



**Faculty of Engineering and Technology
Electrical and Computer Engineering Department**

Operating Systems ENCS3390

First Semester 2024 / 2025

First Project

Student Name: Fadi Bassous

University ID: 1221005

Instructor's name: Dr. Mohammed Khalil

Section: 2

Date: 1 /12 /2024

Table of contents

Table of contents	III
Table of Figures.....	IV
List of Tables:	V
Results:.....	2
1. Naive Approach:	2
2. Multiprocessing Approach:.....	3
3. Multithreading Approach:	7
Questions:	11
1. Environment Description	11
2. How You Achieved the Multiprocessing and Multithreading Requirements: The API and Functions Used	11
3. What percentage is the serial part of your code?	13
Calculate the Maximum Speedup?	13
4. table that compares the performance of the 3 approaches:	14
5. Comment on the differences in performance:	14
conclusion	15

Table of Figures

Figure 1: Naive Result	2
Figure 2: Multiprocessing result (8 children)	3
Figure 3: Multiprocessing result (6 children)	4
Figure 4: Multiprocessing result (4 children)	5
Figure 5: Multiprocessing result (2 children)	6
Figure 6: Multithreading result (8 threads).....	7
Figure 7: Multithreading result (6 threads).....	8
Figure 8: Multithreading result (4 threads).....	9
Figure 9: Multithreading result (2 threads).....	10

List of Tables:

Table 1: Result for all 3 approaches.....	14
---	----

Theory

1. Naive Approach:

The naive approach executes all tasks sequentially within a single thread, processing one task at a time. This straightforward method does not leverage the advantages of parallelism, resulting in underutilization of multi-core processors. While simple to implement, the lack of concurrency leads to longer execution times, especially for computationally intensive or large-scale tasks.

2. Multiprocessing Approach:

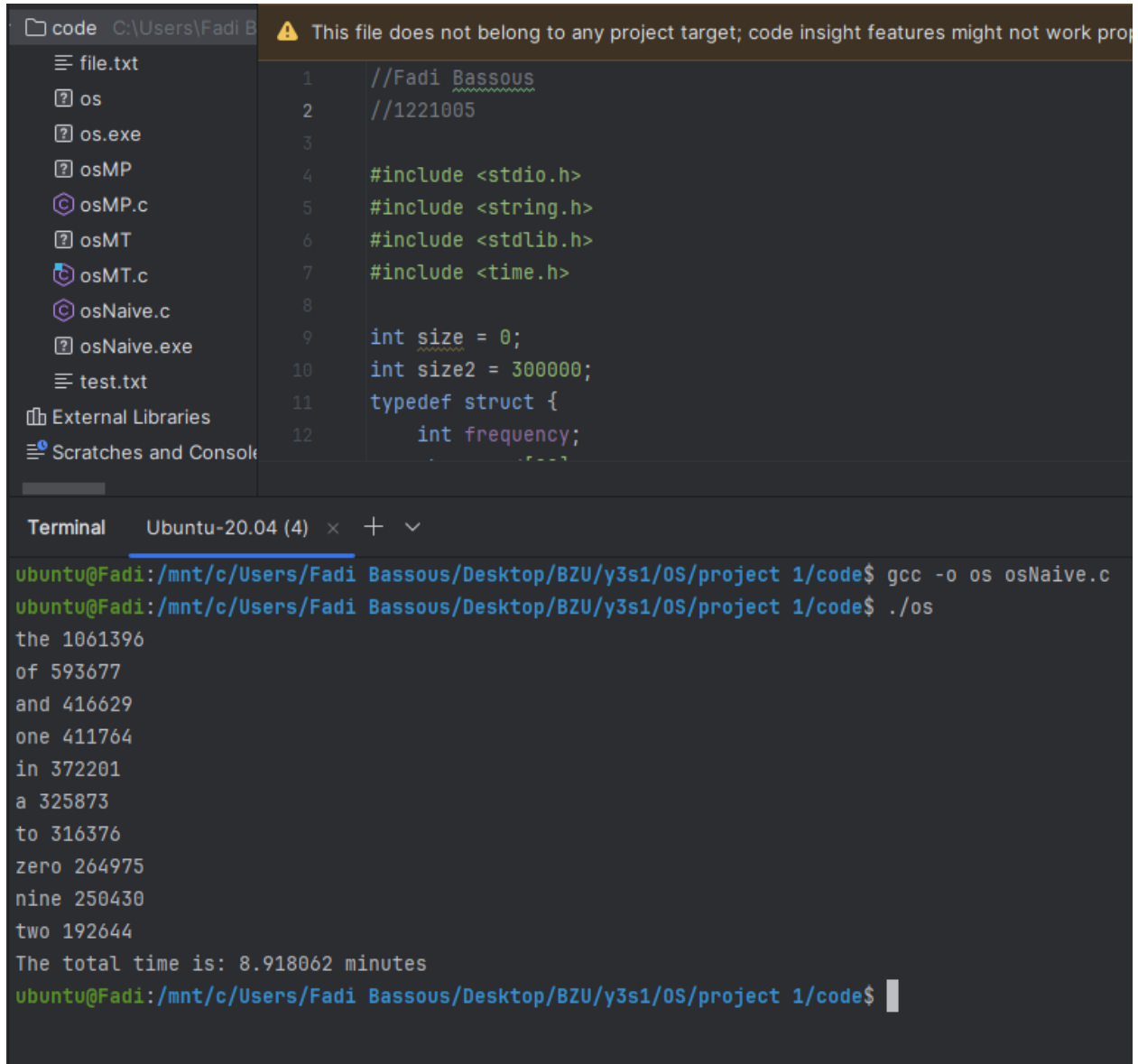
In the multiprocessing approach, multiple independent child processes are spawned to execute tasks in parallel. Each process runs in its own memory space, allowing effective utilization of multi-core CPUs by distributing the workload. This significantly reduces execution time compared to the naive approach. However, creating and managing processes incurs overhead, and inter-process communication can be slower due to isolated memory spaces.

