



Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Operating Systems – ENCS 3390

Report

Prepared by:

Name: Amir Al-Rashayda

Number: 1222596

Instructor: Dr.Mohammed Khalil

Section: 2

Date: 11/25/2024

Abstract	3
Theory “Theoretical without numbers”	4
Procedure “Calculation & Discussion”	13
1. Naive Approach	13
1. Environment Description	13
2. Naive Approach Execution Overview	14
3. Performance Analysis	14
4. Amdahl’s Law Analysis	15
5. Top 10 Most Frequent Words	15
6. Conclusion	16
2. Multiprocessing Approach	17
1. Environment Description	19
2. Performance Analysis	19
3. Multiprocessing Implementation Methodology	19
4. Speedup and Efficiency Calculations	21
4. Discussion	25
3. Multithreading Approach	26
1. Performance Analysis & Implementation Methodology	28
2. Threads Calculation:	29
3. Discussion	32
Performance Comparison of Parallel table	33
Conclusion	34

Abstract

Three methods for determining the top ten most frequent terms in the enwik8 dataset are evaluated in this report: naïve, multiprocessing, and multithreading. The execution time of each method is monitored, and 2, 4, 6, and 8 parallel tasks are used to test multiprocessing and multithreading. Using Amdahl's rule to estimate speedup and identify ideal configurations, the study evaluates scalability, efficiency, and the effect of parallelization. The trade-offs and optimal strategies for utilizing parallel computing are highlighted by the results, which provide notable performance improvements with parallel approaches when compared to the naive technique.

Theory “Theoretical without numbers”

This project explores three approaches for processing large datasets to extract the top 10 most frequent words: naive, multiprocessing, and multithreading. Each approach demonstrates unique strengths and trade-offs in terms of simplicity, resource utilization, and execution time. The following sections provide a concise explanation of each method.

1. Naive Approach

Without using threading, parallelism, or multiprocessing, the Naive Approach completes the operation in a sequential manner. It may be slower for computationally demanding operations, but it is intended to be clear and simple.

A. Overview

This method processes the dataset linearly, completing each task using a single thread. Its simplicity makes it easy to understand and debug, and it avoids concurrency issues such as race conditions. However, it is inefficient for large datasets, as it relies on a single core and cannot leverage multi-core systems.

B. Implementation in Code

The implementation processes the dataset in a single thread. Data structures that hold a word and its frequency, like WordFreq, are used. The program's execution is streamlined by constants like MAX_WORD_LENGTH, which specifies the maximum size of a word, and INITIAL_ARRAY_SIZE, which allots memory for the words.

The file is read sequentially using fscanf, processing one word at a time. For each word, the program searches the existing list of words. If the word is found, its frequency is incremented. Otherwise, it is added to the list. To handle

the growing size of the word list, memory is managed dynamically using malloc for initial allocation and realloc when the list needs to expand. This ensures efficient use of memory while accommodating large numbers of unique words.

Sorting is performed using a heap sort algorithm. This method is efficient for sorting the list of unique words by their frequency, with a time complexity of $O(n\log(n))$. After sorting, the program prints the top 10 most frequent words along with their frequencies.

Throughout the file processing phase, progress reports are given. Every 60 seconds, the program outputs the total number of words processed and the unique words discovered. Users are guaranteed to receive real-time feedback on how the program is functioning thanks to this feature.

C. Key Functions

The program's core function is `process_file`. It manages reading files, creates the word list dynamically, and records progress. If a word is not discovered in the list, the `find_or_add_word` function either adds it or increases its frequency. The `heap_sort` function makes sure that the most frequently occurring words are readily available for output by sorting the words by frequency in decreasing order.

Another important aspect of the implementation is the memory management. By using dynamic allocation, the program efficiently handles the dataset, avoiding unnecessary memory usage while ensuring the program can scale to accommodate large numbers of unique words. Memory deallocation is also implemented to prevent memory leaks.

D. Time Complexity

There are two main aspects that affect this approach's temporal complexity. The difficulty of processing each word is $O(n \cdot m)$, where n is the total number of words in the dataset and m is the average word list size. The complexity of sorting the unique words is $O(k \log k)$, where k is the number of unique words. Combined, the temporal complexity is $O(n \cdot m + k \log k) = (n \cdot m + k \log k)$.

E. Strengths and Limitations

This method is perfect for small datasets or as a starting point for performance comparisons because it is simple to implement. However, processing big datasets like enwik8 is inefficient because to its sequential nature. A single core's performance limits the software, which might cause major lags when handling massive amounts of data.

F. Real-World Relevance

In real-world applications, the Naive Approach works best in situations where the dataset size is controllable or if debugging simplicity and ease are top concerns. It also acts as a starting point for assessing how effective more sophisticated strategies, such as multiprocessing or multithreading, are in utilizing the capabilities of multi-core systems to drastically cut down on processing time.

