Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Operating Systems

ENCS3390

Project Report

Prepared by:

Adnan Odeh **1220175**

Instructor: **Dr. Yazan Abu Farha**

Section: **1**

Date: **1st December 2024**

Abstract:

This project evaluates three approaches—multiprocessing, multithreading, and the naïve method—for identifying the top 10 most frequent words in the *enwik8* dataset. Multiprocessing leverages multiple CPU cores for parallel execution, while multithreading utilizes CPU threads for concurrent processing. The naïve method, in contrast, operates on a single core, resulting in slower execution times.

Performance testing was conducted on a single core for all approaches and extended to 2, 4, 6, and 8 cores or threads for multiprocessing and multithreading. Amdahl's Law was applied to analyze speedup trends as the number of cores increased, helping to identify the point of optimal performance improvement and maximum achievable speedup. The results reveal notable differences in execution efficiency, demonstrating the advantages of parallel and concurrent processing.

Table of Contents

List of Tables:

1. Computer Specification:

The environment used to work the Project:

Table 1.1: Computer Environment

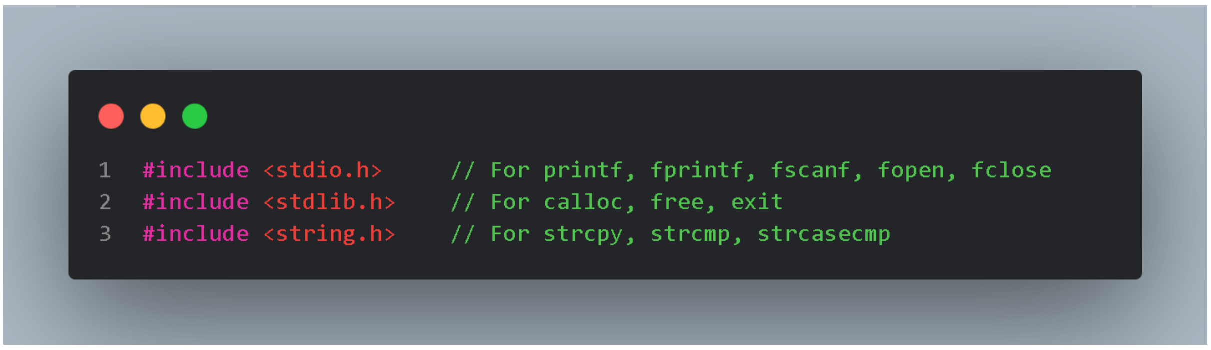
CPU	12th Gen Intel(R) Core (TM) i7-12700H 2.70 GHz
Number of Cores	14 Cores -> P-Cores: 6 E-Cores: 8
Number of Threads	20 Threads
Operating System	Linux Ubuntu (Dual Boot System)
IDE & Run Tools	Visual Studio Code & Linux Ubuntu Terminal

2. Design and Implementation:

2.1 Naive approach:

2.1.1 Libraries and APIs

Here are the necessary libraries used in this approach:



```c
#include <stdio.h>     // For printf, fprintf, fscanf, fopen, fclose
#include <stdlib.h>    // For calloc, free, exit
#include <string.h>    // For strcpy, strcmp, strcasecmp
```

Figure 2.1: Libraries & API's

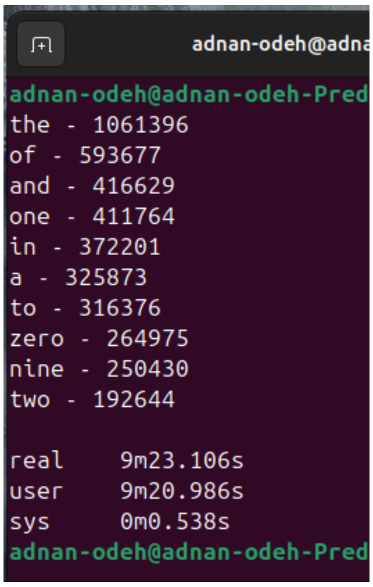
2.1.2 Workflow:

1. Initialize Memory for Data Structures:
   - Calculate the total number of words in the file by using the `numWords()` function.
   - Allocate memory for two arrays of a custom struct Data:
     - allWords: Stores all words from the file.
     - `finalData`: Stores unique words along with their frequencies after filtering.
2. Load All Words:
   - Fill the `allWords` array with words from the file using the `fillAllWords()` function.
3. Filter and Organize Data:
   - Iterate through the allWords array.
   - For each word, check if it already exists in the finalData array using the `addNew()` function:
     - If the word exists, increment its frequency.
     - If not, add the word to `finalData` with an initial frequency of 1.