3. Multithreading Approach:

The multithreading approach creates multiple threads within a single process, enabling tasks to run concurrently. Threads share the same memory space, making data sharing between threads more efficient than in multiprocessing. This approach is well-suited for tasks involving high I/O operations or those that can be efficiently parallelized. However, multithreading introduces challenges such as synchronization issues, race conditions, and the potential for deadlocks, which require careful management to ensure correctness and efficiency.

Results:

1. Naive Approach:



The screenshot displays a code editor window with a file named `osNaive.c` and a terminal window below it. The code editor shows the following C code:

```
1 //Fadi Bassous
2 //1221005
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9 int size = 0;
10 int size2 = 300000;
11 typedef struct {
12     int frequency;
```

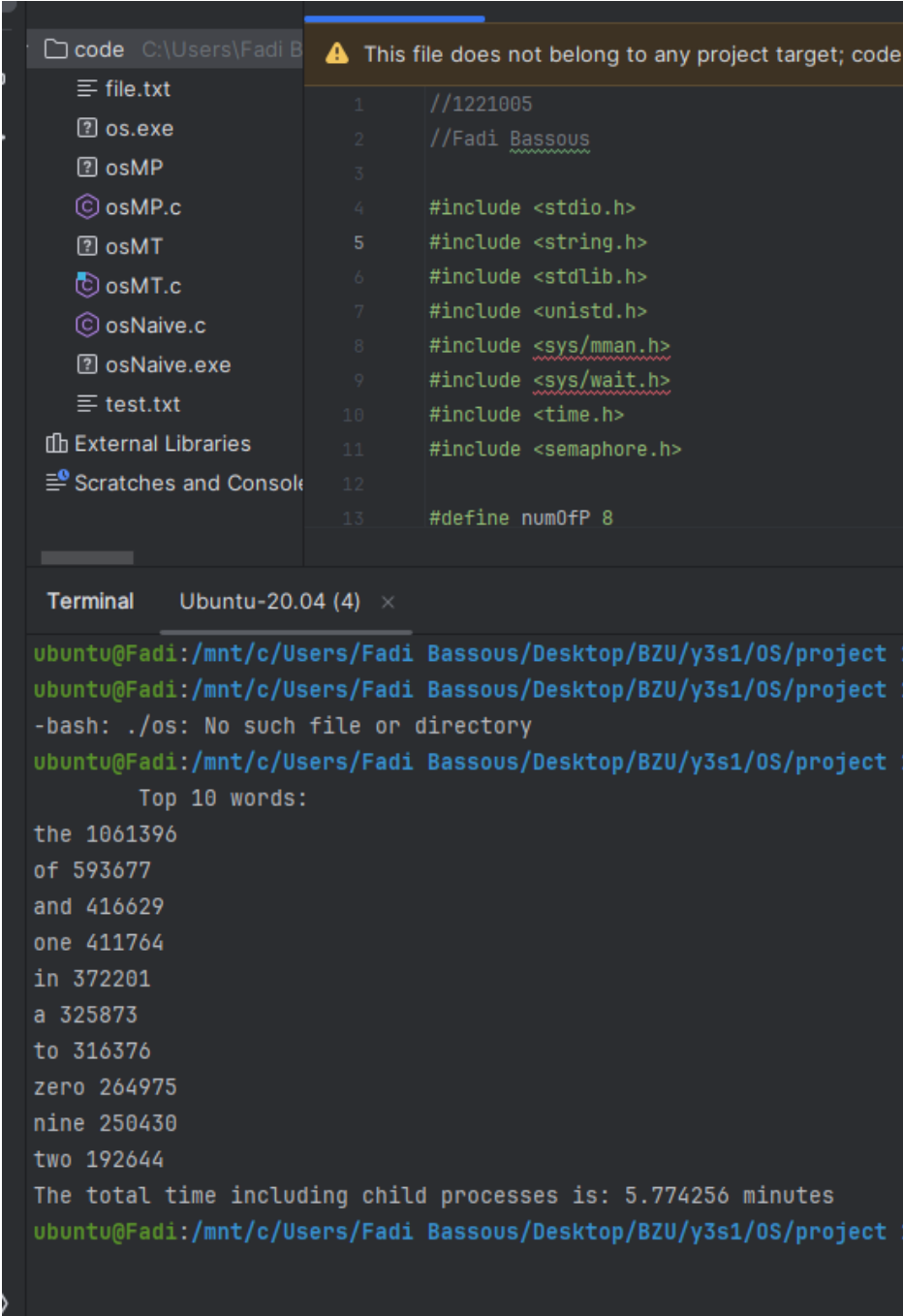
The terminal window shows the execution of the program:

```
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/code$ gcc -o os osNaive.c
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/code$ ./os
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time is: 8.918062 minutes
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/code$
```

Figure 1: Naive Result

2. Multiprocessing Approach:

8 children:



