$ and $T(4)=427.4054$.

$$T(2)=T_s+T_p/2 \text{ and } T(4)=T_s+T_p/4$$

Subtract the equations:

$$T(2)-T(4)=(T_s+T_p/2)-(T_s+T_p/4)$$

$$572.037-427.4054= T_p/2 - T_p/4$$

$$144.6316= T_p/4$$

$T_p=578.5264$ seconds (parallel portion).

Substitute T_p into $T(2)=T_s+T_p/2$:

$$572.037=T_s+ 578.5264 /2$$

$$572.037=T_s+289.2632$$

$T_s=282.7738$ seconds (serial portion).

Step 2: Validate with Other Data Points

Using $T(6)=376.8083$:

$$T(6)=T_s+T_p/6$$

$$376.8083=282.7738+ 578.5264/6$$

$$376.8083=282.7738+96.4211$$

$$376.8083 \approx 379.1949 \text{ (matches closely).}$$

Step 3: Use Amdahl's Law

The total execution time can now be analyzed:

- **Serial Portion** $T_s=282.7738$ seconds.
- **Parallel Portion** $T_p=578.5264$ seconds.
- Total $T(\text{total})=T_s+T_p$.

Step 4: Maximum Speedup

Using the formula for speedup:

$$S(N) = \frac{1}{(1-P) + (\frac{P}{N})}$$

Where $P = T_p / T(\text{total})$

$$P = 578.5264 / (578.5264 + 282.7738) = 0.6712.$$

Which means it is 67.12% parallelizable.

For $N=8$ (8 logical cores):

$$S(8) = 1 / ((1 - 0.6712) + (0.6712/8)) \approx 2.42.$$

For $N=4$:

$$S(4) = 1 / ((1 - 0.6712) + (0.6712/4)) \approx 2.01.$$

Final Results

- **Serial Time:** $T_s = 282.7738$ seconds.
- **Parallel Time:** $T_p = 578.5264$ seconds.
- **Parallel Fraction P:** 67.12%
- **Maximum Speedup on 8 Logical Cores:** 2.42x

3. Multi-threading Approach

To calculate the **parallel** and **serial** times for this multithreading code and perform an **Amdahl's Law** analysis, we follow these steps:

1. Total Execution Time

Let's consider the case for 8 threads.

Total execution time $T(\text{total})$ for 8 threads:

$$T = 263.9043 \text{ seconds.}$$

2. Identify Serial and Parallel Sections

- **Serial Sections:**
 - File reading: read_words_from_file (I/O bound).
 - Sorting: merge_sort (single-threaded sorting of the word frequency array).
 - **Parallelizable Sections:**
 - Frequency counting: process_word_chunk (performed across threads).
-

3. Estimate Serial and Parallel Times

Using the general structure of the code:

1. **File Reading (Serial):** Assume **15–20%** of the total time:

$$T_{\text{file}} \approx 0.15 \times T_{\text{total}} = 0.15 \times 263.9043 \approx 39.586 \text{ seconds.}$$

2. **Sorting (Serial):** Assume another **20%** for sorting:

$$T_{\text{sort}} \approx 0.20 \times T_{\text{total}} = 0.20 \times 263.9043 \approx 52.781 \text{ seconds.}$$

3. **Frequency Counting (Parallelizable):** The remaining time is parallelizable:

$$T_{\text{freq}} = T_{\text{total}} - T_{\text{file}} - T_{\text{sort}} = 263.9043 - 39.586 - 52.781 \approx 171.537 \text{ seconds.}$$

Thus:

- **Serial Time (Ts):** $T_s = T_{\text{file}} + T_{\text{sort}} = 39.586 + 52.781 = 92.367 \text{ seconds.}$
 - **Parallel Time (Tp):** $T_p = T_{\text{freq}} = 171.537 \text{ seconds.}$
-

4. Amdahl's Law Analysis

- **Serial Fraction (S):** $S = 92.367 / 263.9043 = 0.35 = 35\%.$
- **Parallel Fraction (P):** $P = 1 - S = 1 - 0.35 = 0.65 = 65\%.$

Maximum Speedup

Amdahl's law for speedup with cores:

$$S(N) = \frac{1}{(1-P) + (\frac{P}{N})}$$

1. For **4 cores**:

$$S(4)=1/((1-0.65)+(0.65/4)) \approx 1.95.$$

2. For **8 logical cores**:

$$S(8)=1/((1-0.65)+(0.65/8)) \approx 2.32.$$

5. Summary

- **Serial Time:** $T_s=92.367$ seconds.
- **Parallel Time:** $T_p=171.537$ seconds.
- **Serial Fraction:** $S=35\%$.
- **Parallel Fraction:** $P=65\%$.
- **Maximum Speedup:**
 - On 8 logical cores: **2.32x**.