4. Sort the Data by Frequency:
   - Use the `mergeSortN()` function to sort the finalData array in ascending order based on word frequency.
5. Retrieve the Top 10 Most Frequent Words:
   After sorting, extract and display the last 10 elements of the `finalData` array (the most frequent words).

2.1.3 Results and Discussion:

I ran the program using the naïve approach and analyzed both the code and the time taken, considering how it performs on a single core, this was the results.



```
adnan-odeh@adna
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    9m23.106s
user    9m20.986s
sys     0m0.538s
adnan-odeh@adnan-odeh-Pred
```

Figure 3.2: Results in naive

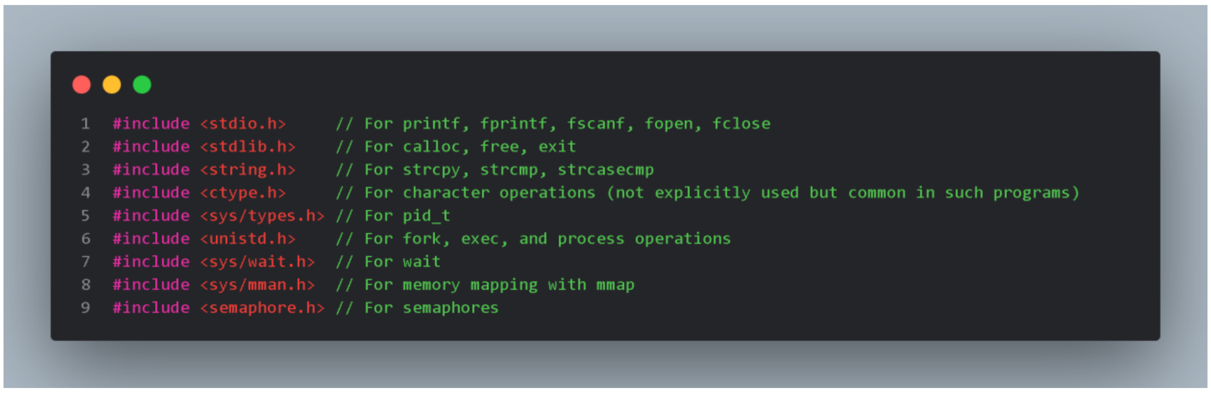
The results obtained are expected given the large size of the file, which requires reading nearly 17 million words.

Naïve here used by inserting the data without sorting it, if I sort the data before inserting it using any sorting algorithm, to the AddNew() function to find the unique words and search for the words added using binary search, this will reduce the time execution for the naïve approach.

2.2 Multiprocessing approach:

2.2.1 Libraries and APIs:

Here are the necessary libraries used in this approach:



```c
#include <stdio.h>     // For printf, fprintf, fscanf, fopen, fclose
#include <stdlib.h>    // For calloc, free, exit
#include <string.h>    // For strcpy, strcmp, strcasecmp
#include <ctype.h>     // For character operations (not explicitly used but common in such programs)
#include <sys/types.h> // For pid_t
#include <unistd.h>    // For fork, exec, and process operations
#include <sys/wait.h>  // For wait
#include <sys/mman.h>  // For memory mapping with mmap
#include <semaphore.h> // For semaphores
```

