Faculty of Engineering & Technology
Electrical & Computer Engineering Department

OPERATING SYSTEMS– (ENCS3390)

Project Report

Prepared by:

Aws Hammad Id : 1221697

Instructor: Dr. Yazan Abu Farha

Section: 1

Date: 28. November. 2024

Abstract

This report evaluates and compares the performance of three programming paradigms in processing large textual datasets, specifically using the enwik8 dataset to identify the top 10 most frequent words. The approaches examined include: (1) a Naive Approach, implemented as a single-threaded, sequential program that processes the dataset linearly without parallelism; (2) a Multiprocessing Approach, where the dataset is divided among multiple child processes (using 2, 4, 6, and 8 processes) to distribute the workload concurrently; and (3) a Multithreading Approach, which utilizes multiple threads (2, 4, 6, and 8 threads) within a single process to parallelize computational tasks. All approaches were implemented in C, and their performance was assessed based on execution time. The results demonstrate the advantages of parallelism, with both multiprocessing and multithreading significantly outperforming the naive approach. Additionally, the impact of varying the number of processes or threads was analyzed to determine the optimal level of parallelism for the task.

Table of contents

Table of Figures

List of Tables

Theory

1. Evaluation Environment and System Configuration:

The experiments were conducted in a virtualized environment using VirtualBox on a Windows host machine. The virtual machine was configured with Ubuntu 64-bit as the guest operating system. The VM was allocated 9 GB of memory and utilized 4 virtual processors with Nested Paging and KVM paravirtualization enabled for enhanced performance. The boot order included floppy, optical, and hard disk options. The graphics configuration featured 16 MB of video memory, a scale factor of 2.0, and a VMSVGA graphics controller. The storage configuration included a 60 GB virtual hard disk (ubuntu.vdi) connected via a SATA controller. The code was written in C, compiled, and executed using the GCC compiler within the Ubuntu environment. Visual Studio Code served as the integrated development environment (IDE) for writing and managing the code. All tests were conducted within this virtualized Linux environment to ensure controlled and reproducible conditions.

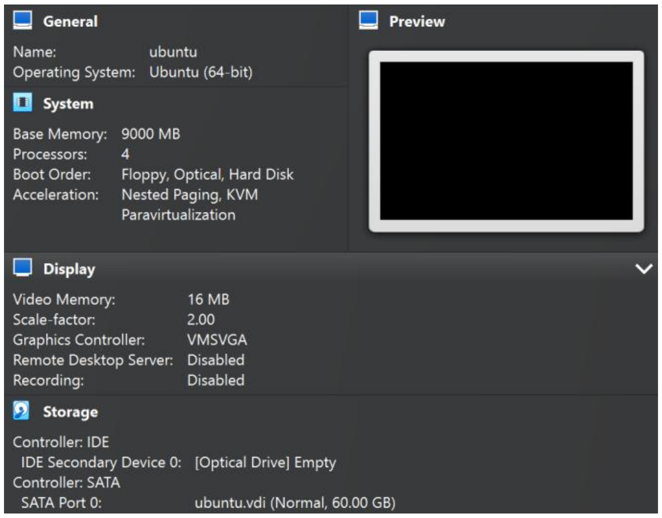

General
Name: ubuntu
Operating System: Ubuntu (64-bit)
System
Base Memory: 9000 MB
Processors: 4
Boot Order: Floppy, Optical, Hard Disk
Acceleration: Nested Paging, KVM Paravirtualization
Display
Video Memory: 16 MB
Scale-factor: 2.00
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled
Storage
Controller: IDE
IDE Secondary Device 0: [Optical Drive] Empty
Controller: SATA
SATA Port 0: ubuntu.vdi (Normal, 60.00 GB)

Figure 1: My Evaluation Environment

2. Multiprocessing:

Multiprocessing is a technique where multiple processes run independently and simultaneously to perform specific tasks. In this paradigm, a parent process creates child processes, each of which operates as an independent entity capable of executing tasks concurrently. Unlike threads, each process has its own memory space, ensuring isolation and reducing risks of data corruption. In C, multiprocessing is typically implemented using the fork() function to create child processes.

To synchronize execution, the parent process often waits for the child processes to complete their tasks using functions like wait() or waitpid(). This approach enhances performance, especially for CPU-intensive tasks, by fully utilizing multi-core processors [1].

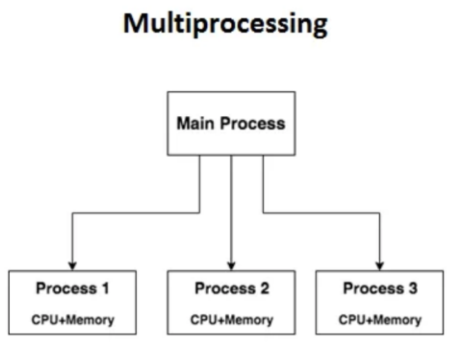


Figure 2: Multiprocessing

3. Multithreading:

Multithreading enables multiple threads to run concurrently within a single process, sharing the same memory and resources. Threads are lightweight and managed by the operating system, allowing efficient parallel execution. In C, threads are commonly created using libraries like pthreads. Multithreading is most effective on multi-core systems, where threads can execute simultaneously. On single-core systems, threads must context switch, which can introduce some overhead. This approach improves performance in tasks requiring high concurrency or parallelism [1].