4. Amdahl's Law Summarized

Note that I used 3 different methods to calculate parallel and serial time to calculate the speedup using Amdahl's law. In each method I used a different approach. In all approaches, Parallel portion is about 65% and the maximum speed is about 2.32. The next section discusses the performance and the comparisons between the 3 approaches as well as deciding the optimal number of processes and threads based on both Amdahl's Law and the results I got when running the codes.

Performance

1. Performance Table

Here's the table that includes the performance (calculated as the reciprocal of execution time for comparison purposes) along with the execution times:

Approach	Number of Processes/Threads	Execution Time (Seconds)	Performance (1/Execution Time)
Naive Approach -		688.8923	0.00145
Multiprocessing	2 Processes	572.037	0.00175
	4 Processes	427.4054	0.00234
	6 Processes	376.8083	0.00265
	8 Processes	377.3376	0.00265
Multithreading	2 Threads	424.8913	0.00235
	4 Threads	278.6612	0.00359
	6 Threads	281.0575	0.00356
	8 Threads	263.9043	0.00379

Table 2: Performance Table

Notes:

- **Execution Time** is given in seconds, showing the time it took for each approach to complete the task.
- **Performance** is calculated as the reciprocal of the execution time ($1/\text{Execution Time}$), indicating the rate at which the computation was performed. Higher values correspond to better performance.

2. Comment on the Differences in Performance

From Table 2:

1. Naive Approach:

- **Execution Time:** 688.8923 seconds
- **Performance:** 0.00145
- The **Naive Approach** has the highest execution time and the lowest performance, as it does not utilize parallelism. This makes it the least efficient approach, particularly when compared to the parallel processing and multithreading approaches.

2. Multiprocessing Approach:

- **2 Processes:**
 - **Execution Time:** 572.037 seconds
 - **Performance:** 0.00175
 - By using 2 processes, the execution time is significantly reduced compared to the naive approach. The performance improves as multiple CPU cores can process parts of the task simultaneously.
- **4 Processes:**
 - **Execution Time:** 427.4054 seconds
 - **Performance:** 0.00234
 - With 4 processes, the execution time continues to decrease. The performance improves further, as more CPU resources are utilized for the parallel execution of the task.
- **6 Processes:**
 - **Execution Time:** 376.8083 seconds
 - **Performance:** 0.00265
 - The performance increases as more processes are introduced. The decrease in execution time shows that the task is becoming more parallelized, which leads to more efficient use of the system's cores.

- **8 Processes:**
 - **Execution Time:** 377.3376 seconds
 - **Performance:** 0.00265
 - There is a slight increase in execution time when moving from 6 to 8 processes, but the performance is roughly the same. This suggests that beyond 6 processes, the overhead of managing processes may counteract any further speedup, potentially due to factors like memory bandwidth or process management overhead.

3. Multithreading Approach:

- **2 Threads:**
 - **Execution Time:** 424.8913 seconds
 - **Performance:** 0.00235
 - With 2 threads, the performance improves significantly compared to the naive approach, but it is slightly worse than using 4 processes. The use of threads allows parallel execution, which speeds up the task. However, using 2 threads is much better than using 2 processes as seen in the table.
- **4 Threads:**
 - **Execution Time:** 278.6612 seconds
 - **Performance:** 0.00359
 - The 4-thread configuration provides a substantial improvement in both execution time and performance. The performance boost here is greater than the multiprocessing approach with 2 or 4 processes, highlighting the efficiency of multithreading for certain tasks.
- **6 Threads:**
 - **Execution Time:** 281.0575 seconds
 - **Performance:** 0.00356
 - Although the performance is still high, the execution time slightly increases when going from 4 to 6 threads. This suggests a diminishing return on performance as the number of threads increases.

- **8 Threads:**
 - **Execution Time:** 263.9043 seconds
 - **Performance:** 0.00379
 - The performance continues to improve with 8 threads, reaching the highest performance value among all configurations. This indicates that, for this task, using more threads maximizes parallelism effectively.

Summary:

- The **Naïve Approach** is the least efficient, with high execution time and low performance.
- The **Multiprocessing Approach** shows a clear improvement in performance as the number of processes increases, but the benefits reach a maximum after 6 and 8 processes due to having 4 physical cores and 8 logical cores.
- The **Multithreading Approach** provides the best performance, with the highest value achieved at 8 threads, demonstrating that multithreading leverages the system's resources more efficiently for this particular task.
- Threads have higher performance than processes (when comparing the same number of threads and processes) and they both (multi-processing and multi-threading) have higher performance than the naïve approach.

The data suggests that using 8 threads provides the optimal balance of speedup and performance, making it the most effective configuration for this task on my system.

