Faculty of Engineering & Technology Electrical &

Computer Engineering Department

Operating system –Project Repor

Instructor: Mohammad Khalil

Student: Aya Hamayel

ID: 1221469

Contents

Introduction

In this report, we explore the problem of identifying the **top 10 most frequent words in the **enwik8 dataset**

The purpose of this report is to compare three different approaches to solve this problem, each using a different method of parallelization:

1. **Naive Approach:** A simple, single-threaded method that processes the dataset sequentially without any parallelism.
2. **Multiprocessing Approach:** A method that splits the work across multiple child processes running in parallel. We will test this approach with 2, 4, 6, and 8 processes to see how it scales.
3. **Multithreading Approach:** A method that uses multiple threads to perform the task in parallel. Similar to the multiprocessing approach, we will test it with 2, 4, 6, and 8 threads.

We will measure the time it takes to complete the task for each approach, compare the results, and analyze the impact of parallelism. Additionally, we will apply **Amdahl's Law** to estimate the maximum possible speedup and determine the best number of processes or threads. The results of each approach will be summarized in a table, followed by a discussion of the differences in execution time and performance.

This report will show how parallel computing techniques like multiprocessing and multithreading can improve the efficiency of large-scale text processing tasks.

Niva approach

Main Structure of the Word Count Program

The program is designed to read a large text file, count word occurrences, and display the **Top 10 most frequent words**, while also measuring the total execution time. Here's a breakdown of its main structure:

---

1. Initialization:

- The program starts by initializing variables, including an array to store words and their counts, a counter for total words, and timing structures for performance measurement.

---

2. Reading the File:

- The file is opened and read word by word.
- Each word is processed by removing punctuation and converting it to lowercase.

---

3. Counting Words:

- For every word read, the program checks if it already exists in the list.
- If found, its count is incremented.
- If not, the word is added to the list with a count of 1.

---

4. Sorting Words:

- After reading the file, the program sorts the words in descending order based on their frequency using **Merge Sort**, ensuring efficiency even with a large dataset.

---

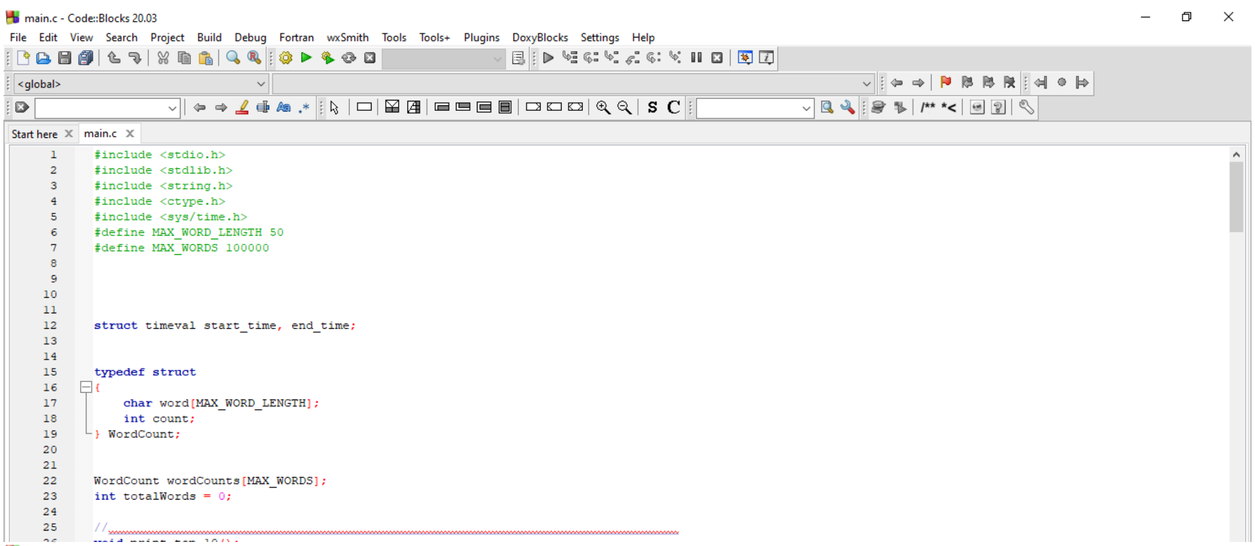
5. Displaying Results:

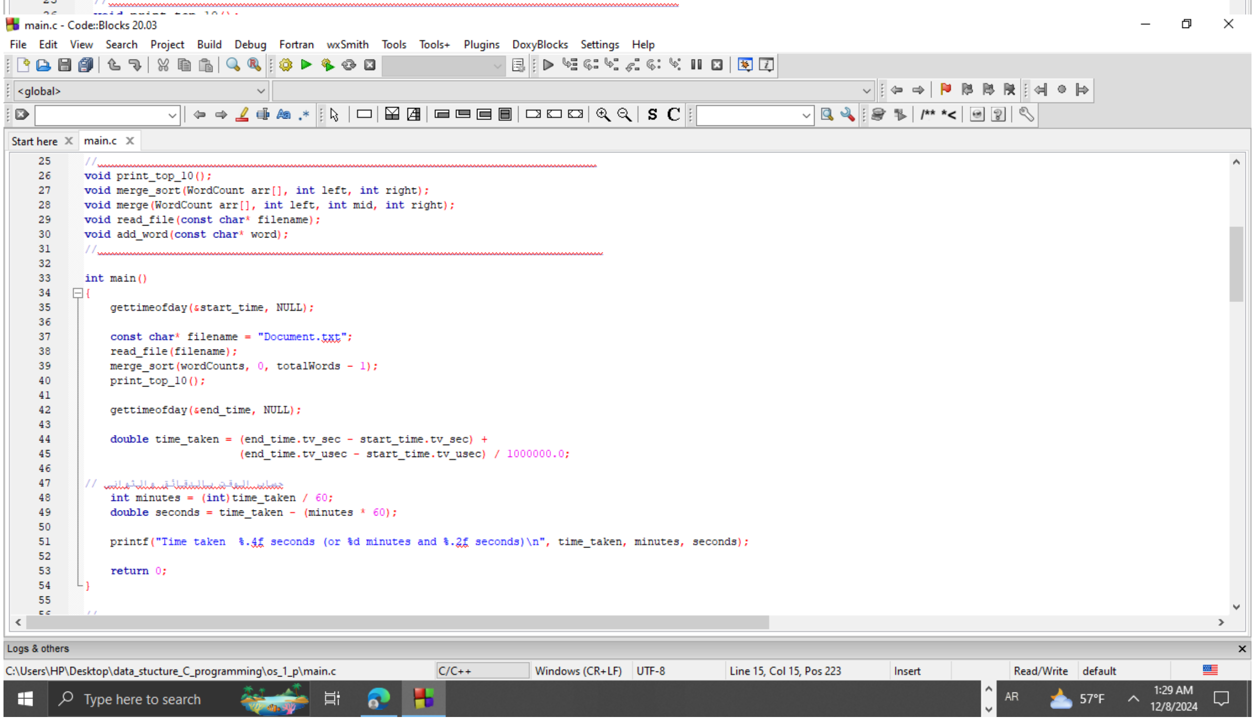
- The top 10 most frequent words are printed.
- Execution time is calculated and displayed in minutes and seconds for performance evaluation.