2. Multiprocessing Approach

A. Overview

By using several processes that can operate independently on several CPU cores or logical processors, multiprocessing is a technique for achieving parallelism. In contrast to multithreading, multiprocessing allows for genuine parallel execution by avoiding restrictions such as the Global Interpreter Lock (GIL) in some programming languages. Because each process has its own

memory space, this method works especially well for computationally demanding tasks.

Utilizing several CPU cores is one of the primary benefits of multiprocessing, as it **greatly enhances performance for activities that can be completed in parallel**. Furthermore, the risk of memory corruption **is decreased by each process allocating memory independently**. But there are drawbacks to this strategy as well, such as increased overhead for setting up and maintaining processes and slower inter-process communication because there is no shared memory.

B. Implementation Details

Depending on the programming language selected, different APIs or libraries can be used to achieve the multiprocessing technique. Usually, a series of crucial procedures are followed throughout installation.

Data splitting is the initial phase, in which the dataset is separated into smaller pieces, one for each procedure to handle. This guarantees that each process receives an equal share of the workload. The program divides the data and then uses APIs like `fork()` in C. Every process carries out a particular task that separately processes the data it has been given.

The outcomes are synchronized after the processes have finished their work. Using shared memory, files, or inter-process communication protocols, the outputs from each process are combined to accomplish this. The top ten most often occurring words in the dataset are then determined by analyzing the results.

Lastly, to evaluate the performance boost brought about by multiprocessing, the execution time is assessed. This is accomplished by using timing functions, such as `clock_gettime()` in C or comparable tools in other languages, to record the program's start and end times.

The method uses the `fork()` system call to generate child processes in the given C code. The dataset is split up by the parent process and processed by the child processes. To determine the most common words, the parent process aggregates the findings that each child writes to a shared resource, like a file or pipe.

C. Analyzing Multiprocessing Performance

The performance of the multiprocessing approach is analyzed by measuring execution time and evaluating speedup using Amdahl's Law. Execution time is measured by recording the duration taken to process the dataset using different numbers of processes, such as 2, 4, 6, and 8. This helps in determining the optimal number of processes for the given system.

Amdahl's Law provides a theoretical framework to estimate the speedup achieved through parallelization. It states that the overall speedup of a program depends on the proportion of the program that can be parallelized and the number of processors available. For instance, if 80% of the code can be parallelized and 8 processes are used, the speedup is calculated as:

$$S = 1 / ((1-P) + P/N)$$

In this case, the speedup would be approximately **4.44 times**. This demonstrates that as the number of processes increases, the speedup approaches a maximum limit, constrained by the serial portion of the code. For optimal performance, the number of processes should match the number of CPU cores to minimize overhead and contention.

D. Result

By using multiple CPU cores to process the dataset in parallel, the multiprocessing strategy outperforms the naive approach by a considerable margin. Because multiprocessing achieves genuine parallelism without being constrained by the GIL, it frequently outperforms the multithreading technique for CPU-bound activities.

The amount of processes employed affects the performance that is seen. Although it underutilizes the CPU, using two processes offers a moderate improvement. On a 4-core computer, four processes often result in best performance, balancing overhead and workload distribution. Because of the additional expense of maintaining more processes and the possibility of CPU contention, **using six or eight processes can result in declining results.**

E. Conclusion

For parallel computing, the multiprocessing strategy works quite well, particularly for computationally demanding applications like identifying the most common phrases in a collection. When compared to naive and multithreaded techniques, it offers notable speed benefits and makes efficient use of the available CPU cores. However, the balance between the number of processes and the available system resources needs to be carefully considered in order to ensure optimal performance. This method shows how multiprocessing can optimize computing efficiency for jobs that can be efficiently parallelized.

3. Multithreading Approach

A. Overview

Programming techniques known as multithreading enable several threads of execution to operate simultaneously inside a single process. Since threads share memory, communication between them is quicker and more effective than with multiprocessing. Because of this, multithreading is especially helpful for jobs requiring a large volume of shared data or I/O activities.

Because threads use less system overhead than processes, multithreading's lightweight nature is one of its primary benefits. Threads can also share resources, which facilitates quicker synchronization and communication. But there are drawbacks to this shared memory architecture as well, like the possibility of race situations, deadlocks, and the requirement for appropriate synchronization techniques. The Global Interpreter Lock (GIL) in Python, which inhibits real parallelism in some programming environments, can limit multithreading for CPU-bound tasks.

B. Implementation Details

To accomplish parallel execution, the multithreading technique adheres to a systematic procedure. Similar to multiprocessing, it starts by breaking the dataset up into smaller pieces. However, threads are launched within the same process rather than forming separate processes. Every thread is given a distinct piece of data to handle on its own.