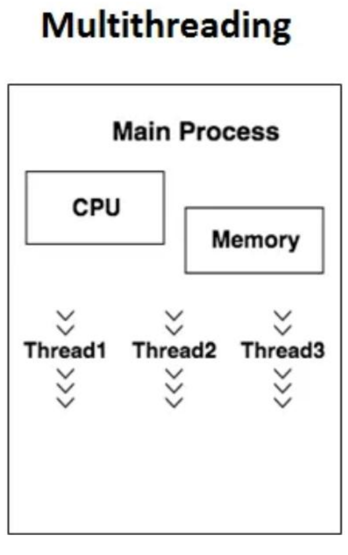


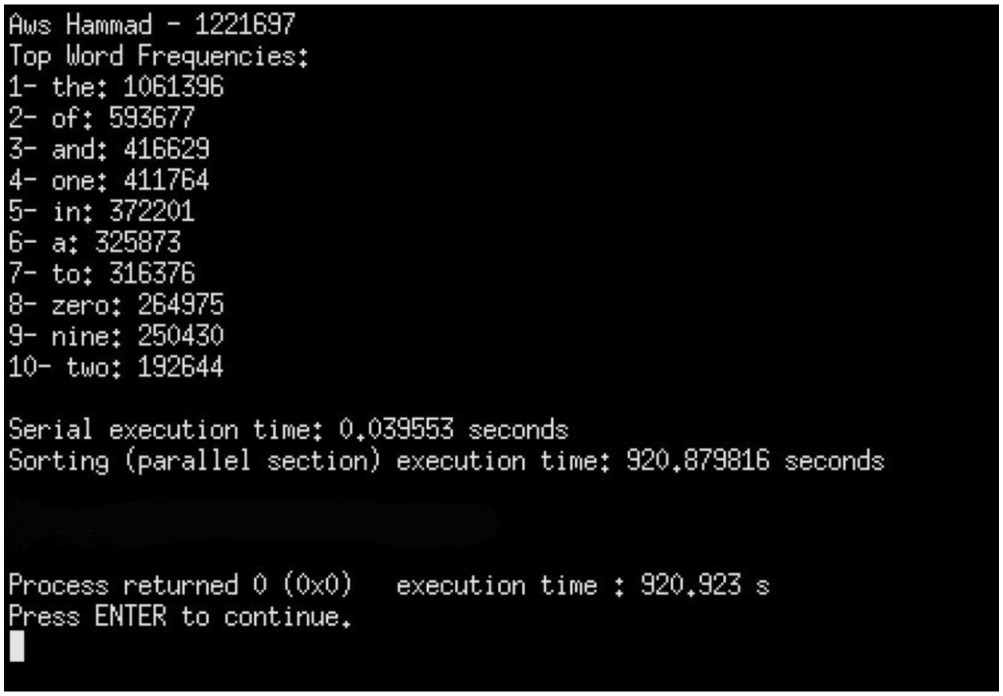
Figure 3: Multithreading

Procedure & Data Analysis

1. Naive approach:

The naive implementation served as a baseline, using a single-threaded approach to process the input file, count word frequencies, and output the results. The program reads the file line by line, tokenizing it into individual words using the **fscanf** function. A dynamic array of structures (**WordFreq**) is used to store the unique words and their respective frequencies. For each word, the program checks if it already exists in the array; if it does, the count is incremented, otherwise, the word is added to the array. This is done sequentially, and if the array exceeds its allocated size, it is resized using **realloc**. After counting the frequencies, the array is sorted in descending order based on word frequency using the **qsort** function and a custom comparison function. Finally, the program prints the top 10 most frequent words. While simple, this approach is limited by its sequential processing and can be inefficient for large datasets, as it involves frequent resizing of the array and multiple passes through the data.

➔ The Naive code is found in appendix A



```
Aws Hammad - 1221697
Top Word Frequencies:
1- the: 1061396
2- of: 593677
3- and: 416629
4- one: 411764
5- in: 372201
6- a: 325873
7- to: 316376
8- zero: 264975
9- nine: 250430
10- two: 192644

Serial execution time: 0.039553 seconds
Sorting (parallel section) execution time: 920.879816 seconds

[redacted]

Process returned 0 (0x0)   execution time : 920.923 s
Press ENTER to continue.
```

Figure 4: Naive approach output

2. Multiprocessing approach:

The multiprocessing implementation used POSIX APIs to distribute the workload of word frequency computation across multiple processes. The main steps included:

File Reading and Division: The input file was read into memory, and the text was split into chunks based on the number of processes. Each process was assigned a chunk, with proper care taken to avoid splitting words across chunks.