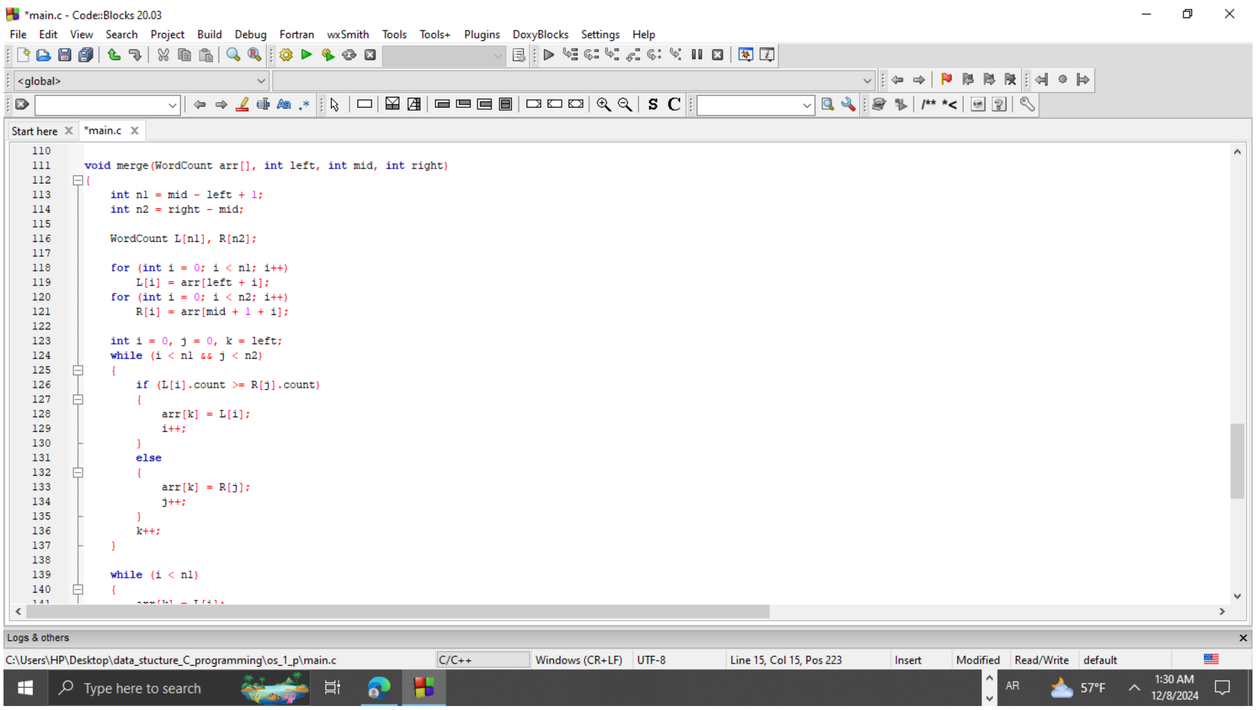
---

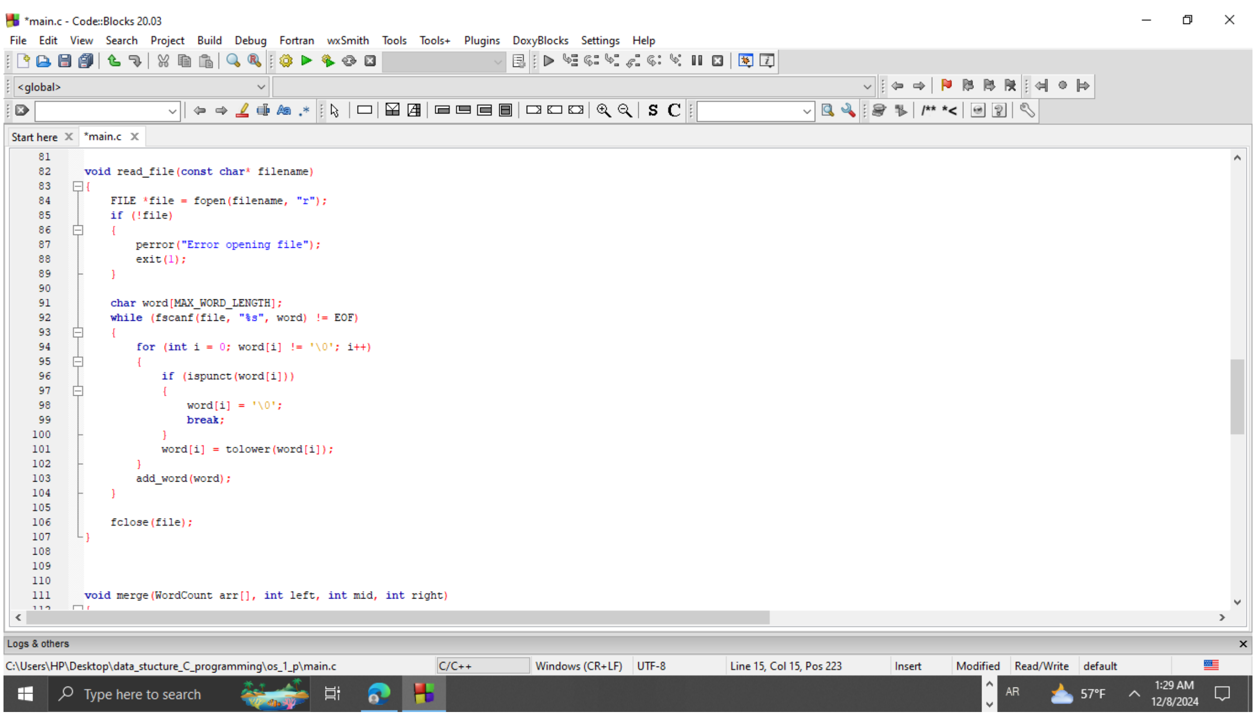
6. Conclusion:

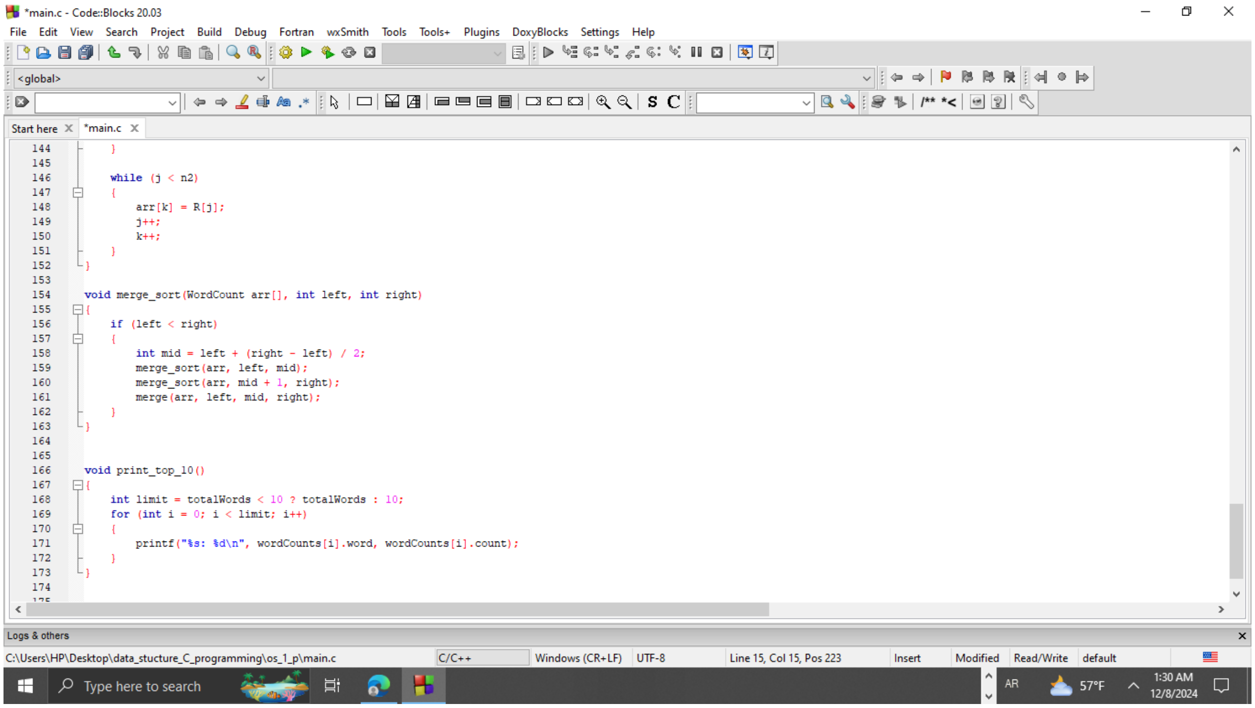
- The program efficiently handles a large file by maintaining an organized structure.
- The use of sorting ensures the correct display of the top words.
- Timing measurement highlights performance, making the program suitable for large-scale text analysis tasks.






*main.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
Start here
*main.c
void merge(WordCount arr[], int left, int mid, int right)
{
int n1 = mid - left + 1;
int n2 = right - mid;
WordCount L[n1], R[n2];
for (int i = 0; i < n1; i++)
L[i] = arr[left + i];
for (int i = 0; i < n2; i++)
R[i] = arr[mid + 1 + i];
int i = 0, j = 0, k = left;
while (i < n1 && j < n2)
{
if (L[i].count >= R[j].count)
{
arr[k] = L[i];
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
while (i < n1)
{
Logs & others
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\main.c
C/C++
Windows (CR+LF)
UTF-8
Line 15, Col 15, Pos 223
Insert
Modified
Read/Write
default
Type here to search
AR
57°F
1:30 AM
12/8/2024


*main.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
Start here
*main.c
void read_file(const char* filename)
{
FILE *file = fopen(filename, "r");
if (!file)
{
perror("Error opening file");
exit(1);
}
char word[MAX_WORD_LENGTH];
while (fscanf(file, "%s", word) != EOF)
{
for (int i = 0; word[i] != '\0'; i++)
{
if (ispunct(word[i]))
{
word[i] = '\0';
break;
}
word[i] = tolower(word[i]);
}
add_word(word);
}
fclose(file);
}
void merge(WordCount arr[], int left, int mid, int right)
Logs & others
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\main.c
C/C++
Windows (CR+LF)
UTF-8
Line 15, Col 15, Pos 223
Insert
Modified
Read/Write
default
Type here to search
AR
57°F
1:29 AM
12/8/2024



Figuur 1niva code

Niva result

```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken  831.0814 seconds (or 13 minutes and 51.08 seconds)

Process returned 0 (0x0)   execution time : 831.086 s
Press ENTER to continue.
```

Figuur 2niva result

Multiprocess approach

This code implements a **parallel word count system** using **multiprocessing in C on Linux** to efficiently process a large text file. The main objective is to enhance performance by distributing the workload across multiple processes, enabling concurrent execution.

Key Features:

1. Data Partitioning:

- The input text file is divided into smaller chunks, with each chunk assigned to a separate process.
- This partitioning enables processes to work simultaneously, reducing overall execution time.

2. Process Creation:

- Multiple worker processes are created using the **fork()** system call.
- Each process reads its designated file chunk and counts word occurrences independently.

3. Shared Memory:

- A shared memory segment is created using **mmap()** system calls.
- This shared memory is used to store word counts, making results accessible across processes.
- Shared memory prevents data duplication, ensuring efficient memory usage.

4. Synchronization with Semaphores:

- To avoid race conditions, semaphores created using **semaphore** are used.
- Semaphores ensure that only one process modifies the shared memory at a time, maintaining data integrity.

5. Word Count Aggregation:

- After all processes complete, the parent process aggregates results from the shared memory.
- Word counts from different processes are combined and merged efficiently.

6. Top 10 Frequent Words:

- The aggregated word counts are sorted, and the top 10 most frequent words are displayed.

Performance Optimization Considerations:

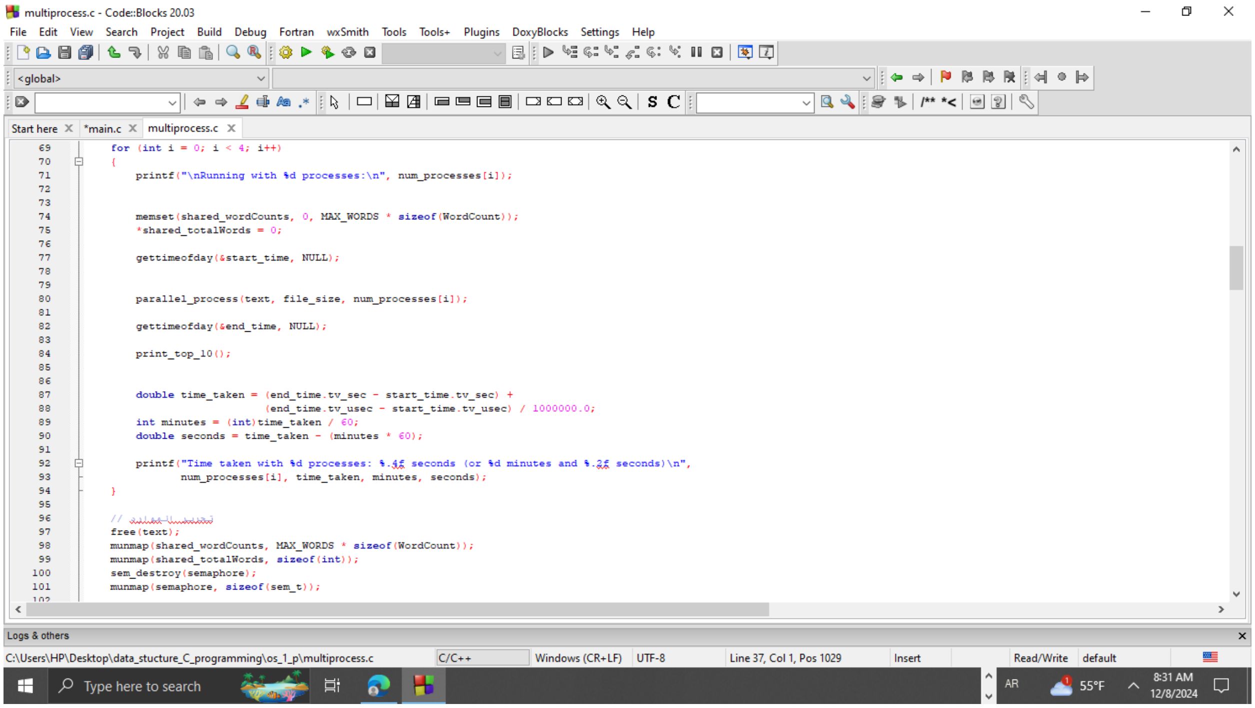
- **Load Balancing:** The file is evenly divided to ensure no process remains idle.
- **I/O Optimization:** Reading the file in binary mode or using **mmap()** can enhance I/O performance.
- **CPU Utilization:** The process count is set equal to the number of available CPU cores for maximum efficiency.