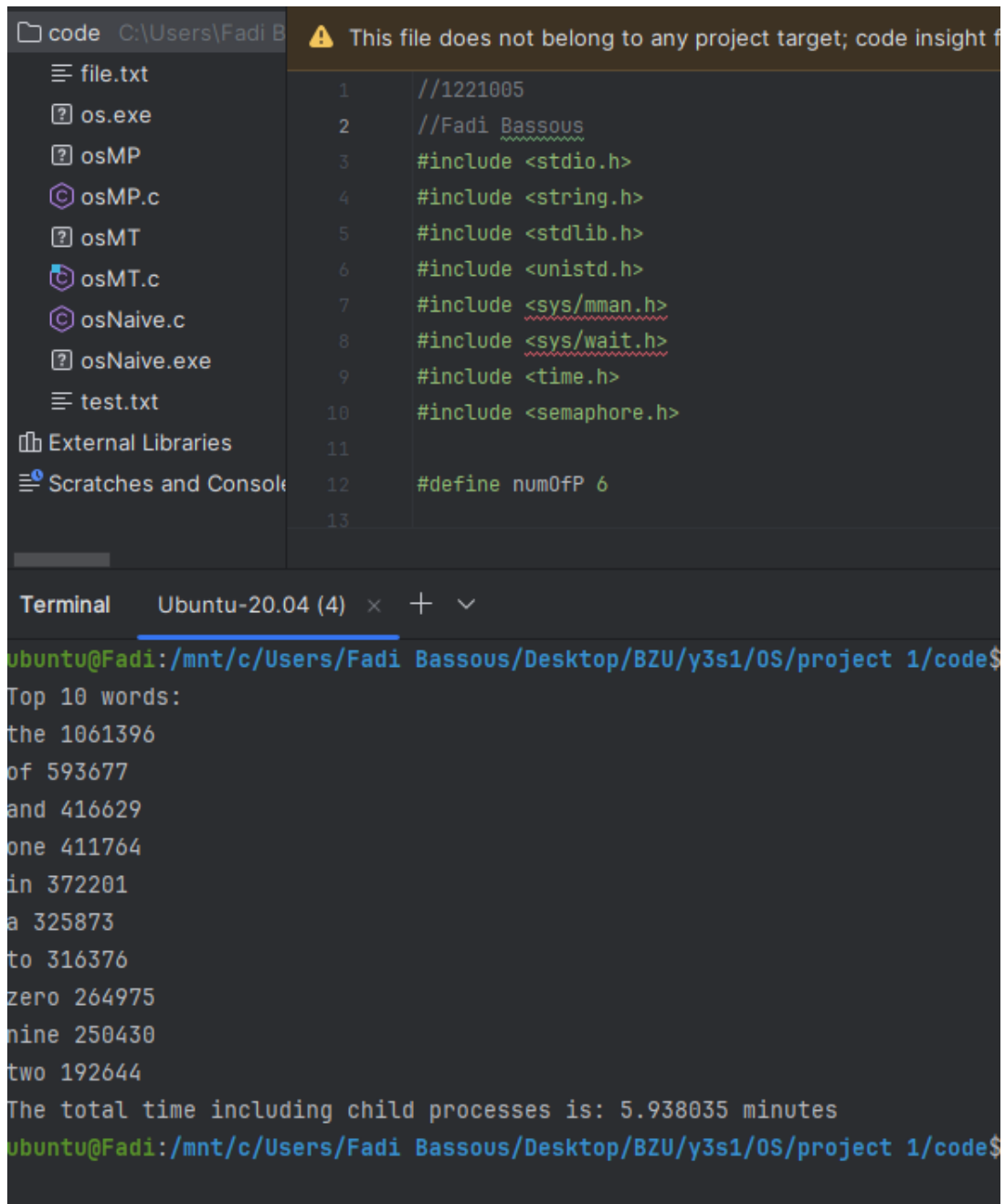
The screenshot shows a code editor with a file named `os.c` and a terminal window. The code in `os.c` includes headers for `stdio.h`, `string.h`, `stdlib.h`, `unistd.h`, `sys/mman.h`, `sys/wait.h`, and `time.h`. It defines `numOfP` as 8. The terminal shows the execution of `./os`, which outputs the top 10 words from a file and the total time including child processes.

```
1 //1221005
2 //Fadi Bassous
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include <unistd.h>
8 #include <sys/mman.h>
9 #include <sys/wait.h>
10 #include <time.h>
11 #include <semaphore.h>
12
13 #define numOfP 8
```

```
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
-bash: ./os: No such file or directory
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
    Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including child processes is: 5.774256 minutes
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
```

Figure 2: Multiprocessing result (8 children)

6 children:



The image shows a code editor window with a file explorer on the left and a terminal at the bottom. The file explorer lists files: file.txt, os.exe, osMP, osMP.c, osMT, osMT.c, osNaive.c, osNaive.exe, test.txt, External Libraries, and Scratches and Console. The code editor displays C code for a multiprocessing program. The code includes standard headers, defines the number of processes as 6, and contains a loop that prints the top 10 words from a file. The terminal shows the output of the program, including the top 10 words and the total time taken.