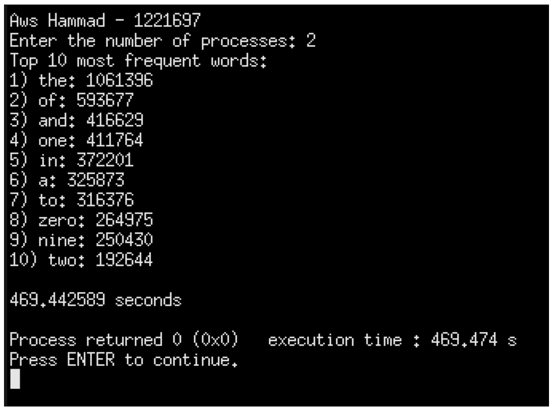
Forking Processes: The **fork()** system call was used to create multiple child processes. Each child process handled a specific portion of the text, counting word frequencies locally. It created a local array where word frequencies were stored, and after processing its chunk, each child merged the results into a shared memory space.

Shared Memory: Shared memory was allocated using **mmap** with **MAP_SHARED**, allowing all processes to write their results to the same region in memory. A semaphore was used to ensure that updates to the shared memory were synchronized and no two processes accessed it concurrently.

Result Merging: After all child processes finished execution, the parent process collected the word frequency data from the shared memory and merged it into a final list. The list was then sorted in descending order using **qsort**.

This approach efficiently utilized multiple CPU cores, which helped to significantly reduce the processing time when compared to the naive, single-threaded approach.

➔ The Multiprocessing code is found in appendix B

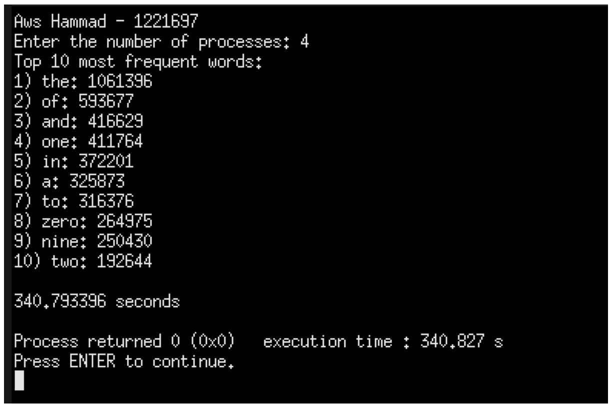


```
Aws Hammad - 1221697
Enter the number of processes: 2
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

469.442589 seconds

Process returned 0 (0x0)   execution time : 469.474 s
Press ENTER to continue.
```

Figure 5: Multiprocessing approach output (2 processes)

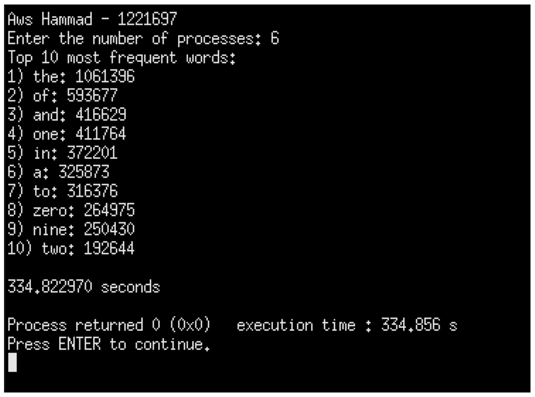


```
Aws Hammad - 1221697
Enter the number of processes: 4
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

340.793396 seconds

Process returned 0 (0x0)   execution time : 340.827 s
Press ENTER to continue.
```

Figure 6: Multiprocessing approach output (4 processes)

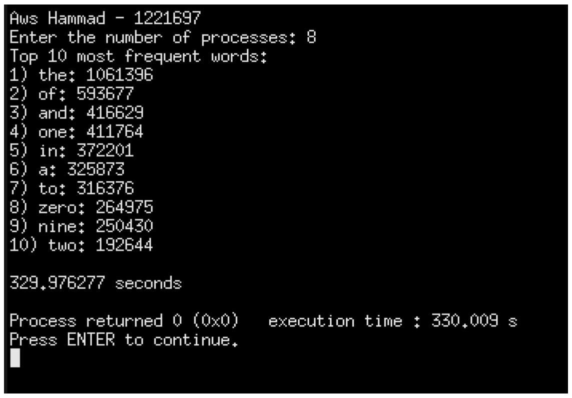


```
Aws Hammad - 1221697
Enter the number of processes: 6
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

334.822970 seconds

Process returned 0 (0x0)   execution time : 334.856 s
Press ENTER to continue.
```

Figure 7: Multiprocessing approach output (6 processes)



```
Aws Hammad - 1221697
Enter the number of processes: 8
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

329.976277 seconds

Process returned 0 (0x0)   execution time : 330.009 s
Press ENTER to continue.
```

Figure 8: Multiprocessing approach output (8 processes)

3. Multithreading approach:

The multithreading implementation used POSIX threads (pthreads) to parallelize the computation of word frequencies. The approach leveraged several key features to achieve parallelism:

Shared Data Structures: Global arrays were used to store the list of words and their frequencies. The inputWords array held the words read from the file, and the uniqueWords array stored the final results. The **uniqueWordCount** variable kept track of the number of unique words. These shared resources were accessed and modified by multiple threads, requiring synchronization to prevent race conditions.