Figure 4.3: Multiprocess Libraries and APIs

2.2.2 Workflow:

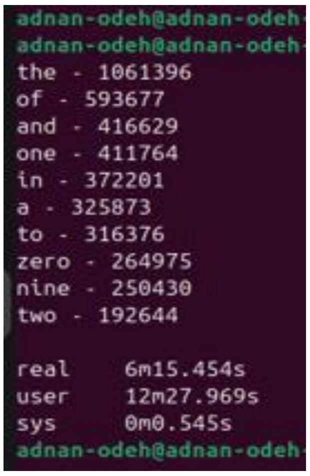
1. **Imported All necessary:**
   to handle memory allocation, process management, shared memory creation, and synchronization between processes.
2. **Defined constants** such as the total number of unique words and the maximum word length.
3. **Defined a Struct**
   - (struct Data) was defined with fields for storing a word and its corresponding frequency.
4. **Main Process Initialization:**
   - The total number of words in the file was counted.
     These words were stored in an array of structures `allWords` using the `fillAllWords()` function.
5. **Semaphore Initialization:**
   - A semaphore was initialized using `sem_init(&sem, 0, 1)`. Here, `&sem` is the address of the semaphore, 0 indicates that the semaphore is used for process scope, and 1 sets the initial value of the semaphore.
   - A new shared memory region (finalData) was allocated using `mmap()`. The parameters passed to `mmap()` were to ensure that the memory map is not corresponds to any file and initialized with the size of the data, and any change in the memory map will be visible to all processes.
   - This finalData will be shared across processes for further data processing.
6. **Process Creation**:
   - The program entered a loop to create multiple child processes, each of which would handle a part of the `allWords` array.
   - Each process was assigned a subset of words, and it was responsible for extracting the unique words and their frequencies, which were then stored in the `finalData` array.
7. **Word Extraction and Synchronization**:

- The `addWordsToArray(data, allWords, start, end)` function was used to extract unique words from the assigned range and store them in a temporary `data` array.
- The function `addNew()` was used to merge the data into the shared `finalData` array. This function included `sem_wait()` and `sem_post()` to ensure proper synchronization between processes, preventing race conditions while accessing and modifying the shared data.

8. **Parent Process Management**:
   - The parent process waited for the child processes to finish using `wait()`.
   - After the child processes terminated, the `finalData` array was sorted using the `mergeSortN()` algorithm to arrange the words by frequency.
   - The 10 most frequent words were then printed out.

2.2.3 Results and Discussion:

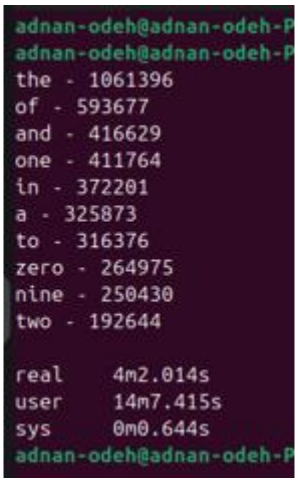
The figures below show the output for the test cases for this approach:



```
adnan-odeh@adnan-odeh
adnan-odeh@adnan-odeh
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    6m15.454s
user    12m27.969s
sys     0m0.545s
adnan-odeh@adnan-odeh
```

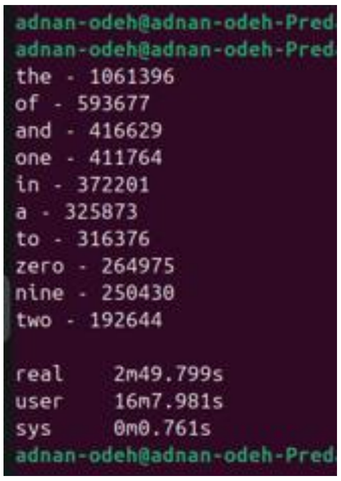
Figure 6.4: Results in Multiprocess 2Cores



```
adnan-odeh@adnan-odeh-P
adnan-odeh@adnan-odeh-P
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    4m2.014s
user    14m7.415s
sys     0m0.644s
adnan-odeh@adnan-odeh-P
```

