Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS3390: Operating Systems

Project Report

Prepared by:

Taleed Mahmoud Hamadneh

1220006

Instructor:

Dr. Yazan Abu Farha

Section: 1

Date: Nov.29.2024

Introduction

This project aims to develop a program to identify the top 10 most frequent words in the enwik8 dataset using three approaches: a naive approach, a multiprocessing approach, and a multithreading approach. The program's execution time will be measured for each method, with the multiprocessing and multithreading approaches tested using different numbers of child processes or threads (2, 4, 6, 8). The report will discuss the development environment, the parallelization techniques used, an analysis based on Amdahl's law, and a performance comparison of the three approaches.

Table of Contents

Table of Figures

List of Tables

Theory

1- Naïve Approach

The naive approach is a straightforward method where the program executes tasks sequentially within a single process or thread, without utilizing any form of parallelism. This approach is simple to implement and does not require managing multiple processes or threads, making it suitable for smaller or less complex tasks. However, since the tasks are executed one at a time, there is no inherent performance gain, and the method becomes inefficient for larger, computationally intensive problems. As a result, the naive approach may not scale well when handling large datasets, as it fails to take advantage of the potential speedups offered by parallel processing techniques.

2- Multiprocessing Approach

The multiprocessing approach utilizes multiple child processes to execute tasks concurrently, allowing for parallel execution across multiple CPU cores. Unlike the naive approach, which runs tasks sequentially in a single process, the multiprocessing approach divides the workload into smaller chunks, each processed by a separate child process. This parallelization can significantly speed up the execution of computationally intensive tasks, as multiple cores can work simultaneously on different parts of the problem. The efficiency of the multiprocessing approach depends on the system's hardware, with performance improvements seen when the number of processes aligns with the number of available cores. However, the overhead of creating and managing processes must be considered.

3- Multithreading Approach

The multithreading approach allows multiple threads to run in parallel within a single process, sharing the same memory space. Tasks are split into smaller subtasks, with each handled by a separate thread. Because threads share memory, communication between them is more efficient than in multiprocessing, where processes have separate memory spaces. Multithreading can boost performance for tasks that involve a lot of parallelism and shared data. However, it requires careful management to prevent issues like race conditions and deadlocks.

Environment

The program was developed on a system equipped with an 11th Gen Intel Core i7-11800H processor, featuring 8 physical cores and 16 logical cores, operating at a speed of 2.3 GHz. The machine is equipped with 16 GB of RAM. The operating system used is Linux, running as a subsystem on Windows through the Windows Subsystem for Linux (WSL) without the need for a full virtual machine. The program was written in C, with Visual Studio serving as the (IDE).

Implementations and APIs

- **Naïve Approach**

  The naive approach was implemented by processing the file's content sequentially without any parallelization. Initially, the words from the dataset were read and stored in an array of structures, with each structure representing a word. A second array of structures was then created to store unique words along with their respective frequencies. The program iterated through the array of file words, checking for each word's presence in the frequency array. If the word was already present, its frequency was incremented; otherwise, the word was inserted into the array with an initial frequency of 1. After populating the frequency array, a nested loop was used to identify the top 10 most frequent words. The outer loop iterated 10 times, while the inner loop traversed the entire frequency array during each iteration to find the word with the highest frequency that had not yet been selected.

- **Multiprocessing Approach**

The multiprocessing approach was designed to utilize multiple processes to divide and process the dataset in parallel. Initially, the program reads the data from the file, similar to the naive approach, storing all words in an array of structures. To facilitate inter-process communication, an array of pipes was created, where each pipe corresponds to a specific process. The array was structured as [number of processes][2], with each pipe containing a write and read end. Since pipes have a fixed capacity for data transfer, the write and read ends of the pipes were set to non-blocking mode, allowing large datasets to be sent in chunks without causing blocking or overflow errors.

The program then created multiple processes, with each process allocated its own local array of structures to store words and their frequencies. To divide the workload, the total number of words in the file was divided by the number of processes to calculate the chunk size. Each process was assigned a start and end index corresponding to its portion of the data in the file array. Each process iterated over its assigned range of words, storing unique words in its local array. If a word already existed in the local array, its frequency was incremented; otherwise, it was added to the array with an initial frequency of 1.

Once a process completed its assigned work, it sent its local array of structures through its corresponding pipe. To handle the large size of the data, the array was divided into smaller chunks and sent in parts, avoiding the pipe becoming full. After all processes completed their tasks, the main program waited for their termination, ensuring all data was written to the pipes. The results from each pipe were then read in parts and merged into a single array. Finally, the program identified the top 10 most frequent words using the same method as in the naive approach, iterating through the merged array to find the words with the highest frequencies.