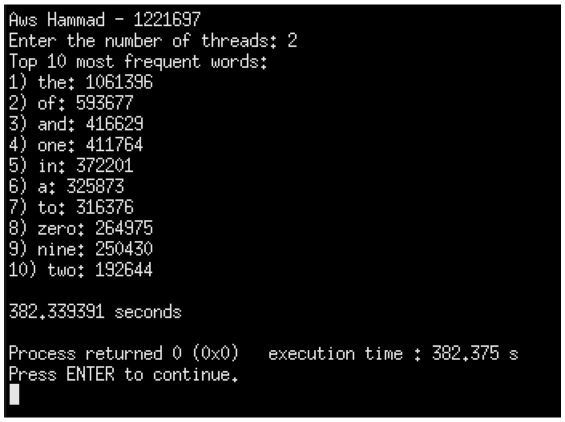
File Division and Thread Creation: The input file was divided into chunks based on the number of threads. Each thread was responsible for processing a specific range of words. The threads were created using **pthread_create**, and each thread received a **ThreadData** structure containing its assigned range of words and a pointer to the shared result array.

Chunk Processing: Each thread processed its assigned chunk of words by scanning through its range, checking for word duplicates, and updating the word frequencies. The threads used a local array to store unique words and their frequencies. After processing, they merged the results back into the shared **uniqueWords array**, updating the global word frequencies. The synchronization of these updates was implicitly managed by controlling access to the shared data structures in a thread-safe manner.

Thread Joining and Sorting: After all threads finished processing, the main thread used **pthread_join** to wait for the threads to complete. Once all threads had joined, the results were sorted using qsort to arrange the words in descending order of frequency.

This multithreading approach provided significant speedup compared to a naive sequential approach by leveraging multiple CPU cores. It had **lower overhead** compared to multiprocessing because the threads shared memory within the same process, avoiding the need for inter-process communication or shared memory allocation, making it more efficient in terms of memory usage and synchronization.

➔ The Multithreading code is found in appendix C

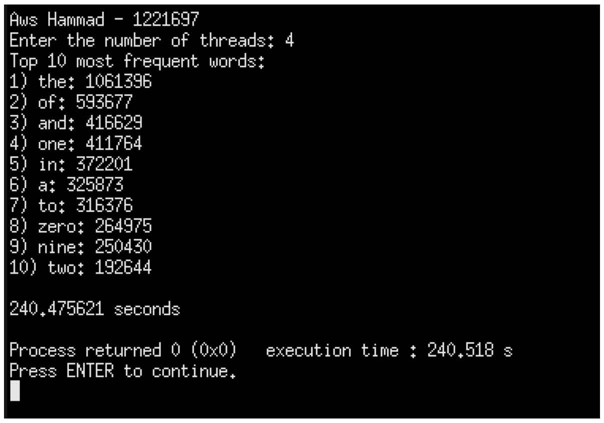


```
Aws Hammad - 1221697
Enter the number of threads: 2
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

382.339391 seconds

Process returned 0 (0x0)   execution time : 382.375 s
Press ENTER to continue.
```

Figure 9: Multithreading approach output (2 threads)

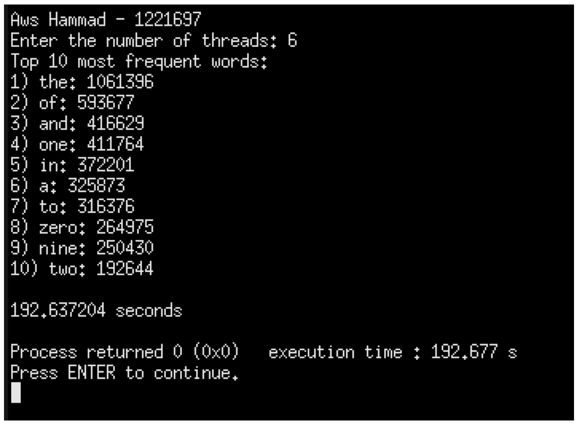


```
Aws Hammad - 1221697
Enter the number of threads: 4
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

240.475621 seconds

Process returned 0 (0x0)   execution time : 240.518 s
Press ENTER to continue.
```

Figure 10: Multithreading approach output (4 threads)

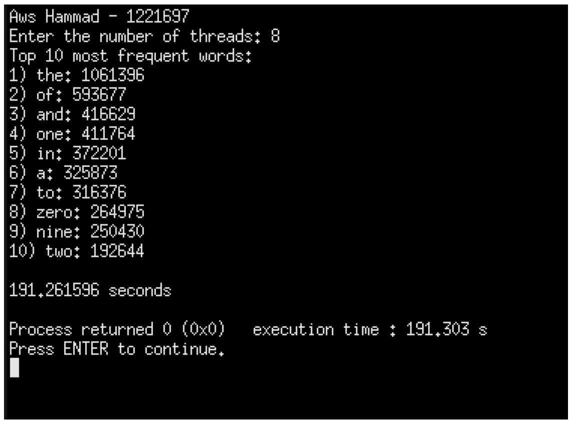


```
Aws Hammad - 1221697
Enter the number of threads: 6
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

192.637204 seconds

Process returned 0 (0x0)   execution time : 192.677 s
Press ENTER to continue.
```