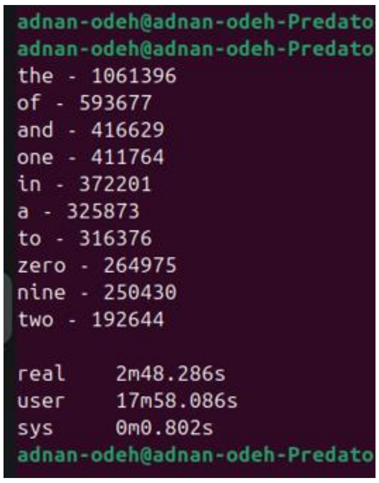
Figure 5.5: Results in Multiprocess 4Cores



```
adnan-odeh@adnan-odeh-Pred
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    2m49.799s
user    16m7.981s
sys     0m0.761s
adnan-odeh@adnan-odeh-Pred
```

Figure 8.6: Results in Multiprocess 6Cores



```
adnan-odeh@adnan-odeh-Predato
adnan-odeh@adnan-odeh-Predato
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    2m48.286s
user    17m58.086s
sys     0m0.802s
adnan-odeh@adnan-odeh-Predato
```

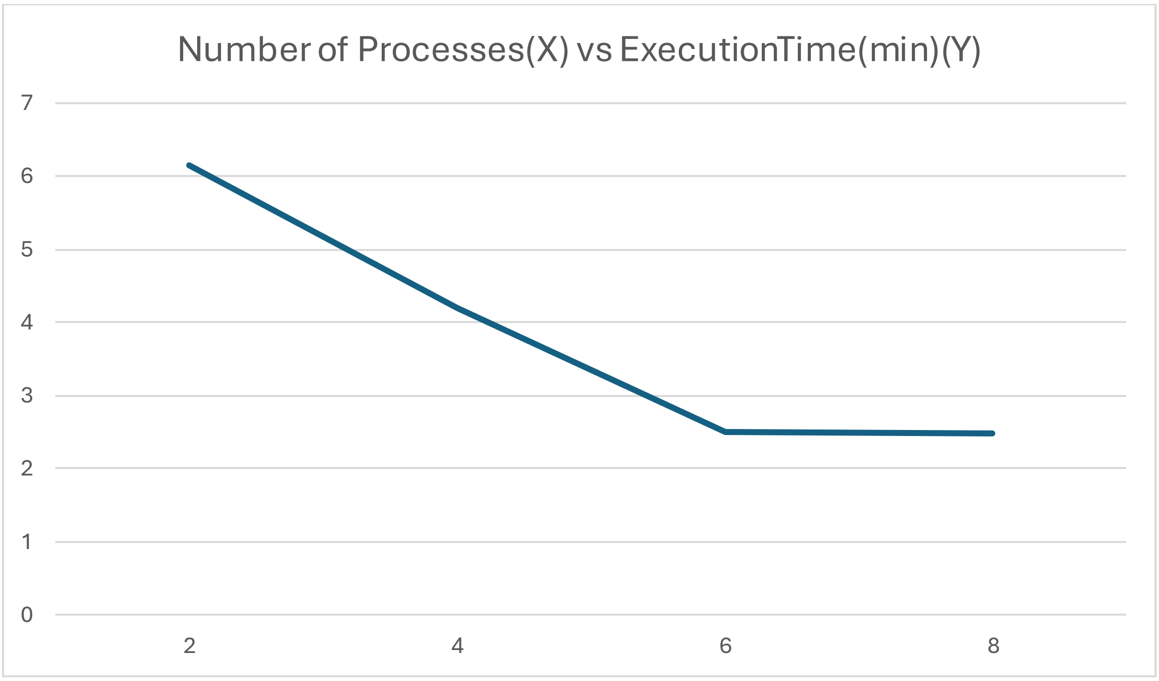
Figure 7.7: Results in Multiprocess 8Cores

All test cases, no matter how many cores were used, gave the same results. However, there was a clear difference in the time taken between the naive approach (single process) and the multiprocessing approach using 2, 4, 6, and 8 processes. The biggest improvement in speed was seen when going from 1 to 2 processes, 2 to 4, and 4 to 6 processes. After that, adding more cores didn't make big a difference and reached to a Dead Point.