Libraries and APIs used in the multiprocessing approach:

- unistd.h Library:

  pipe(int pipefd[2]): Creates a pipe for communication between processes.

  fork(): Creates a new child process.

  close(int fd): Closes the read or write end of a pipe.

  write(int fd, const void *buf, size_t count): Writes data to the pipe.

  read(int fd, void *buf, size_t count): Reads data from the pipe.

- fcntl.h Library:

  fcntl(int fd, int cmd, ...): Sets the pipe to non-blocking mode.

- sys/wait.h Library:

  wait(NULL): Waits for child processes to finish.

- **Multithreading Approach**

  In the multithreading implementation, the program first reads the data from the file, following the same procedure as the naive and multiprocessing approaches. Two shared memory structures are initialized: one for the read-only array containing the file data and another for an array of structs to store words and their frequencies. Two mutexes are created to ensure thread-safe access to the shared memory. An array of threads is initialized based on the desired number of threads, and for each thread, a range specifying its start and end indices is calculated. Threads are then created and assigned to a runner function that processes their respective ranges. Within the runner function, each thread loops through its assigned range and checks if the current word exists in the shared output array. If the word exists, the thread locks the corresponding mutex, updates the frequency, and unlocks it. If the word is not found, the thread locks the second mutex, inserts the word with an initial frequency, and then unlocks it. Once all threads complete, the program identifies the top 10 most frequent words, similar to the naive and multiprocessing approaches.

Libraries and APIs used in the multithreading approach:

- pthread.h Library:

  pthread_create: Creates a new thread, specifying the thread function and any arguments to pass.

  pthread_join: Waits for a specific thread to finish execution before proceeding.

pthread_mutex_init: Initializes a mutex to allow thread-safe access to shared data.

pthread_mutex_lock: Locks a mutex to ensure exclusive access to a shared resource.

pthread_mutex_unlock: Unlocks a previously locked mutex, allowing other threads to access the resource.

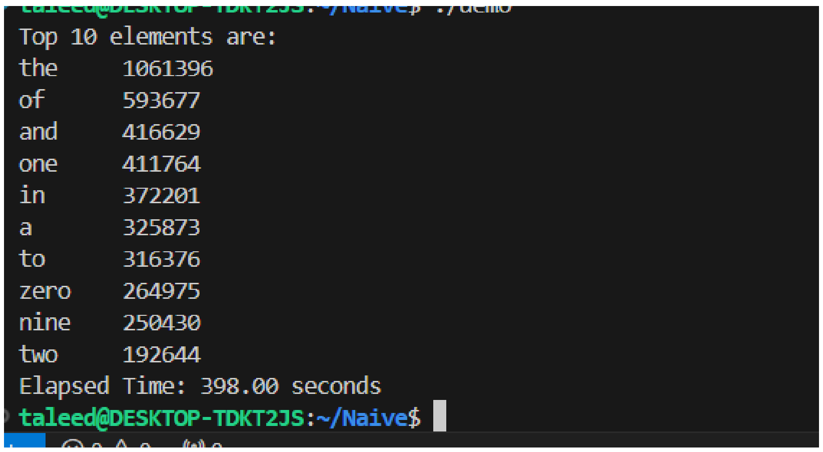
pthread_mutex_destroy: Destroys a mutex when it is no longer needed.

Results and Sample runs

- Naïve Approach

Table 1 Naive Execution time

Execution Time (sec)
398



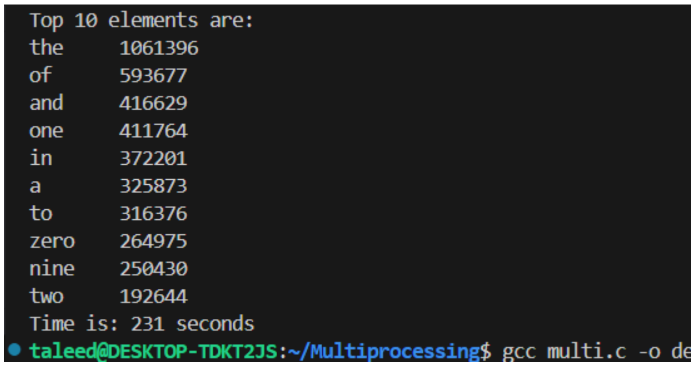
```
taleed@DESKTOP-TDKT2JS:~/Naive$ ./demo
Top 10 elements are:
the      1061396
of       593677
and      416629
one      411764
in       372201
a        325873
to       316376
zero     264975
nine     250430
two      192644
Elapsed Time: 398.00 seconds
taleed@DESKTOP-TDKT2JS:~/Naive$
```