Figure 11: Multithreading approach output (6 threads)



```
Aws Hammad - 1221697
Enter the number of threads: 8
Top 10 most frequent words:
1) the: 1061396
2) of: 593677
3) and: 416629
4) one: 411764
5) in: 372201
6) a: 325873
7) to: 316376
8) zero: 264975
9) nine: 250430
10) two: 192644

191.261596 seconds

Process returned 0 (0x0)   execution time : 191.303 s
Press ENTER to continue.
```

Figure 12: Multithreading approach output (8 threads)

4. Percentage of Serial Portion and Maximum Speedup:

The serial portion of the program, denoted as S, was determined by dividing the time spent on non-parallelizable tasks by the total execution time in the serial implementation. The calculated value of S was $\frac{0.03955}{920.923}$ = 0.00004, which corresponds to just 0.004%. This shows that the serial portion of the program is minimal, with almost the entire workload being parallelizable.

Using Amdahl's Law and the calculated S=0.00004 with n=4 cores, the theoretical maximum speedup Sp was computed as:

$$Sp = \frac{1}{S + \frac{(1-S)}{n}} = \frac{1}{0.00004 + \frac{(1-0.00004)}{4}} \approx 3.9995$$

This result indicates that with four cores, the program achieves a nearly perfect speedup, highlighting the efficiency and effectiveness of the parallelization approach used in this implementation.

5. Optimal Number of Child Processes or Threads:

Based on the performance results, the optimal number of child processes in the multiprocessing approach was determined to be 8, achieving the best runtime of 329.97 seconds. Similarly, the optimal number of threads for the multithreading approach was also 8, delivering the fastest runtime of 191.26 seconds. Notably, the multithreading approach consistently outperformed multiprocessing, likely because of its lower overhead in managing threads and its efficient use of shared memory.

Number of Processes or Threads	Naive Approach (in seconds)	Multiprocessing (in seconds)	Multithreading (in seconds)
1	829.63	x	x
2	x	469.44	382.33
4	x	340.79	240.47
6	x	334.82	192.63
8	x	329.97	191.26

Table 1: Comparison of the performance between the 3 approaches

The multithreading approach consistently outperformed multiprocessing across all tested configurations. For instance, when using 4 processes, the multiprocessing implementation had a runtime of 340.79 seconds, whereas 4 threads in the multithreading approach reduced the runtime to 202.22 seconds. This notable improvement can be attributed to multithreading's use of shared memory, which avoids the costly inter-process communication required in multiprocessing. Furthermore, as the number of processes or threads increased, the performance improvements started to plateau. This trend was evident in both approaches but was more pronounced in multiprocessing due to greater overhead from context switching and synchronization. While using 6 or 8 processes or threads further reduced runtimes, the gains were marginal, aligning with the constraints outlined by Amdahl’s Law.

Discussion:

The analysis highlights the significant performance boost achieved through parallelism in large-scale text-processing tasks. With a negligible serial portion $S = 0.00004$, the program effectively capitalizes on parallel execution, achieving near-ideal theoretical speedups. Multithreading consistently outperformed multiprocessing, primarily due to its reduced overhead and efficient shared memory usage. Although increasing the number of threads or processes improved execution times, the gains eventually plateaued, emphasizing the need to balance parallelism against overhead. Overall, the results demonstrate the efficiency and practicality of parallel computing in handling computationally intensive tasks.

Conclusion

This report explored the application of multiprocessing and multithreading to optimize a text-processing program by distributing its workload across multiple cores. Performance analysis, supported by Amdahl’s Law, revealed that the program’s minimal serial portion (S = 0.004%) allowed it to achieve near-ideal speedup through parallel execution. Multithreading consistently outperformed multiprocessing due to its lower overhead and efficient shared memory management. While increasing the number of threads or processes further reduced execution time, the gains diminished beyond a certain threshold, highlighting the impact of system resource constraints and communication overhead. This study underscores the importance of understanding task parallelization and selecting the most effective method to maximize efficiency, demonstrating the combined value of theoretical models like Amdahl’s Law and practical performance evaluation.

References

[1] https://hiteshmishra708.medium.com/multiprocessing-in-python-c6735fa70f3f **[Accessed 6-12-2024]**

Appendices:

1. Appendix A:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <sys/time.h>

#include <time.h>


#define MAX_WORD_LENGTH 100


typedef struct {

   char word[MAX_WORD_LENGTH];// to have the word in it

   int count;// for the word freq

} WordFreq;


void read_from_file(WordFreq **wordFreqArray, int *wordCount, int *arraySize) {

   FILE *in = fopen("text8.txt", "r");

   if (in == NULL) {

      perror("Error opening file");

      return;