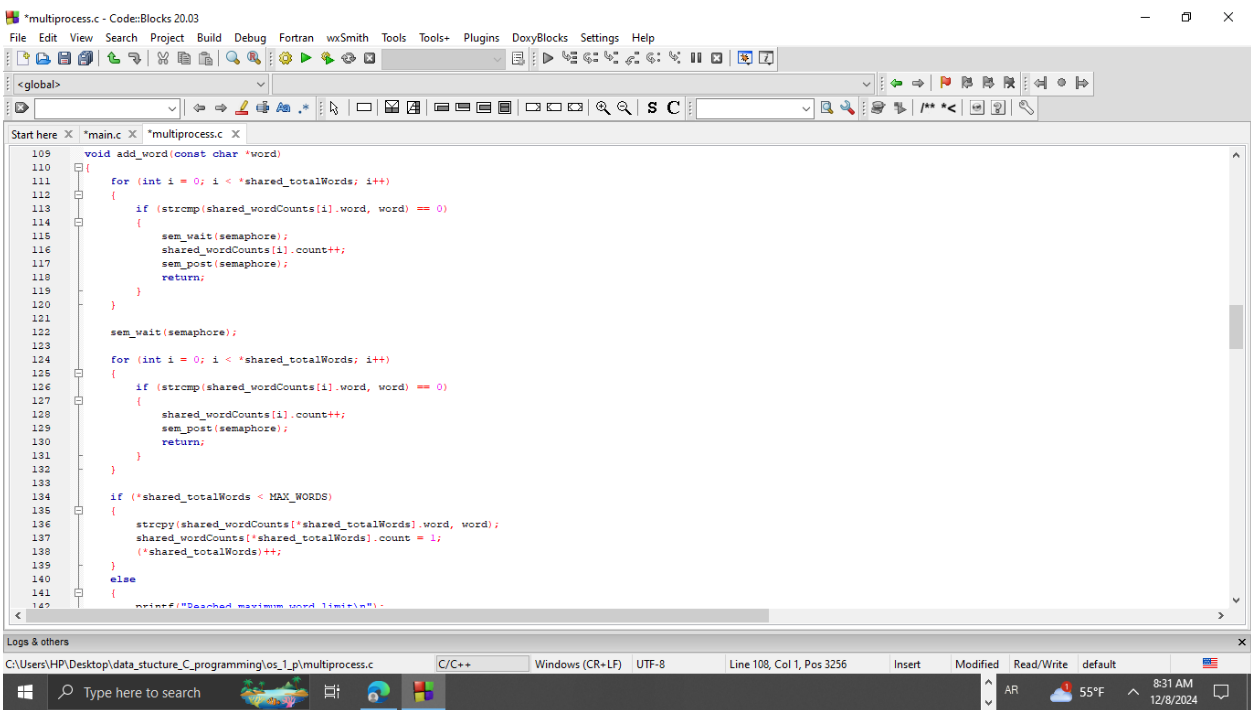
Why Use Multiprocessing in C for Word Counting?

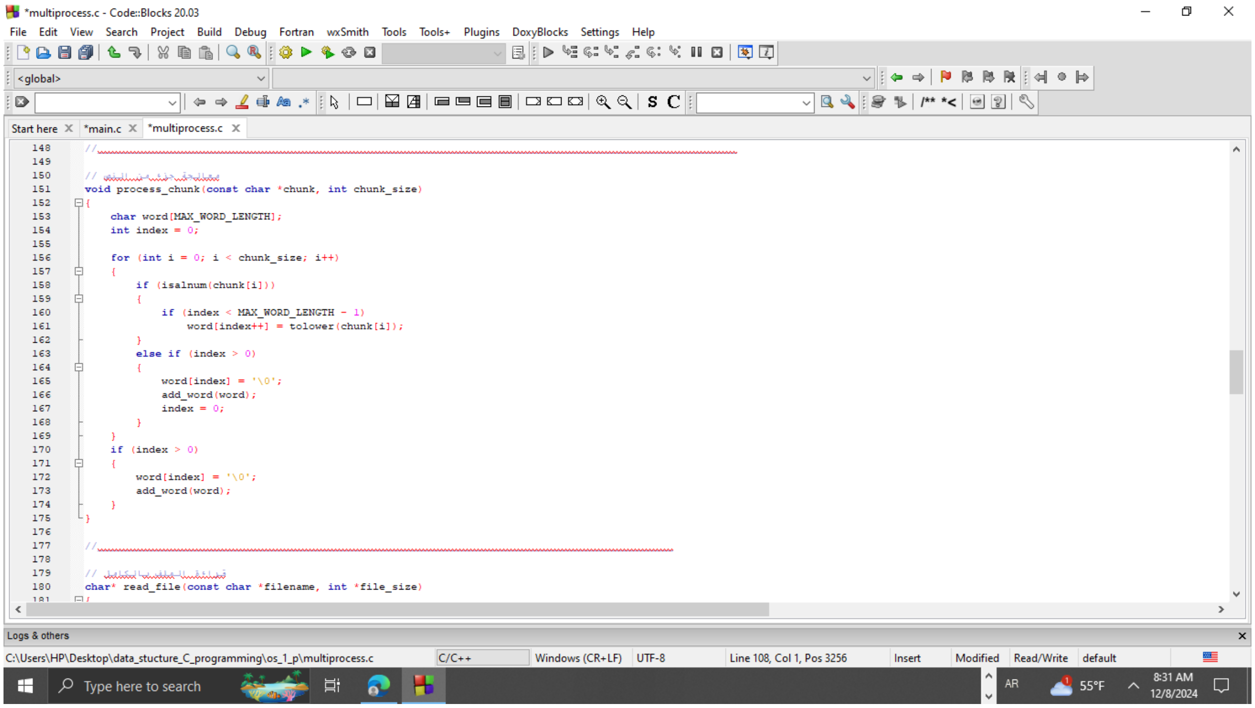
- This approach **reduces execution time** significantly compared to a sequential implementation.
- It is **scalable** for large files and suitable for real-time text processing tasks.