3.Optimal Number of Child Processes and Threads

Based on Amdahl's Law and the data from the table, the **optimal number of child processes or threads** is the point where the speedup is maximized while still maintaining an efficient use of available resources (cores).

Amdahl's Law Analysis:

Amdahl's Law states that the speedup of a parallelized program is limited by the fraction of the code that cannot be parallelized (serial part).

Given:

- My laptop has **4 physical cores** and **8 logical cores**, meaning it can run up to 8 threads or processes effectively.

From the table, we can see that the execution time starts to show diminishing returns after a certain number of processes/threads. This is where the **optimal number of threads or processes** lies.

Analysis:

1. For Processes:

- The performance keeps improving until **6 processes**. Going from **6 to 8 processes** shows almost no improvement, indicating a **diminishing return**. This suggests that **6 processes** are close to the optimal point for multiprocessing, as further increase in processes beyond the number of physical cores starts to cause overhead.

2. For Threads:

- The performance keeps improving until **8 threads**, and the execution time decreases significantly, achieving the highest performance at 8 threads (0.00379). Since I have **8 logical cores**, this configuration is the most optimal for the **multithreading approach**.

Summary:

- **Optimal Number of Processes:** **6 processes** appear to be the optimal configuration based on performance, as beyond this point, the benefit of adding more processes diminishes.
- **Optimal Number of Threads:** **8 threads** are the optimal configuration, as it maximizes performance by utilizing all available logical cores efficiently.

This configuration ensures that the program fully utilizes the available resources, providing the best trade-off between execution, time and performance.

Conclusion

This project analyzed the performance of three approaches—naive, multiprocessing, and multithreading—for determining the top 10 most frequent words in a dataset. By leveraging parallel computing concepts, the multiprocessing and multithreading approaches demonstrated significant improvements in execution time compared to the naive sequential implementation.

The analysis using Amdahl's Law revealed that 35% of the program is inherently serial, limiting the achievable speedup. On a 4-core system, the optimal performance was achieved with 4 processes or threads, as it fully utilized the available cores without incurring additional overhead from context switching.

The results highlight the importance of understanding the balance between parallelism, system hardware, and the characteristics of the workload. This knowledge is essential for designing efficient, scalable solutions that maximize resource utilization while minimizing computational overhead.

References

[1]: “Abraham-Silberschatz-Operating-System-Concepts-10th-2018” Course Book.

[Accessed during the semester].

[2]: Amdahl, G. M. (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities." AFIPS Conference Proceedings.

[Accessed December 8, 2024, at 5:28pm].

[3]: Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann.

[Accessed December 8, 2024, at 5:54pm].

Appendix

Please that I attached the codes with the report in case there are some issues while copy pasting the codes.

1.Code Naïve Approach:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>


#define MAX_WORD_LENGTH 60

#define INITIAL_CAPACITY 18000000

#define TOP_K 10

#define GROWTH_FACTOR 2


// Structure to store word and its frequency
typedef struct {

    char *word;

    int frequency;

} WordFreq;


// Structure to manage dynamic array
typedef struct {

    WordFreq *data;

    int size;

    int capacity;

} WordFreqArray;


// Create dynamic word frequency array with initial memory allocation
WordFreqArray* create_word_freq_array() {

    WordFreqArray *arr = malloc(sizeof(WordFreqArray));

    if (!arr) {

        perror("Memory allocation failed");

        exit(1);

    }
```

```

    }

    arr->data = malloc(INITIAL_CAPACITY * sizeof(WordFreq));
    if (!arr->data) {
        perror("Memory allocation failed");
        free(arr);
        exit(1);
    }

    arr->size = 0;
    arr->capacity = INITIAL_CAPACITY;
    return arr;
}

// Resize array when capacity is reached
void resize_word_freq_array(WordFreqArray *arr) {
    int new_capacity = arr->capacity * GROWTH_FACTOR;
    WordFreq *new_data = realloc(arr->data, new_capacity * sizeof(WordFreq));

    if (!new_data) {
        perror("Memory reallocation failed");
        free(arr->data);
        free(arr);
        exit(1);
    }

    arr->data = new_data;
    arr->capacity = new_capacity;
}

// Free dynamically allocated memory
void free_word_freq_array(WordFreqArray *arr) {
    for (int i = 0; i < arr->size; i++) {
        free(arr->data[i].word);
    }
    free(arr->data);
    free(arr);
}

```

```

// Merge two sorted subarrays by frequency
void merge(WordFreq arr[], int left, int mid, int right) {
    int left_size = mid - left + 1;
    int right_size = right - mid;