   }

   char word[MAX_WORD_LENGTH];// buffer to have words in it

   while (fscanf(in, "%99s", word) != EOF) {// read words one by one

      int found = 0;

      for (int i = 0; i < *wordCount; i++) {// check if the word already exists in the array

         if (strcmp((*wordFreqArray)[i].word, word) == 0) {// if found

            (*wordFreqArray)[i].count++;// increase its freq
```

```
                found = 1;
                break;
            }
        }
        if (!found) {// if not found
            if (*wordCount >= *arraySize) {// resize if needed
                *arraySize *= 2;// double the size
                *wordFreqArray = realloc(*wordFreqArray, (*arraySize) * sizeof(WordFreq));// reallocation
                if (!*wordFreqArray) {
                    perror("Memory reallocation failed");
                    fclose(in);
                    return;
                }
            }
            strcpy((*wordFreqArray)[*wordCount].word, word);// add the word
            (*wordFreqArray)[*wordCount].count = 1;// put its freq to 1
            (*wordCount)++;// increase wordCount by 1
        }
    }
    fclose(in);// close the file
}

// comparison function for qsort
int compare(const void *a, const void *b) {
    WordFreq *freqA = (WordFreq *)a;
    WordFreq *freqB = (WordFreq *)b;
    return freqB->count - freqA->count;// descending order by frequency
}
```

```
int main() {
    struct timeval startSerial, endSerial, startParallel, endParallel;
    int initialSize = 10;
    WordFreq *wordFreqArray = malloc(initialSize * sizeof(WordFreq));
    if (!wordFreqArray) {
        perror("Memory allocation failed");
        return EXIT_FAILURE;
    }
    int wordCount = 0;
    int arraySize = initialSize;
    gettimeofday(&startParallel, NULL);// start parallel time
    read_from_file(&wordFreqArray, &wordCount, &arraySize);// read words and populate the WordFreq array
    gettimeofday(&endParallel, NULL);// end parallel time
    gettimeofday(&startSerial, NULL);// start serial time
    qsort(wordFreqArray, wordCount, sizeof(WordFreq), compare);// sort the array by frequency
    gettimeofday(&endSerial, NULL);// end serial time
    // print the top 10 words
    printf("Aws Hammad - 1221697\nTop Word Frequencies:\n");
    int printCount = wordCount < 10 ? wordCount : 10;// if the words less than 10 make printCount equals them
else equals 10
    for (int i = 0; i < printCount; i++) {
        printf("%d- %s: %d\n", (i+1) ,wordFreqArray[i].word, wordFreqArray[i].count);
    }
    double serialTime = (endSerial.tv_sec - startSerial.tv_sec) + (endSerial.tv_usec - startSerial.tv_usec) /
1000000.0;
    double parallelTime = (endParallel.tv_sec - startParallel.tv_sec) + (endParallel.tv_usec - startParallel.tv_usec) /
1000000.0;
    printf("\nSerial execution time: %.6f seconds\n", serialTime);
    printf("Sorting (parallel section) execution time: %.6f seconds\n\n", parallelTime);
    free(wordFreqArray);// free allocated memory
    return 0;