The table and graph shown below explain what is the meaning of the Dead Point:

Table 2.1: Multiprocessing Comparison

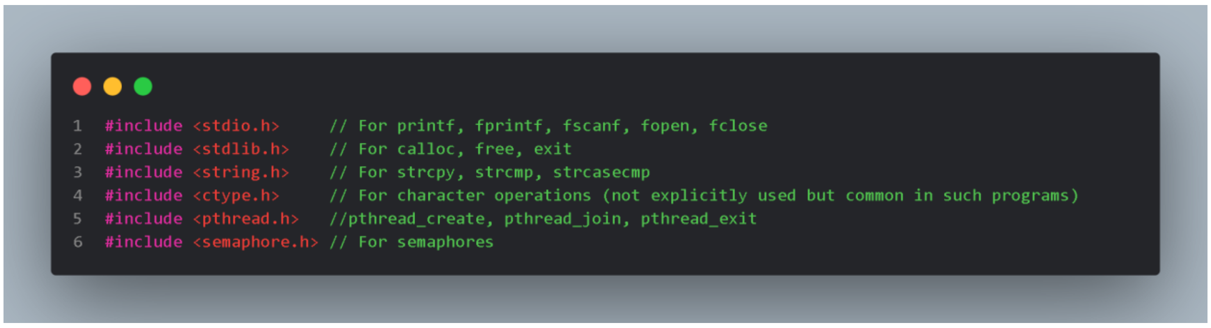
Number of Processes	Execution time (min)
2	**6.15**
4	**4.2**
6	**2.50**
8	**2.48**



2.3 Multithreading

2.3.1 Libraries and APIs:

Here are the necessary libraries used in this approach:



```c
#include <stdio.h>     // For printf, fprintf, fscanf, fopen, fclose
#include <stdlib.h>    // For calloc, free, exit
#include <string.h>    // For strcpy, strcmp, strcasecmp
#include <ctype.h>     // For character operations (not explicitly used but common in such programs)
#include <pthread.h>   //pthread_create, pthread_join, pthread_exit
#include <semaphore.h> // For semaphores
```

Figure 9.8: Libraries and APIs in Multithreading

2.3.2 Workflow:

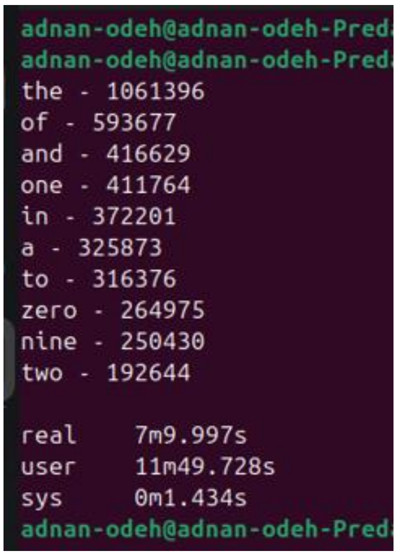
1. **Imported All Necessary Libraries**:
   - Libraries were imported to handle file operations, dynamic memory allocation, string manipulation, thread management, synchronization, and semaphores.
2. **Defined Constants**:
   - constant such that the maximum number of unique words, and the maximum length of the word.
3. **Defined a Struct**:
   - A struct `Data` was defined to store each word and its frequency, as well as the number of elements filled in the array
4. **Main Process Initialization**:
   - The total number of words in the text file was counted using the `numWords()` function.
   - These words were stored in the `allWords` array using the `fillAllWords()` function.
5. **Semaphore Initialization**:
   - A semaphore was initialized using `sem_init(&sem, 0, 1)`. This semaphore controls access to the shared `finalData` array.
   - Memory was allocated for both the `allWords` and `finalData` arrays using `calloc()`, and the semaphore was initialized to ensure synchronization between threads.
6. **Thread Creation**:
   - The program created multiple threads (with the number of threads being set by `numThreads`.
   - Each thread was assigned a subset of the `allWords` array to process, extracting unique words and their frequencies.
7. **Word Extraction and Synchronization**:

- Each thread used `addWordsToArray()` to extract words from its assigned range and store them in a `data` array.
- The `addNew()` function was used to update the shared `finalData` array with the word frequencies, using `sem_wait()` and `sem_post()` for synchronization, ensuring threads didn't race when modifying the shared data.