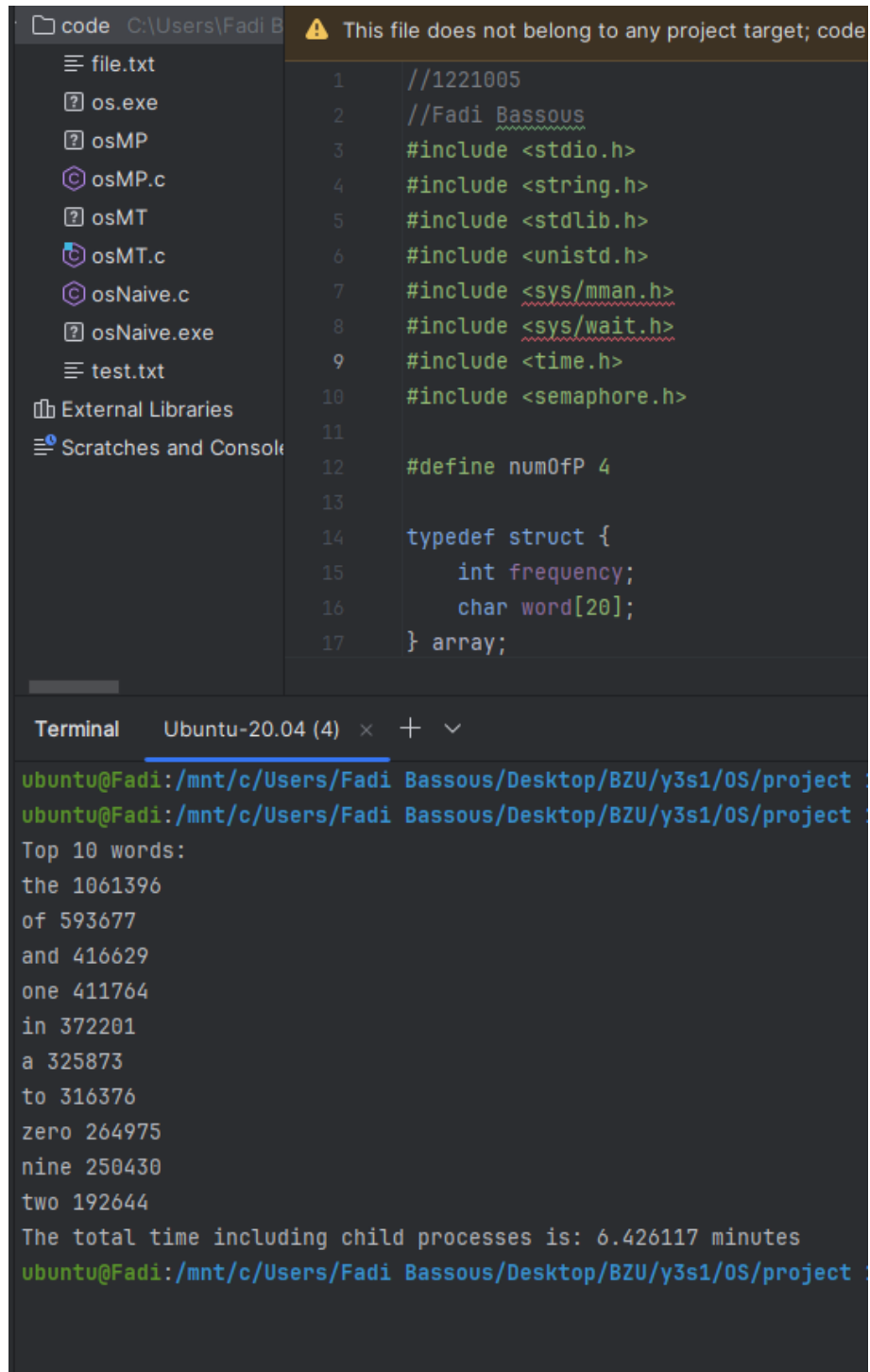
```
1 //1221005
2 //Fadi Bassous
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #include <sys/wait.h>
9 #include <time.h>
10 #include <semaphore.h>
11
12 #define numOfP 6
13
```

Terminal: Ubuntu-20.04 (4) x + v

```
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/code$
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including child processes is: 5.938035 minutes
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/code$
```

Figure 3: Multiprocessing result (6 children)

4 children:



The image shows a Visual Studio Code editor window. The left sidebar displays a file explorer with the following files: file.txt, os.exe, osMP, osMP.c, osMT, osMT.c, osNaive.c, osNaive.exe, test.txt, External Libraries, and Scratches and Console. The main editor area shows a C program with the following code:

```
1 //1221005
2 //Fadi Bassous
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #include <sys/wait.h>
9 #include <time.h>
10 #include <semaphore.h>
11
12 #define numOfP 4
13
14 typedef struct {
15     int frequency;
16     char word[20];
17 } array;
```

Below the editor is a terminal window titled "Terminal Ubuntu-20.04 (4) x + v". The terminal output shows the execution of the program:

```
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including child processes is: 6.426117 minutes
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project :
```

Figure 4: Multiprocessing result (4 children)

2 children:

The image shows a code editor with a file explorer on the left and a terminal at the bottom. The file explorer lists files: file.txt, os.exe, osMP, osMP.c, osMT, osMT.c, osNaive.c, osNaive.exe, test.txt, External Libraries, and Scratches and Console. The code editor displays C code for a program named osMP.c. The code includes standard headers, defines a semaphore, and uses pthreads for multiprocessing. The terminal shows the execution of the program, displaying the top 10 words of a file and the total execution time including child processes.

```
1 //1221005
2 //Fadi Bassous
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #include <sys/wait.h>
9 #include <time.h>
10 #include <semaphore.h>
11
12 #define numOfP 2
13
14 typedef struct {
```