}
```

2. Appendix B:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/mman.h>

#include <semaphore.h>

#include <fcntl.h>

#include <sys/time.h>


struct Word {

    char word[50];// word with max length 50

    int word_freq;// frequency of the word

};


// function declarations

int read_from_file(struct Word **Array);// read words from a file

int frequency(int wordCount, struct Word *Array, struct Word **Array2, int num);// calculate word frequencies
using multiprocessing

void sort(int wordCount, struct Word *Array2);// sorts the words by frequency using qsort


int main() {

    struct timeval end, start;// for measuring the time

    double totalTime;// to store total time

    // start measuring time
    gettimeofday(&start, NULL);
    int numOfProcesses;// number of processes
    printf("Aws Hammad - 1221697 \nEnter the number of processes: ");
    scanf("%d", &numOfProcesses);
    struct Word *arr = NULL;// to store words read from the file
    struct Word *arr3 = NULL;// shared array to store the results that came out from the child processes
    // read words from the file
    int wordCount = read_from_file(&arr);
    if (wordCount == 0) {
       // if there is no words exit the program
       printf("No words read from the file. Exiting...\n");
       return 1;
    }
    // calculate word frequencies with multiprocessing
    int lastCount = frequency(wordCount, arr, &arr3, numOfProcesses);
    // sorting the words on frequency
    sort(lastCount, arr3);
    // display the top 10 words
    printf("Top 10 most frequent words:\n");
    for (int j = 0; j < 10 && j < lastCount; j++) {
       printf("%d) %s: %d\n", j + 1, arr3[j].word, arr3[j].word_freq);
    }
    // end time
    gettimeofday(&end, NULL);
    // total time in ((seconds))
    totalTime = (end.tv_sec - start.tv_sec) * 1000000.0;
    totalTime += (end.tv_usec - start.tv_usec);
    printf("\n%f seconds\n", totalTime / 1000000.0);
    return 0;
```

```
}


// read words from the file into an array
int read_from_file(struct Word **arr) {
    // open the file
    FILE *file = fopen("text8.txt", "r");
    if (file == NULL) {// the file didn't open
        perror("Failed to open the file");
        return 0;
    }
    int capacity = 1000000;// initial capacity of the array
    int c = 0;// counter for the number of words
    // allocate memory for the array of structs
    *arr = (struct Word *)malloc(capacity * sizeof(struct Word));
    if (*arr == NULL) {
        perror("Memory allocation failed");
        fclose(file);
        return 0;
    }
    // read words from the file word word
    while (fscanf(file, "%s", (*arr)[c].word) == 1) {
        (*arr)[c].word_freq = 0; // initialize the frequency to 0
        c++;
        // if the array is full double its capacity
        if (c >= capacity) {
            capacity *= 2;
            *arr = (struct Word *)realloc(*arr, capacity * sizeof(struct Word));
            if (*arr == NULL) {
                perror("Memory reallocation failed");
                fclose(file);
                return 0;
            }
        }
    }
    fclose(file);// close the file
    return c;// return the number of words read
}

// calculates word frequencies using multiple processes and shared memory
int frequency(int wordCount, struct Word *Array, struct Word **Array2, int num) {
    int capacity = 10000000;// the capacity of the shared array
    int pid[num];// an array to store processes IDs
    int chunk_size = wordCount / num;// the number of words per process
    int mod = wordCount % num;// the remaining words to be handled by the last process
    // unlink any existing semaphore
    sem_unlink("/mysemaphore");
    // initialize a semaphore
    sem_t *sem = sem_open("/mysemaphore", O_CREAT, 0644, 1);
    if (sem == SEM_FAILED) {
        perror("Semaphore initialization failed");
        exit(1);
    }
    // allocate shared memory for results array
    *Array2 = mmap(NULL, capacity * sizeof(struct Word), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    if (*Array2 == MAP_FAILED) {
        perror("Shared memory allocation failed for Array2");
        sem_close(sem);
        sem_unlink("/mysemaphore");
        exit(1);
    }
    // allocate shared memory for the count of last words
    int *lastCount = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    if (lastCount == MAP_FAILED) {
        perror("Shared memory allocation failed");
        sem_close(sem);
        sem_unlink("/mysemaphore");
        exit(1);
    }
    *lastCount = 0;// initialize the shared counter to 0
    // create child processes
    for (int r = 0; r < num; r++) {
        pid[r] = fork();// fork a new process
        if (pid[r] == 0) {
            // child process code
            int local_size = chunk_size + (r == num - 1 ? mod : 0);// the remaining words in the last process
            struct Word *local_array = (struct Word *)malloc(local_size * sizeof(struct Word));
            if (local_array == NULL) {
                perror("Memory allocation failed in child");
                exit(1);
            }
            int startIndex = r * chunk_size;// the starting index for this process
            int endIndex = startIndex + local_size;// the ending index for this process
            int localC = 0;// local counter for last words
            // loop through the words
            for (int z = startIndex; z < endIndex; z++) {
```

```
        int duplicate = 0;// to check if the word is already counted
        for (int y = 0; y < localC; y++) {
            if (strcmp(local_array[y].word, Array[z].word) == 0) {
                // if word is found increment its frequency
                local_array[y].word_freq++;
                duplicate = 1;
                break;
            }
        }
        if (!duplicate) {
            // if word is not found add it to the array
            strcpy(local_array[localC].word, Array[z].word);
            local_array[localC].word_freq = 1;
            localC++;
        }
    }
    // merge local results into the shared memory
    sem_wait(sem); // lock the semaphore
    for (int i = 0; i < localC; i++) {
        int found = 0;
        for (int j = 0; j < *lastCount; j++) {
            if (strcmp((*Array2)[j].word, local_array[i].word) == 0) {
                // if word already exists in the shared array update its frequency
                (*Array2)[j].word_freq += local_array[i].word_freq;
                found = 1;
                break;
            }
        }
        if (!found) {
                // if word is new add it to the shared array
                if (*lastCount < capacity) {
                    strcpy((*Array2)[*lastCount].word, local_array[i].word);
                    (*Array2)[*lastCount].word_freq = local_array[i].word_freq;
                    (*lastCount)++;
                }
              }
            }
            sem_post(sem); // unlock the semaphore
            // free local memory and exit the child process
            free(local_array);
            exit(0);
        }
    }
    // wait for all child processes to finish
    for (int r = 0; r < num; r++) {
        wait(NULL);
    }
    // clean up the semaphore and the shared memory
    sem_close(sem);
    sem_unlink("/mysemaphore");
    return *lastCount; // Return the total number of unique words
}

// comparator function for qsort
int compareWords(const void *a, const void *b) {
    struct Word *wordA = (struct Word *)a;
    struct Word *wordB = (struct Word *)b;
    return wordB->word_freq - wordA->word_freq; // compare frequencies
```

```
}

// sorts the words in descending order of frequency
void sort(int wordCount, struct Word *Array2) {
    qsort(Array2, wordCount, sizeof(struct Word), compareWords);
}
```

3. Appendix C:

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

struct Word {
    char word[50];// word with a max length 50
    int word_freq;// the frequency of the word
};

struct Word *input_words = NULL;// array that have the words from the file
struct Word *last_words = NULL;// array to store the unique words
int total_count = 0;// the number of words that have been read from the file
int unique_count = 0;// the number of unique words