Figure 1 Naive result

- Multiprocessing Approach

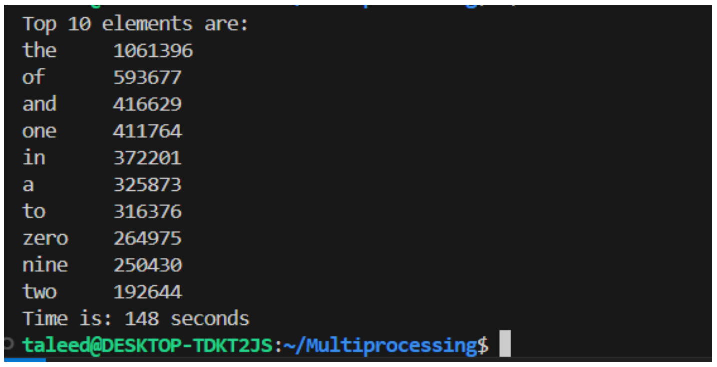
Table 2 Multiprocessing Execution time

Number of processes	Execution Time (sec)
2	231
4	148
6	100
8	88



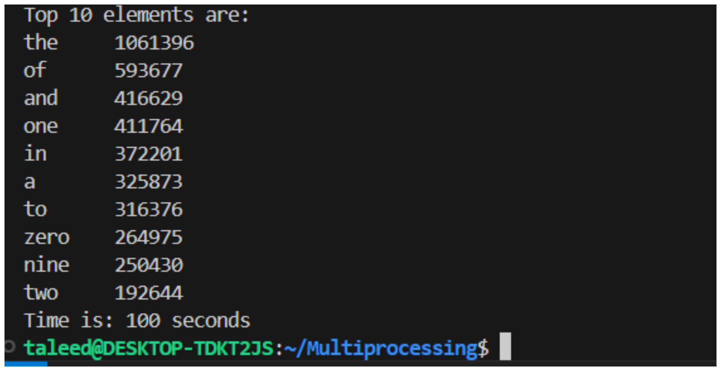
```
Top 10 elements are:
the     1061396
of      593677
and     416629
one     411764
in      372201
a       325873
to      316376
zero    264975
nine    250430
two     192644
Time is: 231 seconds
taleed@DESKTOP-TDKT2JS:~/Multiprocessing$ gcc multi.c -o de
```

Figure 2 Two processes results



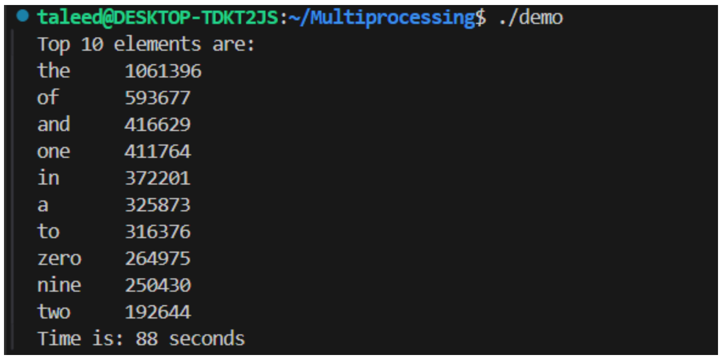
```
Top 10 elements are:
the     1061396
of      593677
and     416629
one     411764
in      372201
a       325873
to      316376
zero    264975
nine    250430
two     192644
Time is: 148 seconds
taleed@DESKTOP-TDKT2JS:~/Multiprocessing$
```

Figure 3 Four processes results



```
Top 10 elements are:
the     1061396
of      593677
and     416629
one     411764
in      372201
a       325873
to      316376
zero    264975
nine    250430
two     192644
Time is: 100 seconds
taleed@DESKTOP-TDKT2JS:~/Multiprocessing$
```