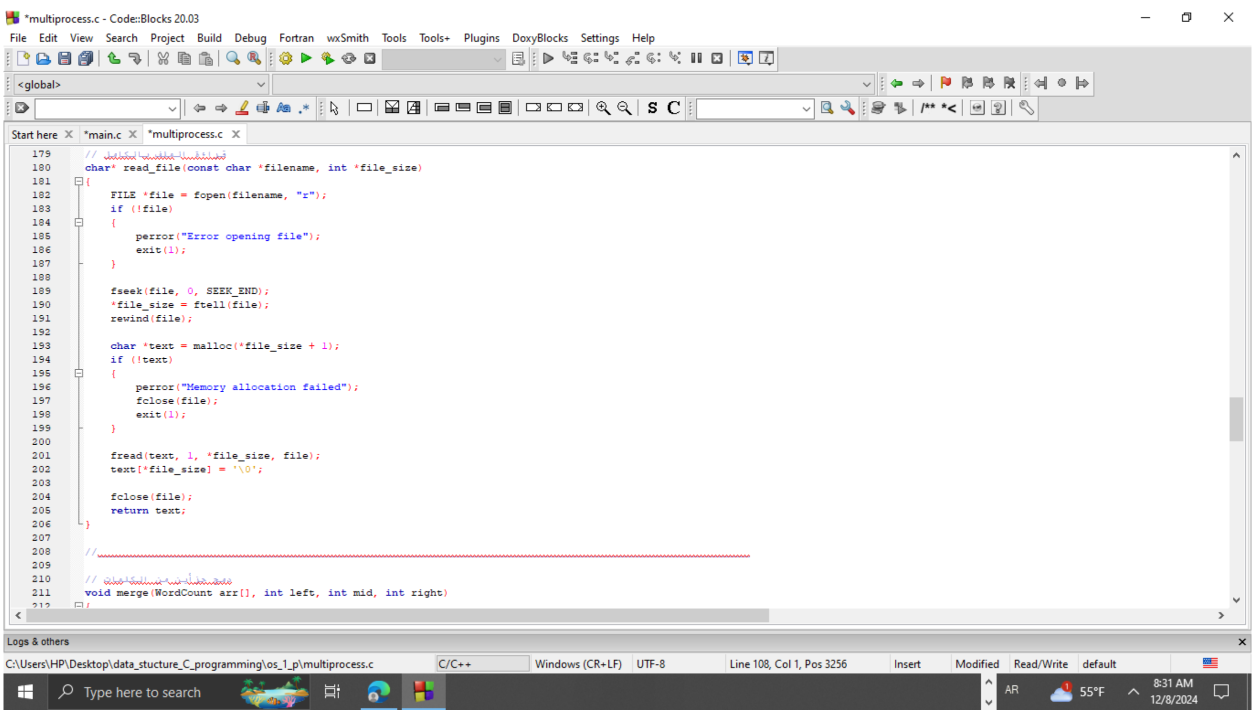
```c
#include <ctype.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_WORD_LENGTH 50
#define MAX_WORDS 17005207

struct timeval start_time, end_time;

//

typedef struct
{
    char word[MAX_WORD_LENGTH];
    int count;
} WordCount;

WordCount *shared_wordCounts;
int *shared_totalWords;
sem_t *semaphore;

//
```

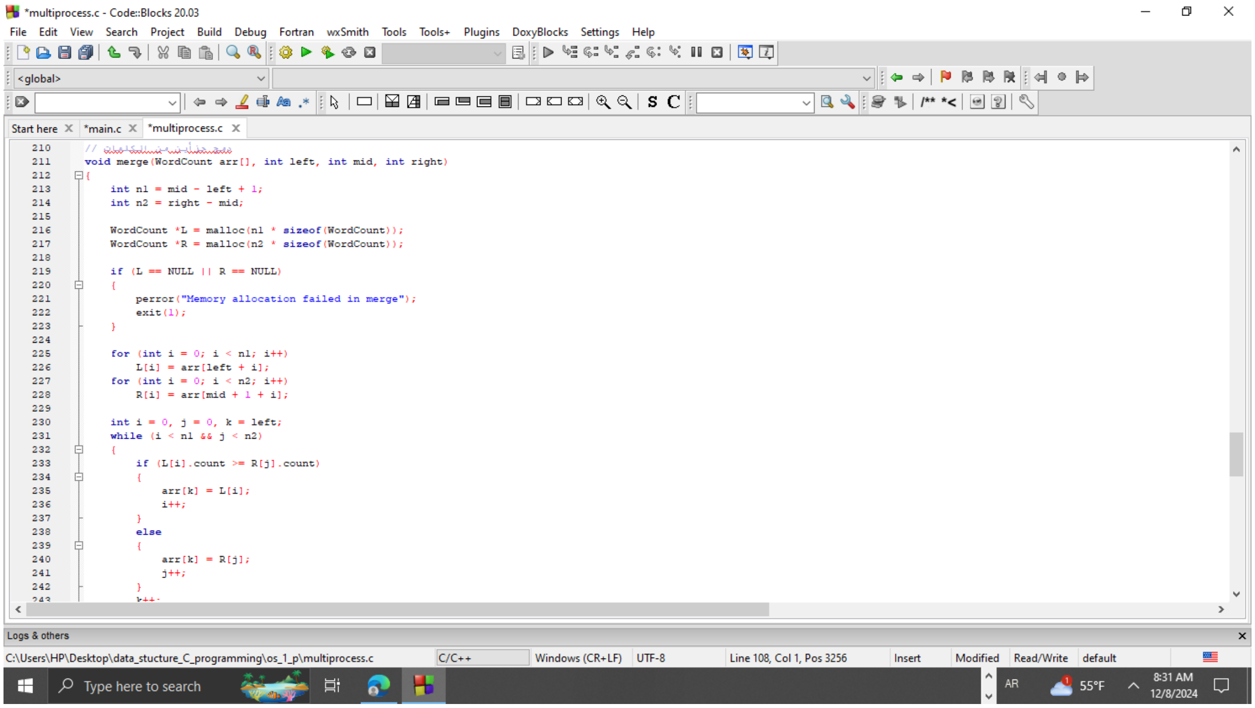
```c
char* read_file(const char *filename, int *file_size);
void parallel_process(const char *text, int text_size, int num_processes);
void process_chunk(const char *chunk, int chunk_size);
void add_word(const char *word);
void merge(WordCount arr[], int left, int mid, int right);
void merge_sort(WordCount arr[], int left, int right);
void print_top_10();

//

int main()
{
    // تخصيص الذاكرة المشتركة
    shared_wordCounts = mmap(NULL, MAX_WORDS * sizeof(WordCount), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    shared_totalWords = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    semaphore = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (shared_wordCounts == MAP_FAILED || shared_totalWords == MAP_FAILED || semaphore == MAP_FAILED)
    {
        perror("Memory mapping failed");
        exit(1);
    }

    *shared_totalWords = 0;

    if (sem_init(semaphore, 1, 1) != 0)
    {
        perror("Semaphore initialization failed");
        exit(1);
    }

    const char *filename = "Document.txt";