Threading libraries or APIs, such as pthread in C, are used to construct threads. Each thread handles its share of the dataset by executing a function after it has been produced. Threads share memory, unlike multiprocessing, which makes data sharing easier but necessitates synchronization techniques to avoid conflicts. To guarantee thread-safe actions while accessing shared resources, mutexes, locks, or semaphores are frequently utilized.

Once all threads complete their work, the results are aggregated to identify the most frequent words in the dataset. Execution timing is also recorded to measure the performance improvement achieved with multithreading. This is typically done by noting the start and end times using timing utilities like `clock_gettime()` in C or similar functions in other languages.

The `pthread_create` function is used to create threads in the C code. The parent thread splits the dataset into segments, creates threads to process each segment, and waits for each thread to finish using `pthread_join`. The most common terms are then identified by combining the outcomes from every thread.

C. Analyzing Multithreading Performance

The execution time for varying numbers of threads, such as 2, 4, 6, and 8, is measured in order to assess the multithreading approach's performance. This makes it possible to examine the effects of adding more threads on performance.

Using Amdahl's Law, the theoretical speedup is calculated based on the proportion of the code that can be parallelized and the number of threads used. For instance, if 80% of the program is parallelizable and 8 threads are used, the speedup is determined as:

$$S = 1 / ((1-P) + P/N)$$

In this example, the speedup would be approximately 4.44 times, similar to the multiprocessing approach. However, the actual speedup may be lower due to overhead from thread management and synchronization.

For CPU-bound operations, the ideal number of threads often corresponds to the number of CPU cores. Because of context switching and conflict, adding more threads than this limit may lead to diminishing returns or even performance loss.

D. Result

The multithreading approach offers significant improvements over the naive approach by allowing multiple threads to process the dataset concurrently. However, compared to multiprocessing, its performance may be limited in CPU-bound tasks due to shared memory constraints and the lack of true parallelism in some programming environments.

The performance observed depends on the number of threads used. With two threads, the improvement is modest as the CPU is underutilized. Four threads usually provide optimal performance on a 4-core machine. Using six or eight threads may lead to diminishing returns because of increased contention for CPU resources and higher synchronization overhead.

E. Conclusion

Multithreading is a powerful tool for parallel computing, particularly for tasks that involve shared memory or I/O operations. It enables efficient communication and synchronization between threads **while consuming less system overhead compared to processes**. However, its effectiveness for CPU-bound tasks can be limited by shared memory contention and synchronization overhead. Proper thread management and synchronization are crucial for achieving optimal performance. By leveraging multithreading, substantial performance gains can be achieved, making it a viable approach for tasks that can be parallelized effectively.

Procedure “Calculation & Discussion”

1. Naive Approach

```
● amir_alrashayda@DESKTOP-L6AF422:~/VSC$ gcc -o Naive_Approach Naive_Approach.c
● amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Naive_Approach
Starting word frequency analysis...
Progress: Processed 3799784 words, 108906 unique words found (1.00 minutes elapsed)
Progress: Processed 6827023 words, 149843 unique words found (2.00 minutes elapsed)
Progress: Processed 9565748 words, 184674 unique words found (3.00 minutes elapsed)
Progress: Processed 12319530 words, 211030 unique words found (4.00 minutes elapsed)
Progress: Processed 14838967 words, 235450 unique words found (5.00 minutes elapsed)

File reading completed. Total words: 17005208, Unique words: 253854

Sorting words by frequency...

Top 10 most frequent words:
Word                Frequency
-----
the                  1061396
of                   593677
and                  416629
one                  411764
in                   372201
a                    325873
to                   316376
zero                 264975
nine                 250430
two                  192644

Total execution time: 353.00 seconds
```

1. Environment Description

- Computer Specifications:
 - Processor: 12th Gen Intel(R) Core(TM) i5-12600KF
 - Base Clock Speed: 3.69 GHz
 - Architecture: **Hybrid (Performance and Efficiency cores) “This is important for our approaches (6 P-cores × 2 Hyper-Threading)”**
 - RAM: 32.0 GB (31.8 GB usable)
- Operating System: Ubuntu 20.04 LTS “WSL Not with full interface”
- Programming Language: C
- IDE Tool: Visual Studio Code

- Virtual Machine: No virtual machine was used; the program was executed directly on the host operating system.

2. Naive Approach Execution Overview

The Naive Approach was implemented to analyze the frequency of words in the enwik8 dataset. The program processed a total of 17,005,208 words and identified 253,854 unique words. The execution time for this approach was 353 seconds.