Figure 4 Six processes results



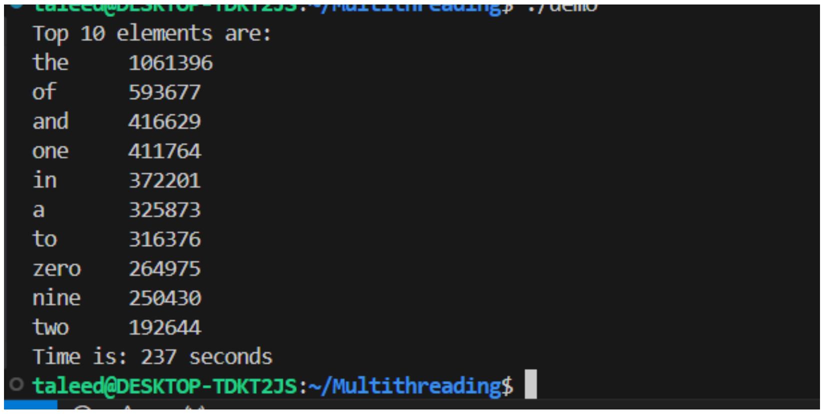
```
taleed@DESKTOP-TDKT2JS:~/Multiprocessing$ ./demo
Top 10 elements are:
the     1061396
of      593677
and     416629
one     411764
in      372201
a       325873
to      316376
zero    264975
nine    250430
two     192644
Time is: 88 seconds
```

Figure 5 Eight processes results

- Multithreading Approach

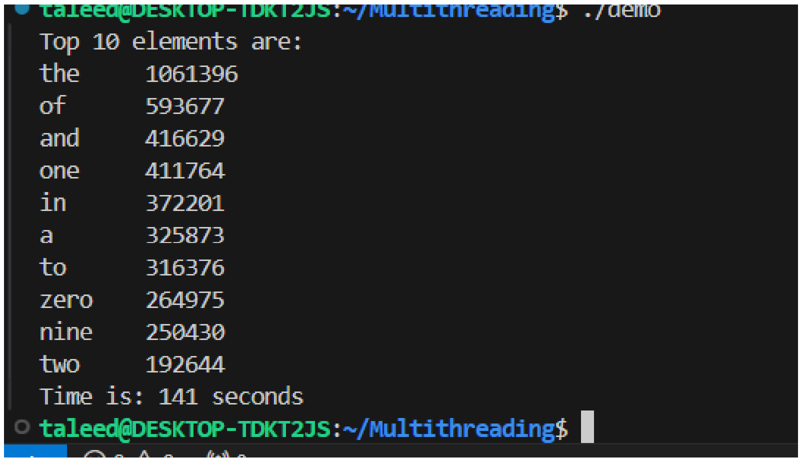
Table 3 Multithreading execution time

Number of threads	Execution Time (sec)
2	237
4	141
6	101
8	90



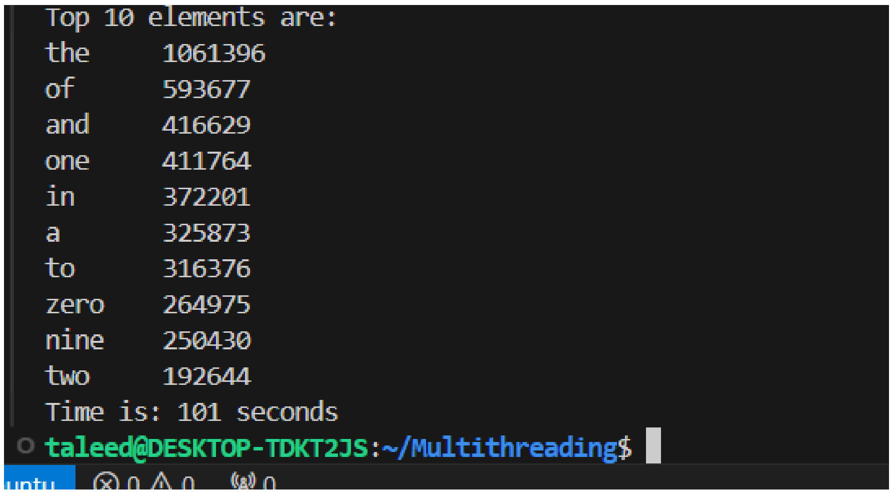
```
Top 10 elements are:
the      1061396
of       593677
and      416629
one      411764
in       372201
a        325873
to       316376
zero     264975
nine     250430
two      192644
Time is: 237 seconds
taleed@DESKTOP-TDKT2JS:~/Multithreading$
```