8. **Main Thread Management**:
   - The main thread waited for all threads to finish using `pthread_join()`.
   - Once all threads completed, the `finalData` array was sorted using the `mergeSortN()` function based on word frequencies.
   - The 10 most frequent words were printed.

2.3.3 Results and Discussion:

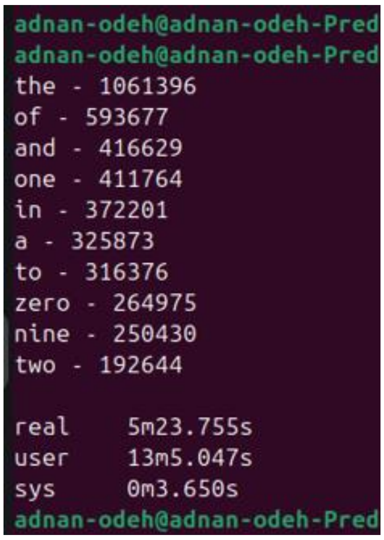
The figures below show the output for the test cases for this approach:



```
adnan-odeh@adnan-odeh-Pred
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    7m9.997s
user    11m49.728s
sys     0m1.434s
adnan-odeh@adnan-odeh-Pred
```

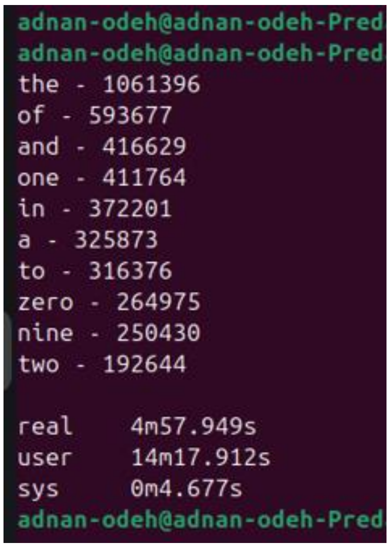
Figure 11.9: Results in Multithreading 2Threads



```
adnan-odeh@adnan-odeh-Pred
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    5m23.755s
user    13m5.047s
sys     0m3.650s
adnan-odeh@adnan-odeh-Pred
```

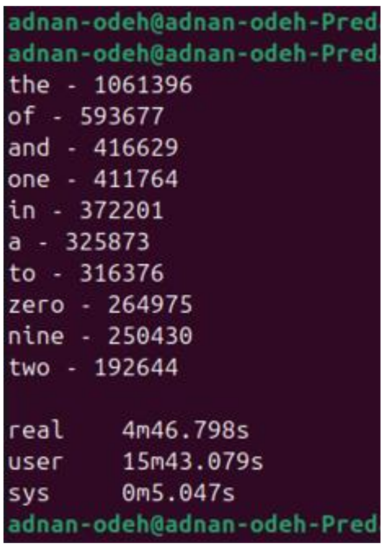
Figure 10.10: Results in Multithreading 4Threads



```
adnan-odeh@adnan-odeh-Pred
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    4m57.949s
user    14m17.912s
sys     0m4.677s
adnan-odeh@adnan-odeh-Pred
```

Figure 13.11: Results in Multithreading 6Threads



```
adnan-odeh@adnan-odeh-Pred
adnan-odeh@adnan-odeh-Pred
the - 1061396
of - 593677
and - 416629
one - 411764
in - 372201
a - 325873
to - 316376
zero - 264975
nine - 250430
two - 192644

real    4m46.798s
user    15m43.079s
sys     0m5.047s
adnan-odeh@adnan-odeh-Pred
```