- Progress Tracking: The program provided progress updates at one-minute intervals, showing the number of words processed and unique words found:
 - 1 minute: 3,799,784 words processed, 108,906 unique words found.
 - 2 minutes: 6,827,023 words processed, 149,843 unique words found.
 - 3 minutes: 9,565,748 words processed, 184,674 unique words found.
 - 4 minutes: 12,319,530 words processed, 211,030 unique words found.
 - 5 minutes: 14,838,967 words processed, 235,450 unique words found.

This incremental progress indicates that the program is effectively reading and processing the data, although the rate of unique word discovery slows down as the total word count increases.

3. Performance Analysis

- The naive approach is characterized by its simplicity, as it does not utilize any child processes or threads. This results in a longer execution time compared to potential multiprocessing or multithreading approaches.
- The total execution time of 353 seconds indicates that the program is likely limited by the single-threaded nature of its design, which can be a bottleneck when processing large datasets.

4. Amdahl's Law Analysis

- Serial Part of the Code: The entire word counting and frequency calculation is done in a single thread, making it entirely serial. Thus, the percentage of the serial part of the code is **100%**.
- Maximum Speedup: Since the code is entirely serial, the maximum speedup is limited to 1 (no speedup).
- Optimal Number of Processes/Threads: Given that the naive approach does not utilize parallel processing, the optimal number of processes or threads will be determined in the subsequent implementations. However, based on Amdahl's Law, the more parallelizable parts of the code, the better the performance with an increased number of processes or threads.

5. Top 10 Most Frequent Words

The output lists the top 10 most frequent words along with their frequencies:

- the: 1,061,396 occurrences
- of: 593,677 occurrences
- and: 416,629 occurrences
- one: 411,764 occurrences
- in: 372,201 occurrences
- a: 325,873 occurrences
- to: 316,376 occurrences
- zero: 264,975 occurrences
- nine: 250,430 occurrences
- two: 192,644 occurrences

This list reflects common English words, which are expected to appear frequently in a large corpus of text.

6. Conclusion

The Naive Approach provides a foundational understanding of the word frequency analysis process. While it successfully completes the task, the execution time highlights the need for optimization through parallel processing techniques. The next steps will involve implementing multiprocessing and multithreading approaches to compare their performance against this baseline, ultimately aiming to achieve faster execution times and better resource utilization. The results from these implementations will provide insights into the effectiveness of parallel processing in handling large datasets.

2. Multiprocessing Approach

For 2 processes

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ gcc -o Multiprocessing_Approach Multiprocessing_Approach.c
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multiprocessing_Approach 2
Starting word frequency analysis with 2 processes...
Process 0: Processed 3580108 words, 105560 unique words
Process 1: Processed 3470662 words, 103316 unique words
Process 0: Processed 6381680 words, 144217 unique words
Process 1: Processed 6189589 words, 143590 unique words
Process 0 completed: 8485298 words processed, 171140 unique words found
Process 1 completed: 8519911 words processed, 172507 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word          Frequency
-----
the            1061396
of              593677
and            416629
one            411764
in             372201
a              325873
to             316376
zero          264975
nine          250430
two           192644

Total execution time: 276.00 seconds
```

For 4 processes

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multiprocessing_Approach 4
Starting word frequency analysis with 4 processes...
Process 1: Processed 3270463 words, 98987 unique words
Process 0: Processed 3243174 words, 100195 unique words
Process 3: Processed 2838254 words, 94817 unique words
Process 2: Processed 3201478 words, 99393 unique words
Process 0 completed: 4257281 words processed, 115004 unique words found
Process 1 completed: 4228018 words processed, 116008 unique words found
Process 2 completed: 4247882 words processed, 114725 unique words found
Process 3 completed: 4272030 words processed, 118448 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word          Frequency
-----
the            1061396
of              593677
and            416629
one            411764
in             372201
a              325873
to             316376
zero          264975
nine          250430
two           192644

Total execution time: 213.00 seconds
```

For 6 processes

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multiprocessing_Approach 6
Starting word frequency analysis with 6 processes...
Process 1 completed: 2818939 words processed, 88933 unique words found
Process 4 completed: 2846661 words processed, 92162 unique words found
Process 3 completed: 2837645 words processed, 93081 unique words found
Process 5 completed: 2835606 words processed, 93343 unique words found
Process 2 completed: 2825900 words processed, 92579 unique words found
Process 0 completed: 2840460 words processed, 93871 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word          Frequency
-----
the            1061396
of             593677
and            416629
one            411764
in             372201
a              325873
to             316376
zero          264975
nine          250430
two           192644