    // Create temporary arrays for merging
    WordFreq *left_arr = malloc(left_size * sizeof(WordFreq));
    WordFreq *right_arr = malloc(right_size * sizeof(WordFreq));

    if (!left_arr || !right_arr) {
        perror("Merge array allocation failed");
        free(left_arr);
        free(right_arr);
        return;
    }

    // Copy data to temporary arrays
    for (int i = 0; i < left_size; i++) {
        left_arr[i].word = strdup(arr[left + i].word);
        left_arr[i].frequency = arr[left + i].frequency;
    }
    for (int j = 0; j < right_size; j++) {
        right_arr[j].word = strdup(arr[mid + 1 + j].word);
        right_arr[j].frequency = arr[mid + 1 + j].frequency;
    }

    // Merge the temporary arrays
    int i = 0, j = 0, k = left;
    while (i < left_size && j < right_size) {
        if (left_arr[i].frequency >= right_arr[j].frequency) {
            arr[k].word = left_arr[i].word;
            arr[k].frequency = left_arr[i].frequency;
            i++;
        } else {
            arr[k].word = right_arr[j].word;

```

```

        arr[k].frequency = right_arr[j].frequency;
        j++;
    }
    k++;
}

// Copy remaining elements
while (i < left_size) {
    arr[k].word = left_arr[i].word;
    arr[k].frequency = left_arr[i].frequency;
    i++;
    k++;
}

while (j < right_size) {
    arr[k].word = right_arr[j].word;
    arr[k].frequency = right_arr[j].frequency;
    j++;
    k++;
}

// Free temporary arrays
free(left_arr);
free(right_arr);
}

// Recursive merge sort for word frequencies
void merge_sort(WordFreq arr[], int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    merge_sort(arr, left, mid);
    merge_sort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

```



```

// Count frequencies of words in the input array
int count_word_frequencies(char **words, int total_words, WordFreqArray *word_freq) {
    for (int i = 0; i < total_words; i++) {
        int found = 0;

        // Check if word already exists
        for (int j = 0; j < word_freq->size; j++) {
            if (strcmp(word_freq->data[j].word, words[i]) == 0) {
                word_freq->data[j].frequency++;
                found = 1;
                break;
            }
        }

        // Add new word if not found
        if (!found) {
            if (word_freq->size >= word_freq->capacity) {
                resize_word_freq_array(word_freq);
            }

            word_freq->data[word_freq->size].word = strdup(words[i]);
            word_freq->data[word_freq->size].frequency = 1;
            word_freq->size++;
        }
    }

    return 1;
}

// Read words from input file
char** read_words_from_file(char *filename, int *total_words) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return NULL;
    }
}

```

```

// Initialize dynamic array for words
int capacity = INITIAL_CAPACITY;
char **words = malloc(capacity * sizeof(char*));
*total_words = 0;

char buffer[MAX_WORD_LENGTH];

// Read words with dynamic reallocation
while (fscanf(file, "%59s", buffer) == 1) {
    if (*total_words >= capacity) {
        capacity *= GROWTH_FACTOR;
        char **temp = realloc(words, capacity * sizeof(char*));
        if (!temp) {
            perror("Memory reallocation failed");
            // Free previously allocated memory
            for (int i = 0; i < *total_words; i++) {
                free(words[i]);
            }
            free(words);
            fclose(file);
            return NULL;
        }
        words = temp;
    }

    words[*total_words] = strdup(buffer);
    (*total_words)++;
}

fclose(file);
return words;
}

int main() {
    char filename[] = "text8.txt"; //name of cleaned dataset in my laptop

```

```

int total_words = 0;
clock_t start, end;
double execution_time;

// Start timing execution
start = clock();

// Read words from file
char **words = read_words_from_file(filename, &total_words);
if (!words) {
    fprintf(stderr, "Failed to read words from file\n");
    return 1;
}

// Create word frequency array
WordFreqArray *word_freq = create_word_freq_array();

// Count word frequencies
if (!count_word_frequencies(words, total_words, word_freq)) {
    fprintf(stderr, "Failed to count word frequencies\n");
    for (int i = 0; i < total_words; i++) {
        free(words[i]);
    }
    free(words);
    free_word_freq_array(word_freq);
    return 1;
}

// Sort words by frequency
merge_sort(word_freq->data, 0, word_freq->size - 1);

// End timing execution
end = clock();
execution_time = ((double) (end - start)) / CLOCKS_PER_SEC;

// Print top frequent words

```

```

printf("Top 10 Most Frequent Words:\n");
int print_limit = (TOP_K < word_freq->size) ? TOP_K : word_freq->size;
for (int i = 0; i < print_limit; i++) {
    printf("%s: %d\n", word_freq->data[i].word, word_freq->data[i].frequency);
}