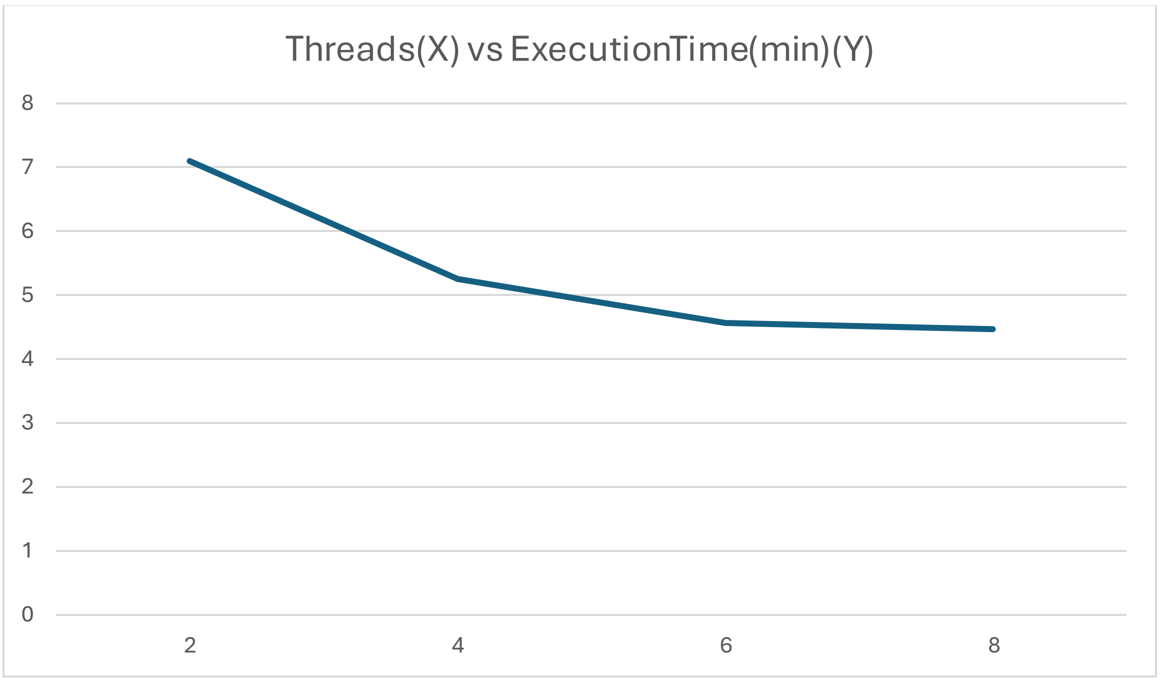
Figure 12.12: Results in Multithreading 8Threads

All test cases, no matter how many threads were used, gave the same results. However, there was a clear difference in the time taken between the naive approach (single core) and the multithreading approach using 2, 4, 6, and 8 threads. The biggest improvement in speed was seen when going from 1 to 2 Threads, 2 to 4. After that, adding more threads didn't make big a difference and reached to a Dead Point.

The table and graph shown below explain what is the meaning of the Dead Point:

Table 2.2: Multithreading Comparison

Number of Threads	Execution time (min)
2	**7.10**
4	**5.24**
6	**4.58**
8	**4.47**



3. Time analysis:

3.1 Multiprocessing analysis:

There was a breakthrough in the time between naïve approach and multiprocessing; to calculate this speed up I calculated the percentage of the serial part and parallel part by dividing the time of the naïve, and time taken for each multiprocessing test.

- The speed when run the 2 cores multiprocessing approach was $\frac{\text{Time taken by naive}}{\text{Time taken for two cores}}$
  $\frac{\mathbf{9.24}\,\boldsymbol{min}}{\mathbf{6.15}\,\boldsymbol{min}}$ **= speedup = 1.5024**
- The speed when run the 4 cores multiprocessing approach was $\frac{\text{Time taken by naive}}{\text{Time taken for four cores}}$
  $\frac{\mathbf{9.24}\,\boldsymbol{min}}{\mathbf{4.2}\,\boldsymbol{min}}$ **= speedup = 2.2**
- The speed when run the 6 cores multiprocessing approach was $\frac{\text{Time taken by naive}}{\text{Time taken for six cores}}$
  $\frac{\mathbf{9.24}\,\boldsymbol{min}}{\mathbf{2.50}\,\boldsymbol{min}}$ **= speedup = 3.696**
- The speed when run the 8 cores multiprocessing approach was $\frac{\text{Time taken by naive}}{\text{Time taken for eight cores}}$
  $\frac{\mathbf{9.24}\,\boldsymbol{min}}{\mathbf{2.48}\,\boldsymbol{min}}$ **= speedup = 3.7258**

These values show the speedup for each test, then to find the serial and parallel percentage, I calculated the avg of all speedup in the tests then using the **Amdhal's Law** Equation I got the serial part:

Avg Speedup = (1.5024+2.2+3.6969+3.7258)/4 = 2.706

N is considered = 14 cores (Total).