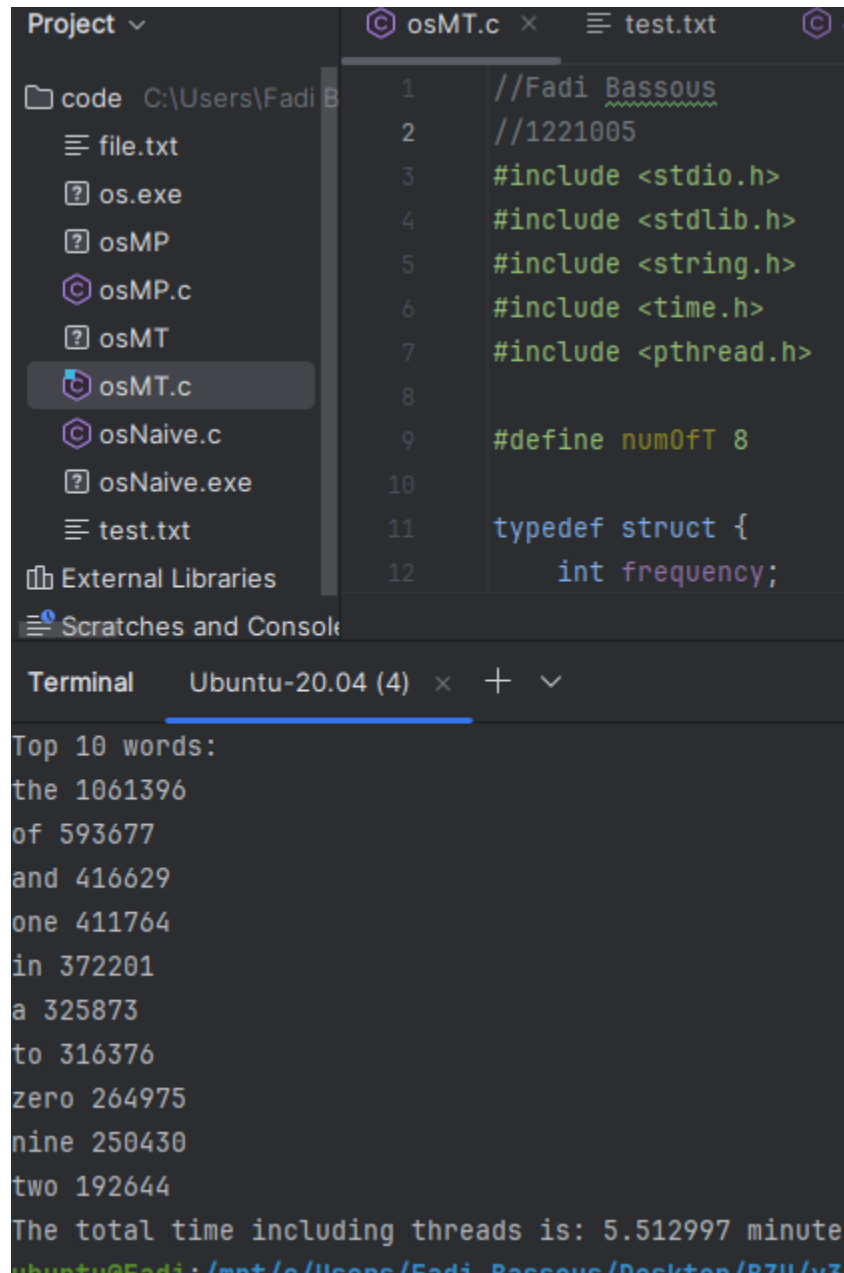
Terminal Ubuntu-20.04 (4) × + ∨

```
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/c
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/c
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including child processes is: 7.672735 minutes
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/c
```

Figure 5: Multiprocessing result (2 children)