Total execution time: 186.00 seconds
```

For 8 processes

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multiprocessing_Approach 8
Starting word frequency analysis with 8 processes...
Process 5 completed: 2121767 words processed, 74521 unique words found
Process 1 completed: 2126018 words processed, 74050 unique words found
Process 2 completed: 2108426 words processed, 76810 unique words found
Process 3 completed: 2119592 words processed, 79936 unique words found
Process 0 completed: 2131263 words processed, 80848 unique words found
Process 4 completed: 2126114 words processed, 80509 unique words found
Process 7 completed: 2127384 words processed, 78918 unique words found
Process 6 completed: 2144647 words processed, 81046 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word          Frequency
-----
the            1061396
of             593677
and            416629
one            411764
in             372201
a              325873
to             316376
zero          264975
nine          250430
two           192644

Total execution time: 200.00 seconds
```

1. Environment Description

- Processor: 12th Gen Intel(R) Core(TM) i5-12600KF
- **10 cores (6 P-cores + 4 E-cores):**
 - Base Clock Speed: 3.69 GHz
 - Architecture: Hybrid (Performance and Efficiency cores)
- RAM: 32.0 GB (31.8 GB usable)
- Operating System: Ubuntu 20.04 LTS “**WSL Not with full interface**”
- Programming Language: C
- IDE Tool: Visual Studio Code
- Virtual Machine: No virtual machine was used; the program was executed directly on the host operating system.

2. Performance Analysis

The multiprocessing approach fundamentally differs from the naive method by implementing parallel processing through multiple child processes. This strategy allows simultaneous processing of different file chunks, significantly reducing the overall execution time. The implementation utilized shared memory for inter-process communication, enabling efficient data exchange and synchronization.

3. Multiprocessing Implementation Methodology

The multiprocessing approach in this implementation was achieved using POSIX process management and inter-process communication APIs from the standard C libraries. Specifically, the following key APIs and functions were utilized:

Process Creation and Management

- `fork()`: The primary function for creating child processes
- `pid_t`: Process ID type for tracking and managing processes
- `wait()` and `waitpid()`: Functions for synchronizing and managing child process completion

Inter-Process Communication Mechanisms

1. Shared Memory:
 - Used `mmap()` for creating shared memory segments
 - `<sys/mman.h>` library for memory mapping
 - Allowed efficient data sharing between processes
2. Temporary File-based Communication:
 - Each process writes its results to a unique temporary file
 - Parent process aggregates results by reading these files
 - Provides a robust method of result collection

Synchronization and Communication

- Implemented file-based synchronization
- Used atomic file write operations
- Minimal use of locks to reduce overhead

Error Handling and Process Management

- Comprehensive error checking for:
 - Process creation
 - Memory allocation
 - File operations
 - Result aggregation

4. Speedup and Efficiency Calculations

To comprehensively analyze the performance, we calculated the speedup and efficiency for different process counts, considering the processor's **6 performance cores (6-P)** for my computer :

1. 2 Processes Calculation:

- Baseline (Naive Approach) Time: 353 seconds
- Multiprocessing Time: 276 seconds
- Speedup = $353 / 276 = 1.23x$
- Efficiency = $1.23 / 2 = 0.615$ (61.5%) ”Note : it’s for the using of **2 Processes** ”

2. 4 Processes Calculation:

- Baseline Time: 353 seconds
- Multiprocessing Time: 213 seconds
- Speedup = $353 / 213 = 1.59x$
- Efficiency = $1.59 / 4 = 0.398$ (39.8%) ”Note : it’s for the using of **4 Processes** ”

3. 6 Processes Calculation:

- Baseline Time: 353 seconds
- Multiprocessing Time: 186 seconds
- Speedup = $353 / 186 = 1.82x$
- Efficiency = $1.82 / 6 = 0.303$ (30.3%) ”Note : it’s for the using of **6 Processes** ”

4. 8 Processes Calculation:

- Baseline Time: 353 seconds
- Multiprocessing Time: 200 seconds
- Speedup = $353 / 200 = 1.695x$
- Efficiency = $1.695 / 8 = 0.212$ (21.2%) ”Note : it’s for the using of **8 Processes** ”

Parallel Components (Approximately 85%):

1. File chunk reading
2. Word tokenization
3. Word frequency counting
4. Local result generation

Serial Components (Approximately 15%):

1. Initial file size calculation
2. Result merging process
3. Final sorting of word frequencies
4. Overall program initialization and cleanup

```
// serial initial
long file_size = get_file_size("text8.txt");
long chunk_size = file_size / num_processes;

// (serial process)
WordFreq* merge_results(WordFreq** process_results, ...) {
    WordFreq* merged = malloc(...);
    for (int p = 0; p < num_processes; p++) {
        for (int i = 0; i < process_unique_words[p]; i++) {
            // Merging is inherently serial
            merge_word_frequencies();
        }
    }
    return merged;
}

// final sorting (serial)
heap_sort(final_results, final_unique_words);
```