// Print statistics
printf("\nTotal Words: %d\n", total_words);
printf("Execution Time: %.4f seconds\n", execution_time);

// Free resources
for (int i = 0; i < total_words; i++) {
    free(words[i]);
}
free(words);
free_word_freq_array(word_freq);

return 0;
}

```

2.Code Multi-processing Approach:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/time.h>

```

```

#define MAX_WORD_LENGTH 60

#define INITIAL_CAPACITY 18000000

#define TOP_K 10

#define GROWTH_FACTOR 2

#define NUM_PROCESSES 8


// Word frequency structure
typedef struct {
    char word[MAX_WORD_LENGTH];
    int frequency;
} WordFreq;


// Dynamic array for word frequencies
typedef struct {
    WordFreq *data;
    int size;
    int capacity;
} WordFreqArray;


// Shared memory structure
typedef struct {
    int size;
    WordFreq data[INITIAL_CAPACITY];
} SharedFreqData;


// Function prototypes
void init_word_freq_array(WordFreqArray *arr);
void add_word_to_freq_array(WordFreqArray *arr, const char *word);
void merge_sort_word_freq(WordFreq *arr, int left, int right);
void merge_word_freq(WordFreq *arr, int left, int mid, int right);
char** read_words_from_file(const char *filename, int *total_words);


// Initialize word frequency array
void init_word_freq_array(WordFreqArray *arr) {
    arr->data = malloc(INITIAL_CAPACITY * sizeof(WordFreq));

```

```

    if (!arr->data) {
        perror("Memory allocation failed");
        exit(1);
    }

    arr->size = 0;
    arr->capacity = INITIAL_CAPACITY;
}

// Add word to frequency array with dynamic resizing
void add_word_to_freq_array(WordFreqArray *arr, const char *word) {
    // Check if word already exists
    for (int i = 0; i < arr->size; i++) {
        if (strcmp(arr->data[i].word, word) == 0) {
            arr->data[i].frequency++;
            return;
        }
    }

    // Resize array if needed
    if (arr->size >= arr->capacity) {
        arr->capacity *= GROWTH_FACTOR;
        arr->data = realloc(arr->data, arr->capacity * sizeof(WordFreq));
        if (!arr->data) {
            perror("Memory reallocation failed");
            exit(1);
        }
    }

    // Add new word
    strncpy(arr->data[arr->size].word, word, MAX_WORD_LENGTH - 1);
    arr->data[arr->size].word[MAX_WORD_LENGTH - 1] = '\0';
    arr->data[arr->size].frequency = 1;
    arr->size++;
}

// Merge subarrays during sorting

```

```

void merge_word_freq(WordFreq *arr, int left, int mid, int right) {
    int left_size = mid - left + 1;
    int right_size = right - mid;

    // Temporary arrays
    WordFreq *left_arr = malloc(left_size * sizeof(WordFreq));
    WordFreq *right_arr = malloc(right_size * sizeof(WordFreq));

    if (!left_arr || !right_arr) {
        perror("Merge allocation failed");
        free(left_arr);
        free(right_arr);
        return;
    }

    // Copy data to temporary arrays
    memcpy(left_arr, &arr[left], left_size * sizeof(WordFreq));
    memcpy(right_arr, &arr[mid + 1], right_size * sizeof(WordFreq));

    // Merge back
    int i = 0, j = 0, k = left;
    while (i < left_size && j < right_size) {
        if (left_arr[i].frequency >= right_arr[j].frequency) {
            arr[k] = left_arr[i];
            i++;
        } else {
            arr[k] = right_arr[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements
    while (i < left_size) {
        arr[k] = left_arr[i];
        i++;
    }

```

```

        k++;
    }

    while (j < right_size) {
        arr[k] = right_arr[j];
        j++;
        k++;
    }

    // Free temporary arrays
    free(left_arr);
    free(right_arr);
}

// Recursive merge sort for word frequencies
void merge_sort_word_freq(WordFreq *arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    merge_sort_word_freq(arr, left, mid);
    merge_sort_word_freq(arr, mid + 1, right);
    merge_word_freq(arr, left, mid, right);
}

// Read words from input file with dynamic memory allocation
char** read_words_from_file(const char *filename, int *total_words) {
    FILE *file = fopen(filename, "r");

    if (!file) {
        perror("Error opening file");
        return NULL;
    }

    // Initial allocation
    int capacity = INITIAL_CAPACITY;
    char **words = malloc(capacity * sizeof(char*));
    *total_words = 0;