3. Multithreading Approach:

8 threads:



```
Project ▾
  code C:\Users\Fadi B
    file.txt
    os.exe
    osMP
    osMP.c
    osMT
    osMT.c
    osNaive.c
    osNaive.exe
    test.txt
  External Libraries
  Scratches and Console

osMT.c x test.txt

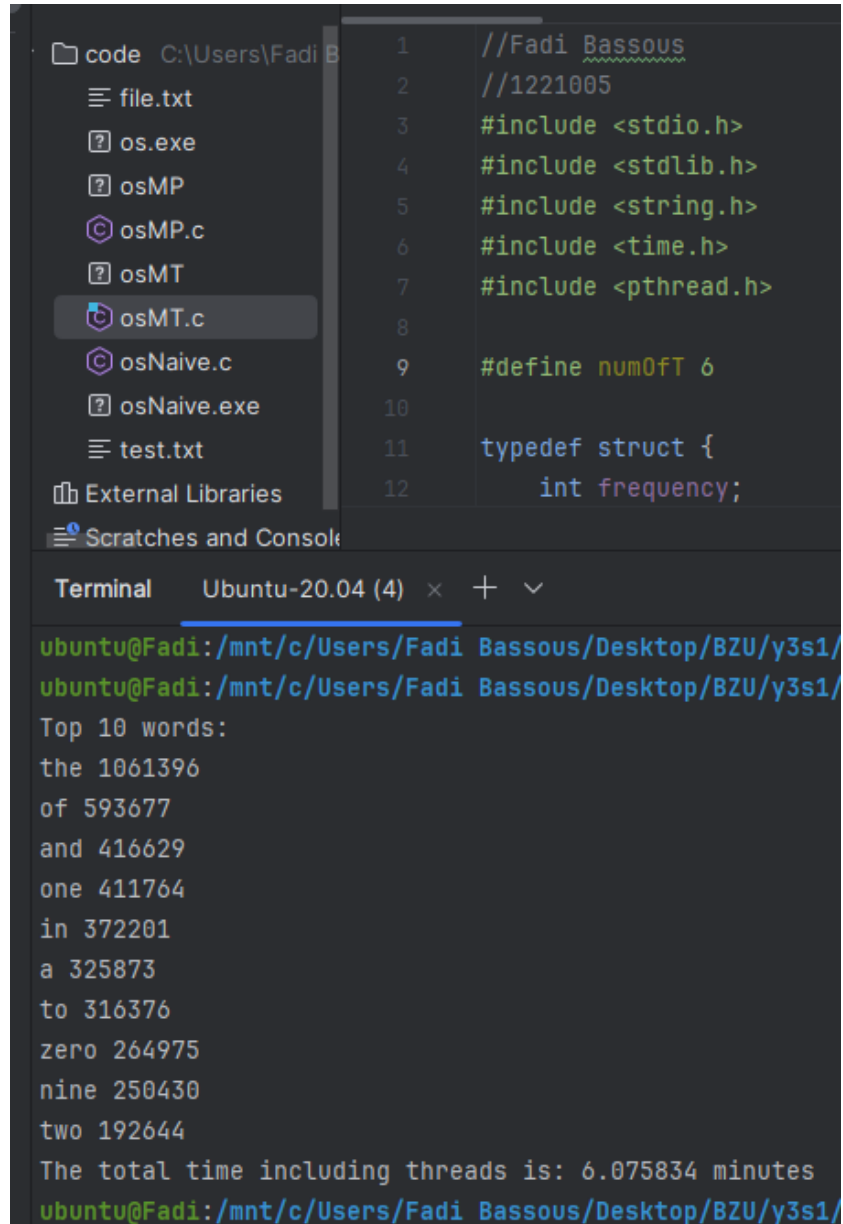
1 //Fadi Bassous
2 //1221005
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <pthread.h>
8
9 #define numOfT 8
10
11 typedef struct {
12     int frequency;
```

```
Terminal Ubuntu-20.04 (4) x + ▾

Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including threads is: 5.512997 minute
ubuntu@Fadi: /mnt/c:/Users/Fadi Bassous/Desktop/B711/h7
```

Figure 6: Multithreading result (8 threads)

6 threads:



The image shows a code editor window with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files like file.txt, os.exe, osMP, osMP.c, osMT, osMT.c, osNaive.c, osNaive.exe, test.txt, External Libraries, and Scratches and Console. The code editor shows a C program with 12 lines of code. The code includes standard headers, defines a constant numOfT as 6, and defines a struct with an int frequency. The terminal window below the code editor shows the output of the program, including the top 10 words and the total time including threads.

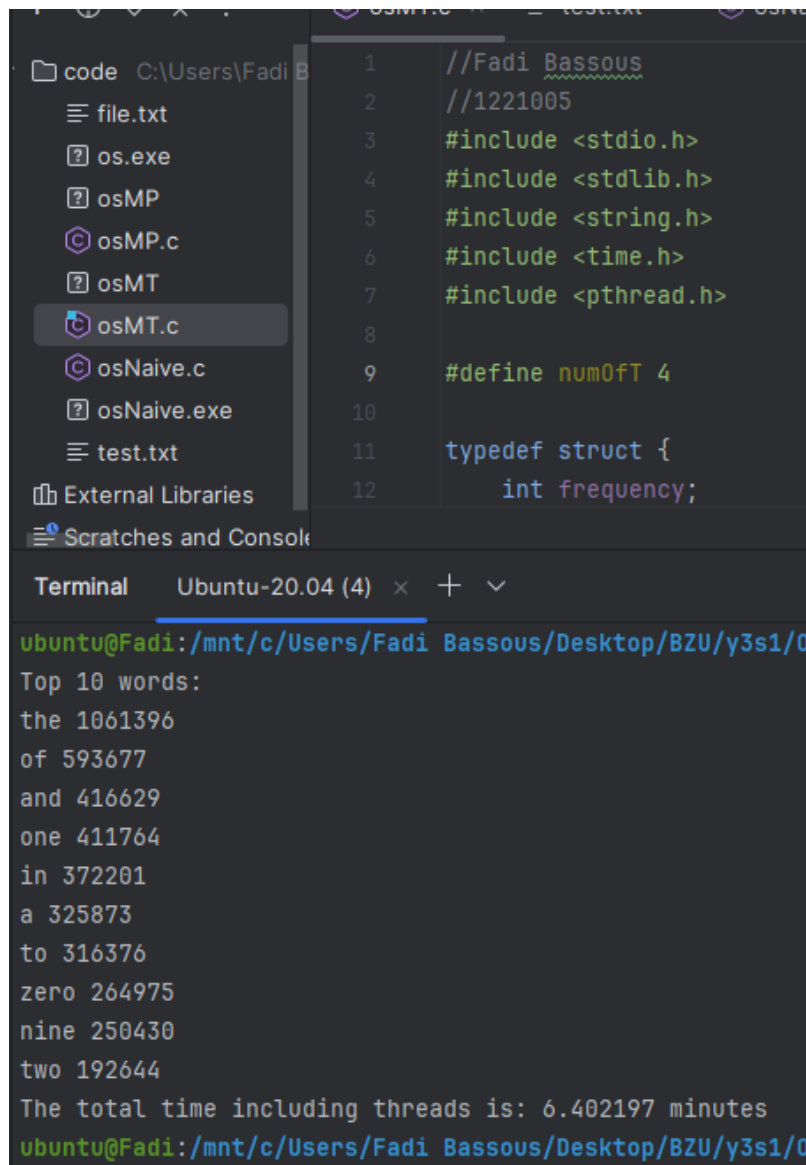
```
1 //Fadi Bassous
2 //1221005
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <pthread.h>
8
9 #define numOfT 6
10
11 typedef struct {
12     int frequency;
```