Amdhal's Law → $\frac{1}{Serial+\frac{1-Serial}{N}}$ **= Speedup**

Serial = 32%, Then Parallel = 68%

- **According the Analysis, we can conclude that the optimal number of cores is equal to 6, after this there is no clear difference in the performance.**

- **According to the Analysis, the maximum speedup is when number of cores is equal to 8.**

Notation: There is another way to calculate the serial and parallel part, is that by finding the time taken from the code that is considered as parallel part in the naïve code and dividing this time by the total time taken from the naïve, the results was that serial part is 4% and parallel part is 96%. I did not consider this way since the results from it is logically not true.

3.2 Multithreading analysis:

The difference is clear between time in the naïve approach and time in multithreading; to calculate this speed up I calculated the percentage of the serial part and parallel part by dividing the time of the naïve, and time taken for each multithreading test.

- The speed when run the 2 threads multithreading approach was $\frac{\text{Time taken by naive}}{\text{Time taken for two threads}}$
  $\frac{9.24\,min}{7.1\,min}$ **= speedup = 1.3**
- The speed when run the 4 threads multithreading approach was $\frac{\text{Time taken by naive}}{\text{Time taken for four threads}}$
  $\frac{9.24\,min}{5.24\,min}$ **= speedup = 1.7633**
- The speed when run the 6 threads multithreading approach was $\frac{\text{Time taken by naive}}{\text{Time taken for six threads}}$
  $\frac{9.24\,min}{4.58\,min}$ **= speedup = 2.0174**
- The speed when run the 8 threads multithreading approach was $\frac{\text{Time taken by naive}}{\text{Time taken for eight threads}}$
  $\frac{9.24\,min}{4.47\,min}$ **= speedup = 2.067**

These values show the speedup for each test, then to find the serial and parallel percentage, I calculated the avg of all speedup in the tests then using the **Amdhal's Law** Equation I got the serial part:

Avg Speedup = (1.3+1.7633+2.0174+2.067)/4 = 1.787

N is considered = 14 cores (Total).

Amdhal's Law → $\frac{1}{Serial+\frac{1-Serial}{N}}$ **= Speedup**

Serial = 52%, Then Parallel = 48%

- **According the Analysis, we can conclude that the optimal number of threads is equal to 6, after this there is no clear difference in the performance.**

- **According to the Analysis, the maximum speedup is when number of threads is equal to 8.**

Notations:

- **There is another way to calculate the serial and parallel part, is that by finding the time taken from the code that is considered as parallel part in the naïve code and dividing this time by the total time taken from the naïve, the results was that serial part is 7% and parallel part is 93%. I did not consider this way since the results from it is logically not true.**

- **There is a difference between the speedup in multiprocessing and speedup in multithreading due to several reasons such as context switching overhead, resource contention, and memory constraints. All of these caused that the serial part is more than parallel part since all of these calculations depend mainly on the speedup [1].**

4. Conclusion:

This project compared the performance of naïve, multiprocessing, and multithreading approaches for finding the top 10 most frequent words in the enwik8 dataset. Multiprocessing showed the best performance by using multiple cores effectively, while multithreading provided some improvement but was limited by issues like context switching and shared resource contention. Both methods reached a point where adding more cores or threads had little effect on performance.

Using Amdahl's Law, we found that the serial portion of the code affects how much speedup can be achieved. The best performance for both multiprocessing and multithreading was at six cores or threads, with the highest speedups at eight. This shows that the choice of method and system setup is important for getting the best results in parallel processing.

5. Reference:

- [Silberschatz. *Operating System Concepts* (10th ed.). Wiley.]
- [1]ThreadContentionScope