```



```

char buffer[MAX_WORD_LENGTH];

// Read words with dynamic reallocation
while (fscanf(file, "%59s", buffer) == 1) {
    if (*total_words >= capacity) {
        capacity *= GROWTH_FACTOR;
        char **temp = realloc(words, capacity * sizeof(char*));
        if (!temp) {
            perror("Memory reallocation failed");
            // Free previously allocated memory
            for (int i = 0; i < *total_words; i++) {
                free(words[i]);
            }
            free(words);
            fclose(file);
            return NULL;
        }
        words = temp;
    }

    words[*total_words] = strdup(buffer);
    (*total_words)++;
}

fclose(file);
return words;
}

int main() {
    // Start timing
    struct timeval start, end;
    gettimeofday(&start, NULL);

    const char *filename = "text8.txt";
    int total_words = 0;

```

```

// Read words from file
char **words = read_words_from_file(filename, &total_words);
if (!words) {
    fprintf(stderr, "Failed to read words from file\n");
    return 1;
}

// Create shared memory for word frequencies
SharedFreqData *shared_data = mmap(NULL, sizeof(SharedFreqData),
    PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS,
    -1, 0);

if (shared_data == MAP_FAILED) {
    perror("mmap failed");
    for (int i = 0; i < total_words; i++) {
        free(words[i]);
    }
    free(words);
    return 1;
}

// Fork child processes with optimized chunk distribution
pid_t pids[NUM_PROCESSES];
int chunk_size = total_words / NUM_PROCESSES;
int remainder = total_words % NUM_PROCESSES;

for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();

    if (pids[i] == -1) {
        perror("fork failed");
        munmap(shared_data, sizeof(SharedFreqData));
        for (int j = 0; j < total_words; j++) {
            free(words[j]);
        }
    }
}

```

```

    }

    free(words);

    exit(1);
} else if (pids[i] == 0) {

    // Child process

    int start = i * chunk_size + (i < remainder ? i : remainder);

    int end = start + chunk_size + (i < remainder ? 1 : 0);

    // Local word frequency array
    WordFreqArray local_freq;
    init_word_freq_array(&local_freq);

    // Count frequencies for this subset of words
    for (int j = start; j < end; j++) {
        add_word_to_freq_array(&local_freq, words[j]);
    }

    // Transfer to shared memory with atomic-like synchronization
    for (int j = 0; j < local_freq.size; j++) {
        int found = 0;
        for (int k = 0; k < shared_data->size; k++) {
            if (strcmp(shared_data->data[k].word, local_freq.data[j].word) == 0) {
                shared_data->data[k].frequency += local_freq.data[j].frequency;
                found = 1;
                break;
            }
        }

        // Add new word if not found
        if (!found && shared_data->size < INITIAL_CAPACITY) {
            strcpy(shared_data->data[shared_data->size].word, local_freq.data[j].word);
            shared_data->data[shared_data->size].frequency = local_freq.data[j].frequency;
            shared_data->size++;
        }
    }
}

```

```

        // Free local resources

        free(local_freq.data);

        exit(0);
    }
}

// Parent process waits for children
for (int i = 0; i < NUM_PROCESSES; i++) {
    int status;

    waitpid(pids[i], &status, 0);

    // Check if child process terminated normally
    if (!WIFEXITED(status)) {
        fprintf(stderr, "Child process %d did not terminate normally\n", pids[i]);
    }
}

// Sort words by frequency
merge_sort_word_freq(shared_data->data, 0, shared_data->size - 1);

// End timing calculation
gettimeofday(&end, NULL);
double execution_time = (end.tv_sec - start.tv_sec) +
    (end.tv_usec - start.tv_usec) / 1000000.0;

// Print top 10 most frequent words
printf("Top 10 Most Frequent Words:\n");
int print_limit = (TOP_K < shared_data->size) ? TOP_K : shared_data->size;
for (int i = 0; i < print_limit; i++) {
    printf("%s: %d\n", shared_data->data[i].word, shared_data->data[i].frequency);
}

// Print statistics
printf("\nTotal Words: %d\n", total_words);
printf("Number of Processes Used: %d\n", NUM_PROCESSES);
printf("Execution Time: %.4f seconds\n", execution_time);

```

```

// Free resources

for (int i = 0; i < total_words; i++) {
    free(words[i]);
}

free(words);

munmap(shared_data, sizeof(SharedFreqData));

return 0;
}

```

3.Code Multi-threading Approach:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX_WORD_LENGTH 60
#define INITIAL_CAPACITY 18000000
#define TOP_K 10
#define GROWTH_FACTOR 2
#define NUM_THREADS 8

// Mutex for thread-safe operations
pthread_mutex_t frequency_mutex = PTHREAD_MUTEX_INITIALIZER;

// Structure to store word and its frequency
typedef struct {
    char *word;
    int frequency;
} WordFreq;

// Structure to manage dynamic array
typedef struct {
    WordFreq *data;
    int size;
    int capacity;
} WordFreqArray;