Terminal Ubuntu-20.04 (4) × + ∨

```
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including threads is: 6.075834 minutes
ubuntu@Fadi:/mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/
```

Figure 7: Multithreading result (6 threads)

4 threads:



The image shows a code editor window with a file explorer on the left and a terminal at the bottom. The file explorer lists files in a directory: file.txt, os.exe, osMP, osMP.c, osMT, osMT.c (selected), osNaive.c, osNaive.exe, test.txt, External Libraries, and Scratches and Console. The code editor displays the contents of osMT.c, which includes headers for stdio.h, stdlib.h, string.h, time.h, and pthread.h. It also defines numOfT as 4 and declares a struct with an int frequency member. The terminal shows the output of a program running on Ubuntu-20.04. It displays the top 10 words and their counts, followed by the total time including threads, which is 6.402197 minutes.

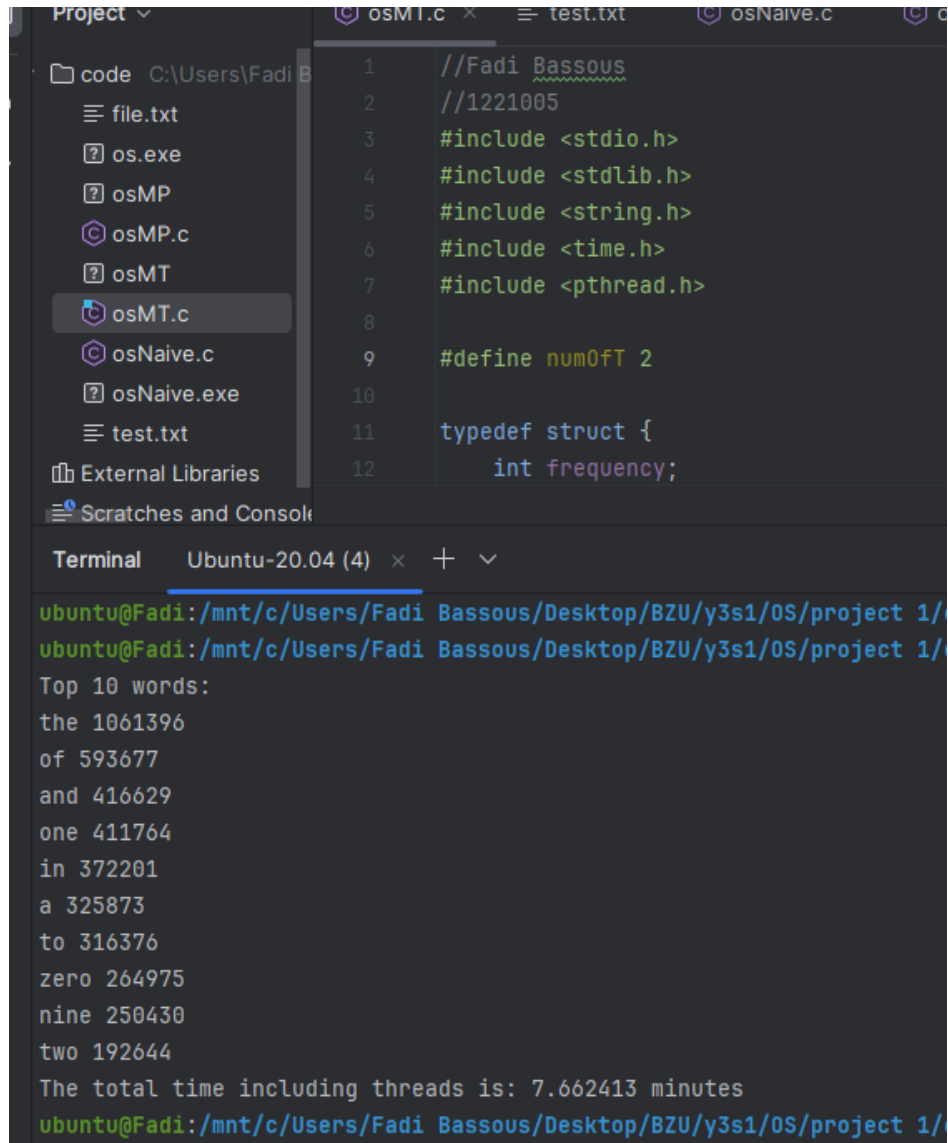
```
1 //Fadi Bassous
2 //1221005
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <pthread.h>
8
9 #define numOfT 4
10
11 typedef struct {
12     int frequency;
```

Terminal: Ubuntu-20.04 (4) x + v

```
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/0
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including threads is: 6.402197 minutes
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/0
```

Figure 8: Multithreading result (4 threads)

2 threads:



The screenshot shows a code editor with a project named 'code' in the left sidebar. The file 'osMT.c' is selected. The code in 'osMT.c' is as follows:

```
1 //Fadi Bassous
2 //1221005
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <pthread.h>
8
9 #define numOfT 2
10
11 typedef struct {
12     int frequency;
```