Figure 6 Two threads results



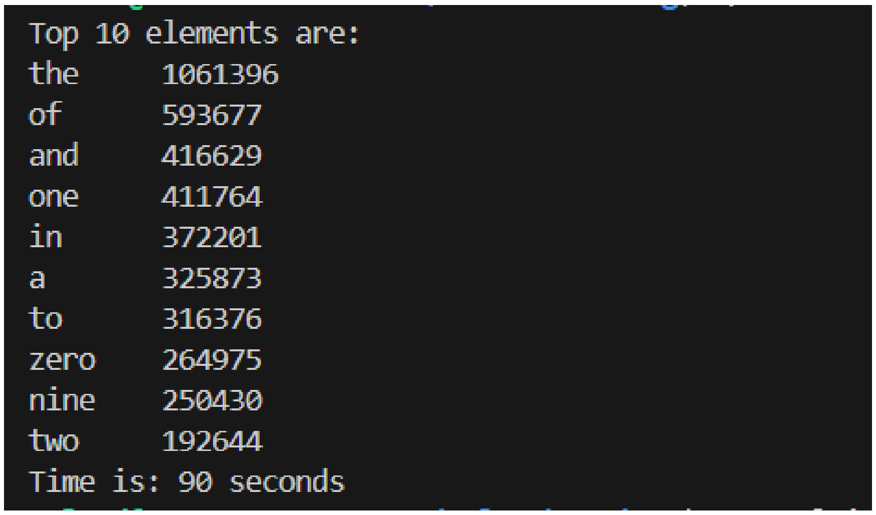
```
taleed@DESKTOP-TDKT2JS:~/Multithreading$ ./demo
Top 10 elements are:
the      1061396
of       593677
and      416629
one      411764
in       372201
a        325873
to       316376
zero     264975
nine     250430
two      192644
Time is: 141 seconds
taleed@DESKTOP-TDKT2JS:~/Multithreading$
```

Figure 7 Four threads results



```
Top 10 elements are:
the      1061396
of       593677
and      416629
one      411764
in       372201
a        325873
to       316376
zero     264975
nine     250430
two      192644
Time is: 101 seconds
taleed@DESKTOP-TDKT2JS:~/Multithreading$
```

Figure 8 Six threads results



```
Top 10 elements are:
the      1061396
of       593677
and      416629
one      411764
in       372201
a        325873
to       316376
zero     264975
nine     250430
two      192644
Time is: 90 seconds
```

Figure 9 Eight threads results

Performance Comparison and Analysis

Performance (Throughput) = 1 / Execution time.

Table 4 Performance comparison

Approach	Execution time (sec)	Performance (units per second)
Naïve	398	2.512 e-3
Multiprocessing (2)	231	4.329 e-3
Multiprocessing (4)	148	6.756 e-3
Multiprocessing (6)	100	0.01
Multiprocessing (8)	88	0.0113
Multithreading (2)	237	4.219 e-3
Multithreading (4)	141	7.092 e-3
Multithreading (6)	101	9.900 e-3
Multithreading (8)	90	0.01111

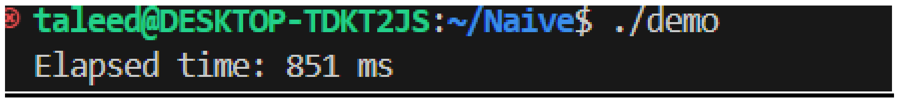
The table shows that the Naïve approach, using a single thread, has the longest execution time (398 seconds) and the lowest performance (2.512e-3 units/sec) since it doesn't use parallelism. In contrast, both Multiprocessing and Multithreading improve as the number of threads or processes increases, with their performance remaining similar. For example, with 4 processes or threads, the execution times are close: 148 seconds for Multiprocessing and 141 seconds for Multithreading. As the number of processes or threads grows, both approaches continue to improve, but their performance remains almost the same.

The similarity in performance comes from the system's 16 logical cores, which are not as powerful as the physical cores. Both Multiprocessing and Multithreading compete for the same CPU resources, and since the task is not big enough to use all the cores effectively, the performance gains start to level off. Moreover, the system's 8 physical cores also limit how much the performance can improve. Furthermore, managing multiple threads or processes adds some overhead, which slows things down. Even though the two approaches work differently, they both perform similarly because of these system limitations, and both are much faster than the Naïve approach.

Amdahl's law Analysis

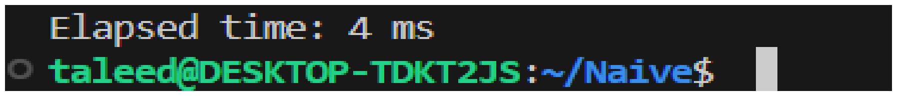
The program consists of two main parts: the serial and the parallel parts. In the serial part, the data is read from the file, and the top 10 most frequent words are identified. The parallel part handles the rest of the work, including searching for words and counting their frequency.