Amdahl's Law Calculation

Amdahl's Law Formula: $\text{Speedup} = 1 / (s + (1-s)/n)$

- s = Serial Fraction (0.15 or 15%)
- n = Number of Processes

2 Processes Analysis

- Theoretical Calculation:
 - $s = 0.15$
 - $n = 2$
 - $\text{Speedup} = 1 / (0.15 + (1-0.15)/2)$
 - $\text{Speedup} = 1 / (0.15 + 0.425)$
 - Theoretical Maximum Speedup $\approx 1.67x$
- Experimental Results:
 - Execution Time: 276 seconds
 - Actual Speedup: $353 / 276 = 1.23x$
 - Efficiency: 61.5%
 - Deviation from Theory: 26.3% **less efficient**

4 Processes Analysis

- Theoretical Calculation:
 - $s = 0.15$
 - $n = 4$
 - $\text{Speedup} = 1 / (0.15 + (1-0.15)/4)$
 - $\text{Speedup} = 1 / (0.15 + 0.2125)$
 - Theoretical Maximum Speedup $\approx 2.24x$
- Experimental Results:
 - Execution Time: 213 seconds
 - Actual Speedup: $353 / 213 = 1.59x$

- Efficiency: 39.8%
- Deviation from Theory: 29% **less efficient**

6 Processes Analysis

- Theoretical Calculation:
 - $s = 0.15$
 - $n = 6$
 - $\text{Speedup} = 1 / (0.15 + (1-0.15)/6)$
 - $\text{Speedup} = 1 / (0.15 + 0.142)$
 - Theoretical Maximum Speedup $\approx 3.42x$
- Experimental Results:
 - Execution Time: 186 seconds
 - Actual Speedup: $353 / 186 = 1.82x$
 - Efficiency: 30.3%
 - Deviation from Theory: 46.8% **less efficient**

8 Processes Analysis

- Theoretical Calculation:
 - $s = 0.15$
 - $n = 8$
 - $\text{Speedup} = 1 / (0.15 + (1-0.15)/8)$
 - $\text{Speedup} = 1 / (0.15 + 0.10625)$
 - Theoretical Maximum Speedup $\approx 4.12x$
- Experimental Results:
 - Execution Time: 200 seconds
 - Actual Speedup: $353 / 200 = 1.69x$
 - Efficiency: 21.2%
 - Deviation from Theory: 59% **less efficient**

4. Discussion

Our multiprocessing word frequency analysis reveals the complex realities of parallel computing. Theoretical Amdahl's Law calculations suggested potential speedups from 1.67x to 4.12x, but actual performance showed more modest improvements, with 6 processes achieving the optimal 1.82x speedup.

The experimental results highlighted significant efficiency losses (26.3% to 59%), demonstrating that parallelization is not straightforward. The 6-process configuration represented the best balance between computational resources and performance, aligning with the processor's 6 performance cores.

Key bottlenecks included synchronization overhead, serial code limitations, and process management complexity. The 15% serial fraction critically constrained speedup potential, showing that not all computational tasks are equally parallelizable.

Practical insights include:

1. Minimizing serial code segments
2. Implementing intelligent load balancing
3. Understanding hardware architecture
4. Managing inter-process communication

The study emphasizes that theoretical speedup predictions require empirical validation, highlighting the complexity of parallel computing and the need for context-specific optimization strategies.

3. Multithreading Approach

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multithreading_Approach 2
Starting word frequency analysis with 2 threads...
Thread 0: Processed 3761963 words, 108346 unique words
Thread 1: Processed 3761877 words, 108346 unique words
Thread 1: Processed 6652455 words, 147718 unique words
Thread 0: Processed 6654520 words, 147736 unique words
Thread 0 completed: 8485298 words processed, 171140 unique words found
Thread 1 completed: 8485298 words processed, 171140 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word                Frequency
-----
the                  1076960
of                   600742
and                  420226
one                  380148
in                   371238
a                    331668
to                   319104
zero                 251902
nine                 224582
is                   192664

Total execution time: 244.00 seconds
```

```
amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multithreading_Approach 4
Starting word frequency analysis with 4 threads...
Thread 2: Processed 3287201 words, 100924 unique words
Thread 0: Processed 3310672 words, 101180 unique words
Thread 1: Processed 3316600 words, 101227 unique words
Thread 3: Processed 3298397 words, 101017 unique words
Thread 1 completed: 4257281 words processed, 115004 unique words found
Thread 0 completed: 4257281 words processed, 115004 unique words found
Thread 3 completed: 4257281 words processed, 115004 unique words found
Thread 2 completed: 4257281 words processed, 115004 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word                Frequency
-----
the                  1093528
of                   607164
and                  428220
one                  404760
in                   376856
a                    323440
to                   317176
zero                 270240
nine                 235852
two                  198520