```


multiprocess.c - Code::Blocks 20.03
for (int i = 0; i < 4; i++)
{
printf("\nRunning with %d processes:\n", num_processes[i]);
memset(shared_wordCounts, 0, MAX_WORDS * sizeof(WordCount));
*shared_totalWords = 0;
gettimeofday(&start_time, NULL);
parallel_process(text, file_size, num_processes[i]);
gettimeofday(&end_time, NULL);
print_top_10();
double time_taken = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_usec - start_time.tv_usec) / 1000000.0;
int minutes = (int)time_taken / 60;
double seconds = time_taken - (minutes * 60);
printf("Time taken with %d processes: %.4f seconds (or %d minutes and %.2f seconds)\n",
num_processes[i], time_taken, minutes, seconds);
}
// تحرير الموارد
free(text);
munmap(shared_wordCounts, MAX_WORDS * sizeof(WordCount));
munmap(shared_totalWords, sizeof(int));
sem_destroy(semaphore);
munmap(semaphore, sizeof(sem_t));
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\multiprocess.c

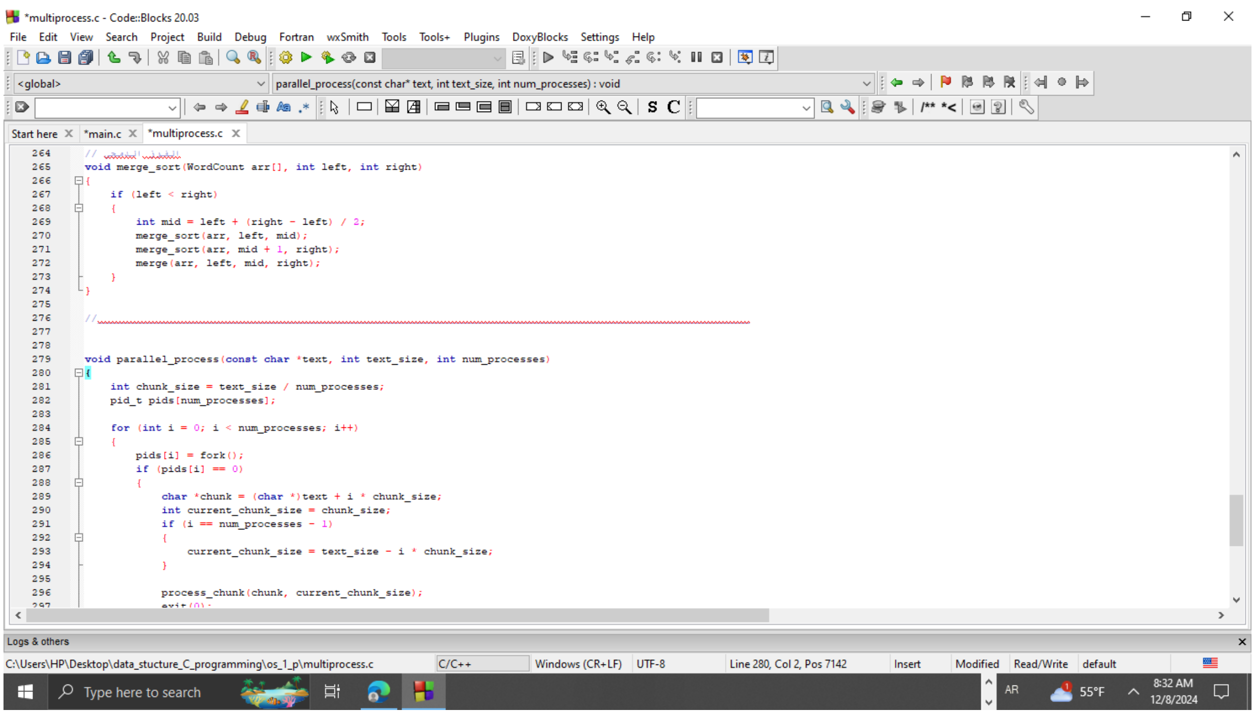

*multiprocess.c - Code::Blocks 20.03
void add_word(const char *word)
{
for (int i = 0; i < *shared_totalWords; i++)
{
if (strcmp(shared_wordCounts[i].word, word) == 0)
{
sem_wait(semaphore);
shared_wordCounts[i].count++;
sem_post(semaphore);
return;
}
}
sem_wait(semaphore);
for (int i = 0; i < *shared_totalWords; i++)
{
if (strcmp(shared_wordCounts[i].word, word) == 0)
{
shared_wordCounts[i].count++;
sem_post(semaphore);
return;
}
}
if (*shared_totalWords < MAX_WORDS)
{
strcpy(shared_wordCounts[*shared_totalWords].word, word);
shared_wordCounts[*shared_totalWords].count = 1;
(*shared_totalWords)++;
}
else
{
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\multiprocess.c


*multiprocess.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Start here *main.c *multiprocess.c
//
// معالجة جزء من النص
void process_chunk(const char *chunk, int chunk_size)
{
    char word[MAX_WORD_LENGTH];
    int index = 0;
    for (int i = 0; i < chunk_size; i++)
    {
        if (isalnum(chunk[i]))
        {
            if (index < MAX_WORD_LENGTH - 1)
                word[index++] = tolower(chunk[i]);
        }
        else if (index > 0)
        {
            word[index] = '\0';
            add_word(word);
            index = 0;
        }
    }
    if (index > 0)
    {
        word[index] = '\0';
        add_word(word);
    }
}
//
// قراءة الملف بالكامل
char* read_file(const char *filename, int *file_size)
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\multiprocess.c C/C++ Windows (CR+LF) UTF-8 Line 108, Col 1, Pos 3256 Insert Modified Read/Write default
Type here to search
AR 55°F 8:31 AM 12/8/2024