// function declarations
int read_from_file(struct Word **words);// to read words from the file
int calculate_Freq(int totalWords, struct Word *words, struct Word **result, int threadCount);// to calculate word frequency using threads
void sort(int wordCount, struct Word *words);// to sort words by frequency
void *thread_freq(void *data);// to calculate frequency in a thread


typedef struct {
    int start;// the starting index for the thread
    int end;// the ending index for the thread
    struct Word *result;// array to store the results for the thread
} ThreadData;


int main() {
    struct timeval start, end;// to measure execution time
    double total_time;// to store total time
    gettimeofday(&start, NULL);// start measuring time
    int threadNum; // number of threads
    printf("Aws Hammad - 1221697\nEnter the number of threads: ");
    scanf("%d", &threadNum);
    total_count = read_from_file(&input_words);// read words from the file
    unique_count = calculate_Freq(total_count, input_words, &last_words, threadNum);// calculate frequency of
each word
    sort(unique_count, last_words);// sort the words by frequency
    printf("Top 10 most frequent words:\n");// display the top 10 words
    for (int i = 0; i < 10 && i < unique_count; i++) {
        printf("%d) %s: %d\n", i + 1, last_words[i].word, last_words[i].word_freq);
    }
    gettimeofday(&end, NULL);// stop measuring time
    // calculate elapsed time in seconds
    total_time = (end.tv_sec - start.tv_sec) * 1000000.0;
    total_time += (end.tv_usec - start.tv_usec);
    printf("\n%f seconds\n", total_time / 1000000.0);
    return 0;
```

```
}

// to read words from the file into an array
int read_from_file(struct Word **words) {
    FILE *file = fopen("/home/aws-hammad/Downloads/text8.txt", "r");// open the file
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 0;
    }
    int initialSize = 1000000;// initial size of the array
    int count = 0;// counter for the number of words
    *words = (struct Word *)malloc(initialSize * sizeof(struct Word));// allocate memory for the array
    while (fscanf(file, "%s", (*words)[count].word) == 1) {// read words from the file word by word
        (*words)[count].word_freq = 0;// initialize the frequency to 0
        count++;
        if (count >= initialSize) {// increase the array size if the count is more than the size
            initialSize *= 2;
            *words = (struct Word *)realloc(*words, initialSize * sizeof(struct Word));
        }
    }
    fclose(file);// close the file
    return count;// return the total number of words read
}

// function to calculate word frequencies using multiple threads
int calculate_Freq(int totalWords, struct Word *words, struct Word **result, int threadCount) {
    int size = totalWords / threadCount;// size of each chunk of words
    int remainder = totalWords % threadCount;// remainder words to be handled by the last thread
    *result = (struct Word *)malloc(totalWords * sizeof(struct Word));// allocate memory for the result array
    pthread_t threads[threadCount];// array to store thread IDs

    ThreadData threadData[threadCount];// array to store thread-specific data

    for (int i = 0; i < threadCount; i++) {// create threads

        threadData[i].start = i * size;// set start index

        threadData[i].end = threadData[i].start + size;// set end index

        if (i == threadCount - 1)// add remainder to the last thread

            threadData[i].end += remainder;

        threadData[i].result = *result;

        pthread_create(&threads[i], NULL, thread_freq, &threadData[i]);// create thread

    }

    for (int i = 0; i < threadCount; i++) {// wait for all threads to complete

        pthread_join(threads[i], NULL);

    }

    return unique_count;// return the total number of unique words

}


// function to calculate frequency of words in a thread

void *thread_freq(void *data) {

    ThreadData *threadData = data;// get thread-specific data

    struct Word *local_array = (struct Word *)malloc((threadData->end - threadData->start) * sizeof(struct
Word));// local array to store unique words

    int local_count = 0;// count of unique words in this thread

    for (int i = threadData->start; i < threadData->end; i++) {// calculate frequency of words in the assigned range

        int exists = 0;// flag to check if the word already exists in the local array

        for (int j = 0; j < local_count; j++) {

            if (strcmp(local_array[j].word, input_words[i].word) == 0) {

                local_array[j].word_freq++;// increment frequency if word exists

                exists = 1;

                break;
            }
        }
        if (!exists) {
            strcpy(local_array[local_count].word, input_words[i].word);// copy the the word to the local array
            local_array[local_count].word_freq = 1;// put its freq 1
            local_count++;// increase the counter by 1
        }
    }
    for (int i = 0; i < local_count; i++) {// merge local results into the shared result array
        int found = 0;// to check if the word is already exists in the shared array
        for (int j = 0; j < unique_count; j++) {
            if (strcmp(threadData->result[j].word, local_array[i].word) == 0) {// if found
                threadData->result[j].word_freq += local_array[i].word_freq;// update its frequency
                found = 1;// its found
                break;
            }
        }
        if (!found) {
            strcpy(threadData->result[unique_count].word, local_array[i].word);// copy word to the shared array
            threadData->result[unique_count].word_freq = local_array[i].word_freq;
            unique_count++;//increase its freq by 1
        }
    }
    free(local_array); // free local array memory
    pthread_exit(0);
}

// function to compare for qsort
int compareWords(const void *a, const void *b) {
    struct Word *wordA = (struct Word *)a;
    struct Word *wordB = (struct Word *)b;
    return wordB->word_freq - wordA->word_freq; // descending order by frequency
}

// function to sort words by frequency
void sort(int wordCount, struct Word *words) {
    qsort(words, wordCount, sizeof(struct Word), compareWords);
}
```