The terminal output at the bottom shows the execution of the program on Ubuntu-20.04 (4). The output is as follows:

```
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/
Top 10 words:
the 1061396
of 593677
and 416629
one 411764
in 372201
a 325873
to 316376
zero 264975
nine 250430
two 192644
The total time including threads is: 7.662413 minutes
ubuntu@Fadi: /mnt/c/Users/Fadi Bassous/Desktop/BZU/y3s1/OS/project 1/
```

Figure 9: Multithreading result (2 threads)

Questions:

1. Environment Description

- **Computer Specs:**
 - **Processor:** 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30 GHz
 - **Cores:** 8 physical cores, 16 threads (Hyper-threading enabled)
 - **Installed RAM:** 16.0 GB (15.7 GB usable)
 - **Storage:** Local Disk C: 475 GB (90.4 GB free)
 - **System Type:** 64-bit operating system, x64-based processor
- **Operating System:**
 - Windows 11 (configured for WSL Ubuntu Terminal usage)
- **Programming Language:** C
- **IDE Tool:** CLion
- **Virtual Machine:** No virtual machine was used; the code was executed through the Ubuntu terminal configured via WSL (Windows Subsystem for Linux).

2. How You Achieved the Multiprocessing and Multithreading Requirements: The API and Functions Used

- **Multiprocessing**

The following system calls and synchronization mechanisms were utilized to implement multiprocessing:

fork() and **waitpid():**

The fork() system call was used to create multiple child processes, with each process working independently on a portion of the dataset.

The waitpid() function ensured that the parent process waited for all child processes to complete execution before proceeding with result aggregation.

Shared Memory with **mmap()**:

The `mmap()` system call was used to create shared memory with the `MAP_SHARED` and `MAP_ANONYMOUS` flags.

This allowed all child processes to write their computed results into a common memory space, ensuring efficient data sharing.

Semaphores (sem_t):

To prevent data corruption caused by multiple child processes accessing shared memory simultaneously, a semaphore was employed.

The semaphore was initialized using `sem_init()` and protected critical sections of code where shared memory was updated.

Steps:

Divide the dataset into chunks proportional to the number of child processes (`numOfP`).

Each child process processes its allocated chunk, calculates word frequencies locally, and stores results in shared memory.

A semaphore ensures that shared memory updates are performed sequentially.

The parent process combines the results after all child processes have completed execution.

- **Multithreading**

The multithreading solution utilized lightweight threads and shared memory for efficient parallelization of computation. Key API functions and synchronization mechanisms used are:

pthread_create() and pthread_join():

Threads were created using the `pthread_create()` function, with each thread working on a designated portion of the dataset.

Threads were joined back to the main thread using `pthread_join()` upon completion of their tasks, ensuring all computations were complete before merging results.

Mutex (**pthread_mutex_t**):

A mutex was used to ensure thread-safe access to shared resources, such as the result array. This prevented race conditions and maintained data integrity during concurrent updates.

Steps:

Divide the dataset into chunks proportional to the number of threads (numOfT).

Each thread processes its chunk and computes local word frequencies.

A mutex synchronizes access to the shared result array, ensuring orderly updates.

The main thread merges results from all threads after they finish execution.

3. What percentage is the serial part of your code?

$$T_{\text{serial}} = T_{\text{total}} - T_{\text{parallel}}$$

$$T_{\text{serial}} = 8.918062 - 8.361548 = 0.556514 \text{ minutes}$$

$$S = T_{\text{serial}} / T_{\text{total}} = 0.556514 / 8.918062 \approx 0.0624 \text{ (6.24\%)}$$

Calculate the Maximum Speedup?

$$\text{Speedup} = S + N(1 - S) = 0.0624 + 8(1 - 0.0624) = 1 / 0.17961 \approx 5.57$$

this means the maximum speedup with 8 cores is approximately **5.57x**.

4. table that compares the performance of the 3 approaches:

Program / Run time(minute)	Naive	Multiprocessing	Multithreading
_____	8.91	_____	_____
2. Childe/Thread	_____	7.67	7.66
4. Childe/Thread	_____	6.42	6.40
6. Childe/Thread	_____	5.93	6.07
8. Childe/Thread	_____	5.77	5.51

Table 1: Result for all 3 approaches

5. Comment on the differences in performance:

Single-threaded approach:

The single-threaded approach is the most straightforward and, for execution time, the longest. This is because of the inherent lack of parallelism and underutilization of available system resources. It shows the important overhead introduced in transitioning to parallel methods of multiprocessing and multithreading.

Multiprocessing Approach:

Execution time improves considerably while moving from a single process to two child processes, thereby leveraging parallelism effectively. But once the number of child processes is increased beyond this point, the return diminishes. Sometimes the execution time may even stabilize or increase slightly due to the overhead of managing extra processes and inter-process communication.

Multithreading Approach:

With one thread, the execution time is a lot higher compared to the single-threaded and multiprocessing approaches. This is most likely due to the added overhead of managing threads. However, with two threads, the execution time reduces dramatically, making it competitive with the multiprocessing approach. Beyond two threads, additional threads show minimal performance gains; this could be due to synchronization overhead and contention for shared resources.

conclusion

This project demonstrates the significant performance benefits of parallel computing for large dataset analysis. Both multiprocessing and multithreading approaches outperformed the naive method. The study emphasizes the importance of balancing parallelism and overhead for optimal efficiency, guided by theoretical principles such as Amdahl's law.