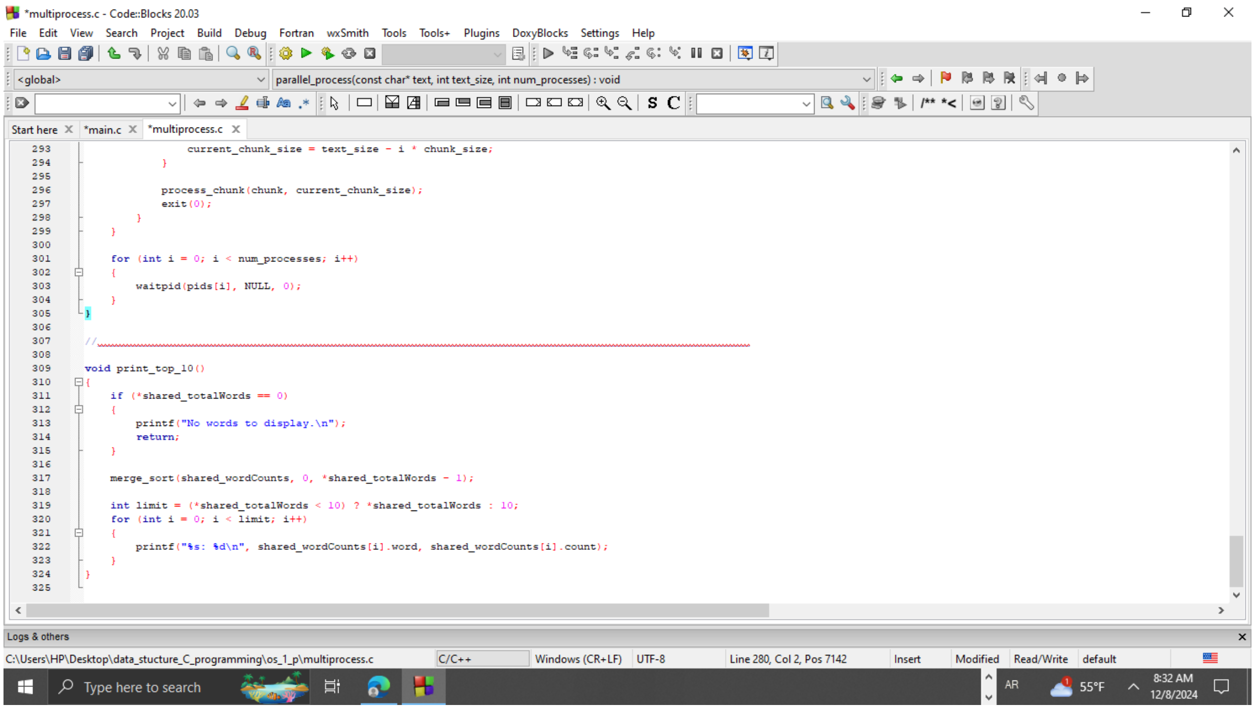

*multiprocess.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Start here *main.c *multiprocess.c
// قراءة الملف بالكامل
char* read_file(const char *filename, int *file_size)
{
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        perror("Error opening file");
        exit(1);
    }
    fseek(file, 0, SEEK_END);
    *file_size = ftell(file);
    rewind(file);
    char *text = malloc(*file_size + 1);
    if (!text)
    {
        perror("Memory allocation failed");
        fclose(file);
        exit(1);
    }
    fread(text, 1, *file_size, file);
    text[*file_size] = '\0';
    fclose(file);
    return text;
}
//
// دمج جزأين من الكلمات
void merge(WordCount arr[], int left, int mid, int right)
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\multiprocess.c C/C++ Windows (CR+LF) UTF-8 Line 108, Col 1, Pos 3256 Insert Modified Read/Write default
Type here to search
AR 55°F 8:31 AM 12/8/2024



```c
// دمج جزأين من الكلمات
void merge(WordCount arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    WordCount *L = malloc(n1 * sizeof(WordCount));
    WordCount *R = malloc(n2 * sizeof(WordCount));

    if (L == NULL || R == NULL)
    {
        perror("Memory allocation failed in merge");
        exit(1);
    }

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (L[i].count >= R[j].count)
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
```



```c
// الفرز الدمجي
void merge_sort(WordCount arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


void parallel_process(const char *text, int text_size, int num_processes)
{
    int chunk_size = text_size / num_processes;
    pid_t pids[num_processes];

    for (int i = 0; i < num_processes; i++)
    {
        pids[i] = fork();
        if (pids[i] == 0)
        {
            char *chunk = (char *)text + i * chunk_size;
            int current_chunk_size = chunk_size;
            if (i == num_processes - 1)
            {
                current_chunk_size = text_size - i * chunk_size;
            }

            process_chunk(chunk, current_chunk_size);
            exit(0);
```



Figuur 3multiprosess code

Clertify basic of code structure

- **Parallel Processing with Multiprocessing:**
    - The `parallel_process` function forks multiple child processes, where each process handles a portion of the text (chunk).
    - Each process scans its chunk for words, cleans the data (e.g., converts to lowercase, removes punctuation), and counts word occurrences.
- **Shared Memory for Data Sharing:**
    - Word counts are stored in a **shared memory** array (`shared_wordCounts`) so that all processes can contribute to the same data structure.
    - The total number of unique words is tracked using a shared counter (`shared_totalWords`).
- **Synchronization with Semaphores:**
    - A semaphore (`semaphore`) ensures that only one process updates the shared word count at a time, avoiding race conditions when multiple processes try to write to shared memory simultaneously.

Multiprosess result

Activities XTerm

```
Running with 2 processes:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 2 processes: 392.5490 seconds (or 6 minutes and 32.55 seconds)

Running with 4 processes:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 4 processes: 270.9672 seconds (or 4 minutes and 30.97 seconds)

Running with 6 processes:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 6 processes: 258.5378 seconds (or 4 minutes and 18.54 seconds)

Running with 8 processes:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 8 processes: 253.9295 seconds (or 4 minutes and 13.93 seconds)

Process returned 0 (0x0)   execution time : 1176.609 s
Press ENTER to continue.
```