// Thread argument structure
typedef struct {
    char **words;
    int start;

```

```

    int end;
    WordFreqArray *shared_word_freq;
} ThreadArgs;

// Create dynamic word frequency array with initial memory allocation
WordFreqArray* create_word_freq_array() {
    WordFreqArray *arr = malloc(sizeof(WordFreqArray));
    if (!arr) {
        perror("Memory allocation failed");
        exit(1);
    }
    arr->data = malloc(INITIAL_CAPACITY * sizeof(WordFreq));
    if (!arr->data) {
        perror("Memory allocation failed");
        free(arr);
        exit(1);
    }
    arr->size = 0;
    arr->capacity = INITIAL_CAPACITY;
    return arr;
}

// Resize array when capacity is reached
void resize_word_freq_array(WordFreqArray *arr) {
    int new_capacity = arr->capacity * GROWTH_FACTOR;
    WordFreq *new_data = realloc(arr->data, new_capacity * sizeof(WordFreq));

    if (!new_data) {
        perror("Memory reallocation failed");
        free(arr->data);
        free(arr);
        exit(1);
    }

    arr->data = new_data;
    arr->capacity = new_capacity;
}

// Free dynamically allocated memory
void free_word_freq_array(WordFreqArray *arr) {
    for (int i = 0; i < arr->size; i++) {
        free(arr->data[i].word);
    }
    free(arr->data);
    free(arr);
}

// Merge two sorted subarrays by frequency
void merge(WordFreq arr[], int left, int mid, int right) {
    int left_size = mid - left + 1;
    int right_size = right - mid;

    // Create temporary arrays for merging
    WordFreq *left_arr = malloc(left_size * sizeof(WordFreq));
    WordFreq *right_arr = malloc(right_size * sizeof(WordFreq));

    if (!left_arr || !right_arr) {
        perror("Merge array allocation failed");
        free(left_arr);
        free(right_arr);
        return;
    }

    // Copy data to temporary arrays
    for (int i = 0; i < left_size; i++) {

```

```

    left_arr[i].word = strdup(arr[left + i].word);
    left_arr[i].frequency = arr[left + i].frequency;
}
for (int j = 0; j < right_size; j++) {
    right_arr[j].word = strdup(arr[mid + 1 + j].word);
    right_arr[j].frequency = arr[mid + 1 + j].frequency;
}

// Merge the temporary arrays
int i = 0, j = 0, k = left;
while (i < left_size && j < right_size) {
    if (left_arr[i].frequency >= right_arr[j].frequency) {
        arr[k].word = left_arr[i].word;
        arr[k].frequency = left_arr[i].frequency;
        i++;
    } else {
        arr[k].word = right_arr[j].word;
        arr[k].frequency = right_arr[j].frequency;
        j++;
    }
    k++;
}

// Copy remaining elements
while (i < left_size) {
    arr[k].word = left_arr[i].word;
    arr[k].frequency = left_arr[i].frequency;
    i++;
    k++;
}

while (j < right_size) {
    arr[k].word = right_arr[j].word;
    arr[k].frequency = right_arr[j].frequency;
    j++;
    k++;
}

// Free temporary arrays
free(left_arr);
free(right_arr);
}

// Recursive merge sort for word frequencies
void merge_sort(WordFreq arr[], int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    merge_sort(arr, left, mid);
    merge_sort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Thread function to process word frequencies
void* process_word_chunk(void *arg) {
    ThreadArgs *thread_args = (ThreadArgs*)arg;

    // Local frequency array for thread
    WordFreqArray local_word_freq;
    local_word_freq.data = malloc(INITIAL_CAPACITY * sizeof(WordFreq));
    local_word_freq.size = 0;
    local_word_freq.capacity = INITIAL_CAPACITY;

    // Process words in assigned chunk
    for (int i = thread_args->start; i < thread_args->end; i++) {

```

```

int found = 0;

// Check local frequencies first
for (int j = 0; j < local_word_freq.size; j++) {
    if (strcmp(local_word_freq.data[j].word, thread_args->words[i]) == 0) {
        local_word_freq.data[j].frequency++;
        found = 1;
        break;
    }
}

// Add new word to local array if not found
if (!found) {
    if (local_word_freq.size >= local_word_freq.capacity) {
        local_word_freq.capacity *= GROWTH_FACTOR;
        local_word_freq.data = realloc(local_word_freq.data,
                                       local_word_freq.capacity * sizeof(WordFreq));
    }

    local_word_freq.data[local_word_freq.size].word = strdup(thread_args->words[i]);
    local_word_freq.data[local_word_freq.size].frequency = 1;
    local_word_freq.size++;
}
}