Total execution time: 154.00 seconds
```

```

amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multithreading_Approach 6
Starting word frequency analysis with 6 threads...
Thread 4 completed: 2840460 words processed, 93871 unique words found
Thread 3 completed: 2840460 words processed, 93871 unique words found
Thread 2 completed: 2840460 words processed, 93871 unique words found
Thread 5 completed: 2840461 words processed, 93871 unique words found
Thread 0 completed: 2840460 words processed, 93871 unique words found
Thread 1 completed: 2840460 words processed, 93871 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word                Frequency
-----
the                  1079976
of                   611940
and                  428886
one                  426918
in                   378408
a                    319584
to                   312906
zero                 271872
nine                 246216
two                  200346

Total execution time: 125.00 seconds

```

```

amir_alrashayda@DESKTOP-L6AF422:~/VSC$ ./Multithreading_Approach 8
Starting word frequency analysis with 8 threads...
Thread 3 completed: 2131263 words processed, 80848 unique words found
Thread 0 completed: 2131263 words processed, 80848 unique words found
Thread 7 completed: 2131265 words processed, 80848 unique words found
Thread 6 completed: 2131263 words processed, 80848 unique words found
Thread 4 completed: 2131263 words processed, 80848 unique words found
Thread 5 completed: 2131263 words processed, 80848 unique words found
Thread 2 completed: 2131263 words processed, 80848 unique words found
Thread 1 completed: 2131263 words processed, 80848 unique words found

Sorting words by frequency...

Top 10 most frequent words:
Word                Frequency
-----
the                  1075169
of                   620232
one                  440304
and                  423504
in                   377128
a                    316824
to                   315808
zero                 261552
nine                 247176
two                  201016