<u>Serial Part time:</u>



```
taleed@DESKTOP-TDKT2JS:~/Naive$ ./demo
Elapsed time: 851 ms
```

Figure 10 Reading from file time (serial)

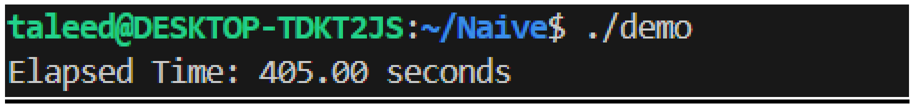


```
Elapsed time: 4 ms
taleed@DESKTOP-TDKT2JS:~/Naive$
```

Figure 11 Top 10 time (serial)

$T_{serial} = 855$ ms

<u>Parallel Part time:</u>



```
taleed@DESKTOP-TDKT2JS:~/Naive$ ./demo
Elapsed Time: 405.00 seconds
```

Figure 12 Parallel part time

$T_{parallel} = 405$ sec

<u>Serial Part Percentage:</u>

$\mathbf{s} = T_{serial} / T_{total} = 855 \text{ ms} / 405.855 \text{ sec} = 2.106\text{ e-}3 = 0.21\ \%$

The maximum speedup according to the available number of cores:

$$Speed\ up = \frac{1}{\frac{1-s}{N} + s}$$

N = 16 logical cores

S = 2.106 e-3

1-S = 0.9978

Max Speed up = 15.511

Since the serial part of the program is small and the parallel part is large, increasing the number of threads or processes will keep improving the performance.

The optimal number of processes or threads appears to be 8, as it offers the best execution times of 88 and 90 seconds. However, as the number of threads increases to 16, the speedup continues to improve, reaching 15.5. This suggests that while 8 is a strong choice, performance can still benefit from going up to 16.

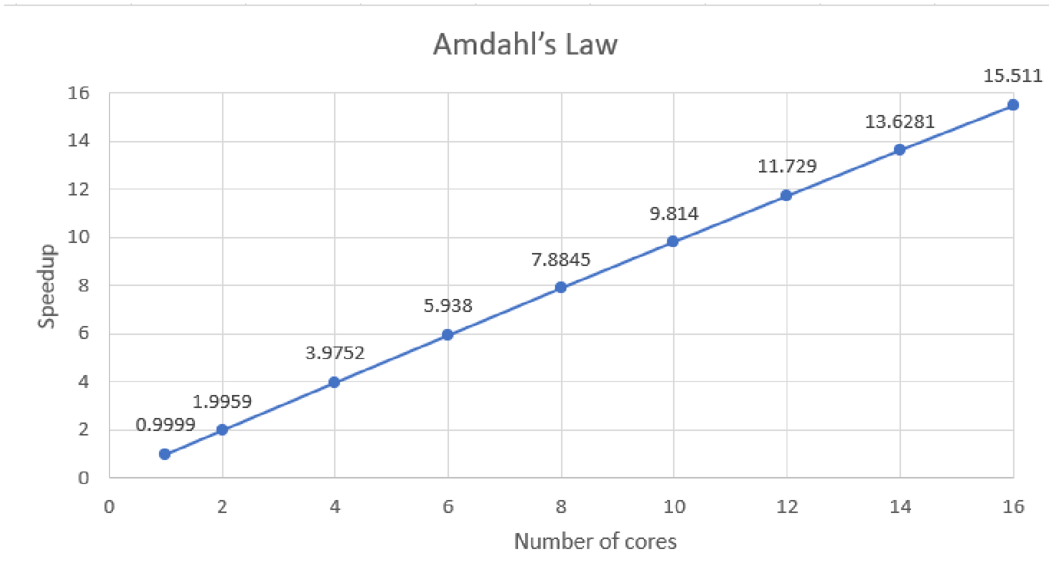


Figure 13 Amdahl's Law Chart

Conclusion

In this project, we implemented a program to find the top 10 most frequent words in the enwik8 dataset and compared three approaches: naive, multiprocessing, and multithreading. The naive approach, which did not use parallelization, had the slowest execution time, while multiprocessing and multithreading showed significant improvements as the number of processes or threads increased. Applying Amdahl's law highlighted how the serial portion of the code affects performance and the limitations in speedup even with additional cores. It also demonstrated how parallel processing can effectively enhance performance for large-scale data processing tasks.