// Merge local results with shared array
pthread_mutex_lock(&frequency_mutex);
for (int i = 0; i < local_word_freq.size; i++) {
    int found = 0;
    for (int j = 0; j < thread_args->shared_word_freq->size; j++) {
        if (strcmp(thread_args->shared_word_freq->data[j].word,
                    local_word_freq.data[i].word) == 0) {
            thread_args->shared_word_freq->data[j].frequency +=
                local_word_freq.data[i].frequency;
            found = 1;
            break;
        }
    }

    // Add new word to shared array if not found
    if (!found) {
        if (thread_args->shared_word_freq->size >=
            thread_args->shared_word_freq->capacity) {
            resize_word_freq_array(thread_args->shared_word_freq);
        }

        thread_args->shared_word_freq->data[thread_args->shared_word_freq->size].word =
            strdup(local_word_freq.data[i].word);
        thread_args->shared_word_freq->data[thread_args->shared_word_freq->size].frequency =
            local_word_freq.data[i].frequency;
        thread_args->shared_word_freq->size++;
    }
}
pthread_mutex_unlock(&frequency_mutex);

// Free local resources
for (int i = 0; i < local_word_freq.size; i++) {
    free(local_word_freq.data[i].word);
}
free(local_word_freq.data);

return NULL;
}

// Read words from input file

```



```

char** read_words_from_file(char *filename, int *total_words) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return NULL;
    }

    // Initialize dynamic array for words
    int capacity = INITIAL_CAPACITY;
    char **words = malloc(capacity * sizeof(char*));
    *total_words = 0;

    char buffer[MAX_WORD_LENGTH];

    // Read words with dynamic reallocation
    while (fscanf(file, "%59s", buffer) == 1) {
        if (*total_words >= capacity) {
            capacity *= GROWTH_FACTOR;
            char **temp = realloc(words, capacity * sizeof(char*));
            if (!temp) {
                perror("Memory reallocation failed");
                // Free previously allocated memory
                for (int i = 0; i < *total_words; i++) {
                    free(words[i]);
                }
                free(words);
                fclose(file);
                return NULL;
            }
            words = temp;
        }

        words[*total_words] = strdup(buffer);
        (*total_words)++;
    }

    fclose(file);
    return words;
}

int main() {
    char filename[] = "text8.txt"; // name of input file
    int total_words = 0;

    // Time tracking structures
    struct timeval start, end;
    double execution_time;

    // Start timing execution
    gettimeofday(&start, NULL);

    // Read words from file
    char **words = read_words_from_file(filename, &total_words);
    if (!words) {
        fprintf(stderr, "Failed to read words from file\n");
        return 1;
    }

    // Create shared word frequency array
    WordFreqArray *word_freq = create_word_freq_array();

    // Create thread handles
    pthread_t threads[NUM_THREADS];
    ThreadArgs thread_args[NUM_THREADS];

```

```

// Distribute work among threads
int chunk_size = total_words / NUM_THREADS;
int remainder = total_words % NUM_THREADS;

// Create threads
for (int i = 0; i < NUM_THREADS; i++) {
    // Calculate start and end for each thread
    int start = i * chunk_size + (i < remainder ? i : remainder);
    int end = start + chunk_size + (i < remainder ? 1 : 0);

    // Prepare thread arguments
    thread_args[i].words = words;
    thread_args[i].start = start;
    thread_args[i].end = end;
    thread_args[i].shared_word_freq = word_freq;

    // Create thread
    if (pthread_create(&threads[i], NULL, process_word_chunk, &thread_args[i]) != 0) {
        perror("Thread creation failed");
        // Cleanup resources
        for (int j = 0; j < total_words; j++) {
            free(words[j]);
        }
        free(words);
        free_word_freq_array(word_freq);
        return 1;
    }
}

// Wait for all threads to complete
for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
}

// Sort words by frequency
merge_sort(word_freq->data, 0, word_freq->size - 1);

// End timing execution
gettimeofday(&end, NULL);
execution_time = (end.tv_sec - start.tv_sec) +
    (end.tv_usec - start.tv_usec) / 1000000.0;

// Print top frequent words
printf("Top 10 Most Frequent Words:\n");
int print_limit = (TOP_K < word_freq->size) ? TOP_K : word_freq->size;
for (int i = 0; i < print_limit; i++) {
    printf("%s: %d\n", word_freq->data[i].word, word_freq->data[i].frequency);
}

// Print statistics
printf("\nTotal Words: %d\n", total_words);
printf("Number of Threads Used: %d\n", NUM_THREADS);
printf("Execution Time: %.4f seconds\n", execution_time);

// Free resources
for (int i = 0; i < total_words; i++) {
    free(words[i]);
}
free(words);
free_word_freq_array(word_freq);

return 0;
}

```