Total execution time: 118.00 seconds

```

1. Performance Analysis & Implementation Methodology

The threading approach fundamentally differs from the naive method by implementing parallel processing through multiple threads. This strategy allows simultaneous processing of different file chunks, significantly reducing the overall execution time. The implementation utilized POSIX pthread libraries for creating and managing parallel execution units, enabling efficient data exchange and synchronization.

Multiprocessing Implementation Methodology The threading approach in this implementation was achieved using POSIX thread (pthread) management APIs. Key implementation strategies included:

Thread Creation and Management

- `pthread_create()`: Primary function for creating threads
- `pthread_join()`: Function for synchronizing thread completion
- `pthread_t`: Thread identifier for tracking and managing threads

Inter-Thread Communication Mechanisms

- Shared Memory:
 - Direct memory access between threads
 - Minimal overhead compared to inter-process communication
 - Utilized global or dynamically allocated shared structures
 - Synchronization Primitives:
 - Mutex locks for critical section protection
 - Atomic operations for thread-safe updates
 - Minimal locking to reduce contention overhead
3. Speedup and Efficiency Calculations To comprehensively analyze the performance, we calculated the speedup and efficiency for different thread counts:

2. Threads Calculation:

- Baseline (Naive Approach) Time: 353 seconds
- Threading Time: 244 seconds
- Speedup = $353 / 244 = 1.45x$
- Efficiency = $1.45 / 2 = 0.725$ (72.5%)

4-Threads Calculation:

- Baseline Time: 353 seconds
- Threading Time: 154 seconds
- Speedup = $353 / 154 = 2.29x$
- Efficiency = $2.29 / 4 = 0.573$ (57.3%)

6-Threads Calculation:

- Baseline Time: 353 seconds
- Threading Time: 125 seconds
- Speedup = $353 / 125 = 2.82x$
- Efficiency = $2.82 / 6 = 0.470$ (47.0%)

8-Threads Calculation:

- Baseline Time: 353 seconds
- Threading Time: 118 seconds
- Speedup = $353 / 118 = 2.99x$
- Efficiency = $2.99 / 8 = 0.374$ (37.4%)

Amdahl's Law Calculation for Threads

Amdahl's Law :

- $\text{Speedup} = 1 / (s + (1-s)/n)$
- Serial Fraction (0.15 or 15%)
- n = Number of Threads

2-Threads Analysis

Theoretical Calculation: $s = 0.15$

- $n = 2$
- $\text{Speedup} = 1 / (0.15 + (1-0.15)/2)$
- $\text{Speedup} = 1 / (0.15 + 0.425)$
- Theoretical Maximum Speedup $\approx 1.67x$

Experimental Results: Execution Time: 244 seconds

- Actual Speedup: $353 / 244 = 1.45x$
- Efficiency: 72.5% Deviation from Theory: 13.2% less efficient

4-Threads Analysis

Theoretical Calculation: $s = 0.15$

- $n = 4$
- $\text{Speedup} = 1 / (0.15 + (1-0.15)/4)$
- $\text{Speedup} = 1 / (0.15 + 0.2125)$
- Theoretical Maximum Speedup $\approx 2.24x$

Experimental Results: Execution Time: 154 seconds

- Actual Speedup: $353 / 154 = 2.29x$

- Efficiency: 57.3% Deviation from Theory: 2.2% less efficient

6-Threads

Analysis Theoretical Calculation: $s = 0.15$

- $n = 6$
- $\text{Speedup} = 1 / (0.15 + (1-0.15)/6)$
- $\text{Speedup} = 1 / (0.15 + 0.142)$
- Theoretical Maximum Speedup $\approx 3.42x$

Experimental Results: Execution Time: 125 seconds

- Actual Speedup: $353 / 125 = 2.82x$
- Efficiency: 47.0% Deviation from Theory: 17.5% less efficient

8-Threads

Analysis Theoretical Calculation: $s = 0.15$

- $n = 8$
- $\text{Speedup} = 1 / (0.15 + (1-0.15)/8)$
- $\text{Speedup} = 1 / (0.15 + 0.10625)$
- Theoretical Maximum Speedup $\approx 4.12x$

Experimental Results: Execution Time: 118 seconds

- Actual Speedup: $353 / 118 = 2.99x$
- Efficiency: 37.4% Deviation from Theory: 27.4% less efficient

3. Discussion

Our threading word frequency analysis reveals the nuanced performance characteristics of parallel computing. Experimental results demonstrated significant performance improvements compared to the naive approach, with thread counts scaling more efficiently than processes.

The threading implementation showed progressive speedup, with 8 threads achieving nearly 3x acceleration compared to the sequential approach. Unlike process-based parallelization, threading exhibited more consistent efficiency gains, likely due to lower synchronization and communication overhead.

The 6-thread configuration represented an optimal balance between computational resources and performance, closely aligning with the processor's 6 performance cores. However, performance gains exhibited diminishing returns beyond 6 threads, highlighting the importance of understanding hardware architecture and thread scheduling.


Note: i'm using i5-12600KF with the wsl (6 P-cores \times 2 "Hyper-Threading") so using 6-8 logical threads , Likely utilizing Hyper-Threading technology so it did provides best balance of performance and overhead for the 6-8 logical threads , and it's likely has mapped to my 12 logical cores (6 P-cores \times 2 Hyper-Threading)







Performance Comparison of Parallel table

Performance Comparison of Parallel Approaches #Amir Rasmi AL-Rashayda #1222596			
Metric	Sequential	Multiprocessing	Multithreading
Total Execution Time	353 seconds	200 seconds	118 seconds
Speedup (2 Units)	1.0x	1.23x	1.45x
Speedup (4 Units)	1.0x	1.59x	2.29x
Speedup (6 Units)	1.0x	1.82x	2.82x
Speedup (8 Units)	1.0x	1.69x	2.99x
Efficiency (2 Units)	-	61.5%	72.5%
Efficiency (4 Units)	-	39.8%	57.3%
Efficiency (6 Units)	-	30.3%	47.0%
Efficiency (8 Units)	-	21.2%	37.4%
Best Performance	Baseline	6 Processes	8 Threads
Peak Speedup	1.0x	1.82x (6P)	2.99x (8T)
Resource Utilization	Low	Moderate	High
Synchronization	Minimal	High	Moderate
Memory Efficiency	Low	Moderate	High
Scalability	Poor	Limited	Good
Implementation	Low	High	Moderate
Recommended Approach	Multithreading (8 threads)		
Key Advantages	1. Highest speedup (2.99x) 2. Most efficient resource utilization 3. Lower synchronization overhead 4. Better hardware architecture alignment		

Conclusion

The revolutionary potential of parallel computing is demonstrated by comparing sequential, multiprocessing, and multithreading systems for word frequency analysis. With an impressive 2.99x speedup over the sequential method and an execution time reduction from 353 to 118 seconds, my research shows that multithreading—especially with 8 threads—offers the most performance gains. The multithreading approach is the most efficient way to handle large-scale text processing tasks because it maximizes computational distribution, minimizes synchronization overhead, and effectively utilizes hardware resources. These results not only emphasize how crucial parallel processing methods are, but they also offer insightful information on how to create high-performance computational solutions that can significantly increase computational efficiency in various data processing and analysis fields.

 You have to submit a report, along with your code, that discusses the following:

-  1- Describe your environment: computer specs (Cores? Speed? Memory?), OS, programming language, and IDE tool, and whether you used a virtual machine.
-  2- How you achieved the multiprocessing and multithreading requirements, i.e. The API and functions that you used.
-  3- An analysis according to Amdahl's law. What percentage is the serial part of your code? What is the maximum speedup according to the available number of cores? What is the optimal number of child processes or threads?
-  4- A table that compares the performance of the 3 approaches.
-  5- Comment on the differences in performance, and
-  6- Conclusion.