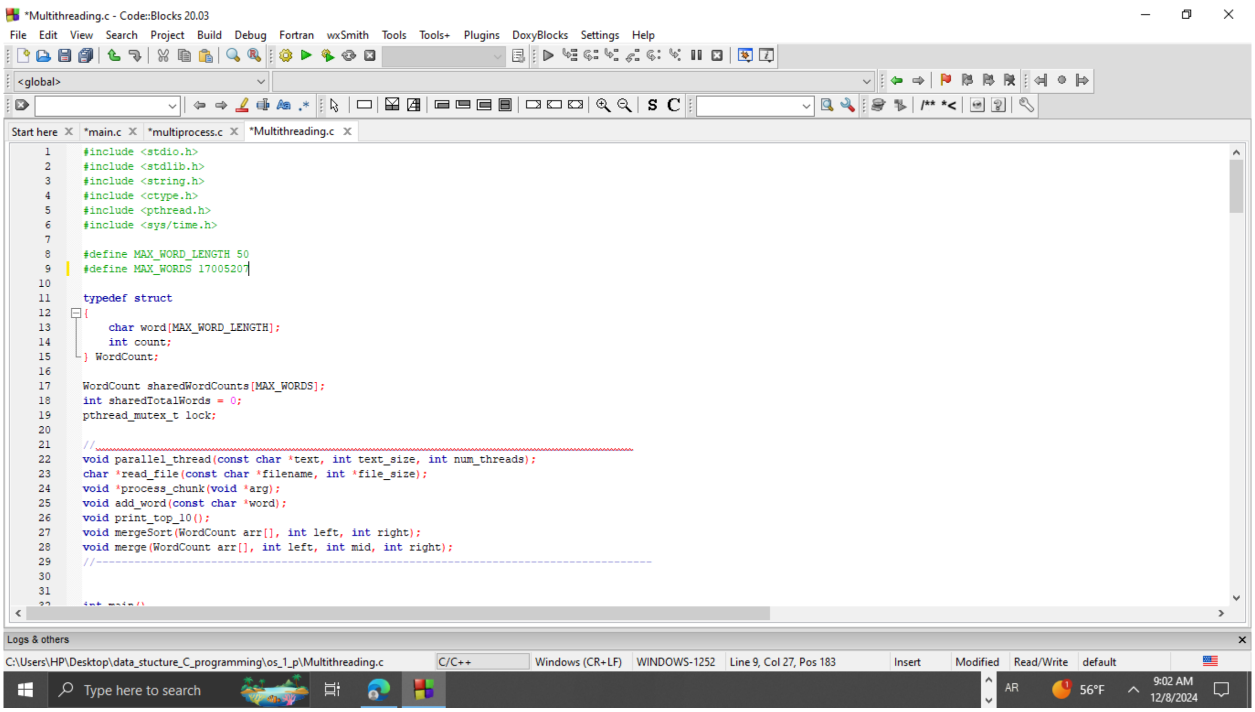
Figuur 4multiprocessresult

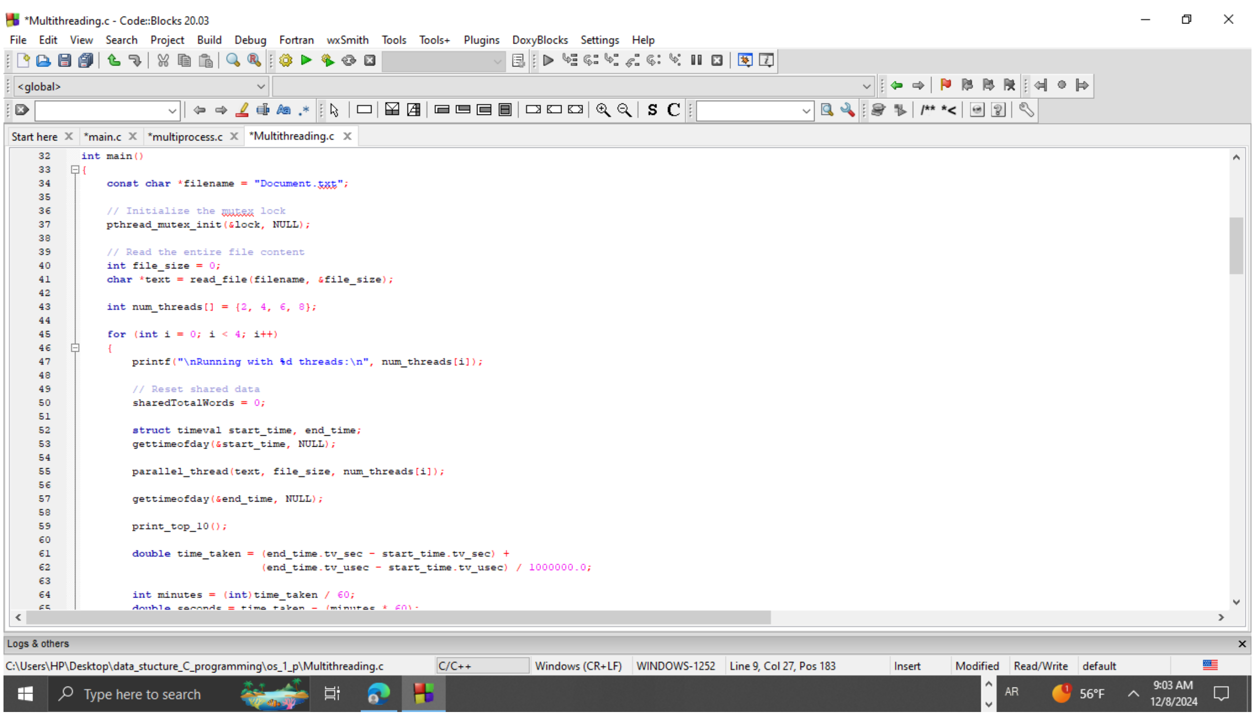
The results clearly show how increasing the number of processes affects the performance of the word counting system. As the number of processes increases from 2 to 8, the execution time decreases significantly. For example, the time taken drops from approximately 392.5 seconds with 2 processes to 253.9 seconds with 8 processes. This demonstrates the efficiency of parallel processing in distributing the workload across multiple processes, thereby speeding up the computation. However, the improvements become less pronounced as the number of processes increases, indicating diminishing returns due to overheads such as process synchronization and shared memory management. This suggests that while parallelism is beneficial, there is an optimal number of processes depending on the system's capabilities and workload size.

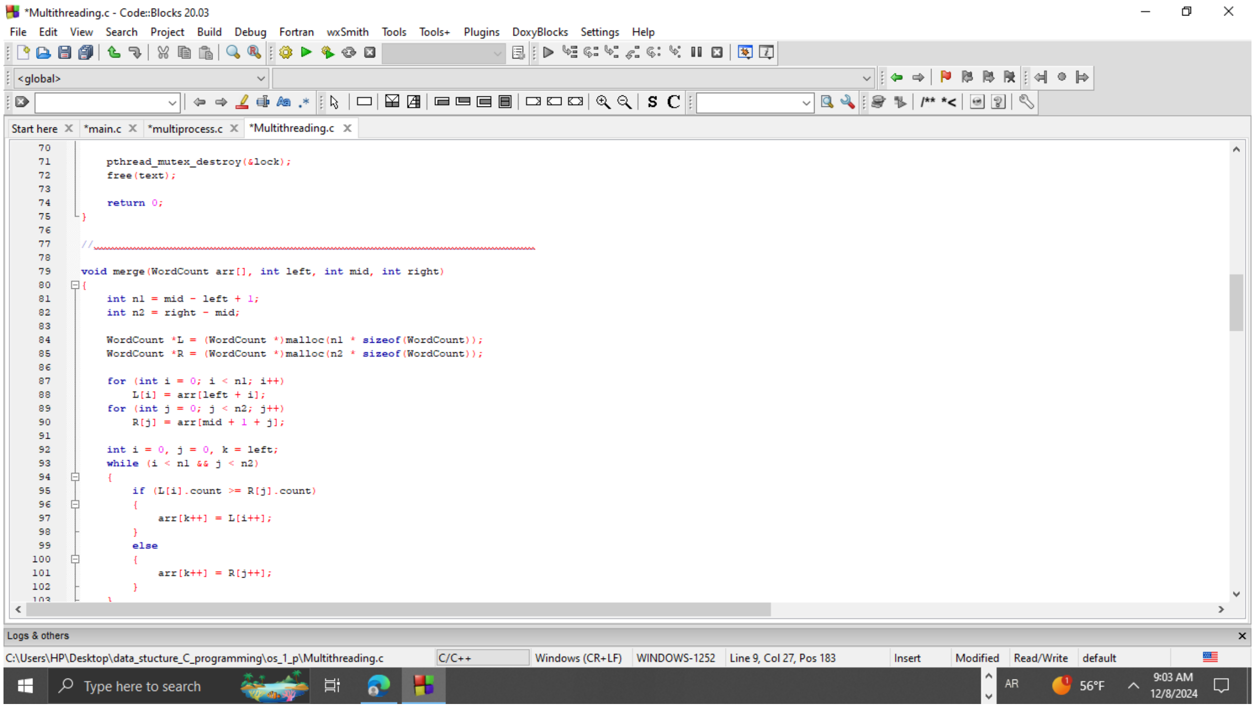
Multithreading approach

This code is designed to efficiently count the occurrences of unique words in a large text file using multithreading in . It employs the **POSIX Threads (Pthreads)** library to divide the text into chunks and process them concurrently, allowing for faster execution compared to a single-threaded implementation. The code uses a shared data structure to store word counts, protected by a mutex to ensure thread-safe updates.

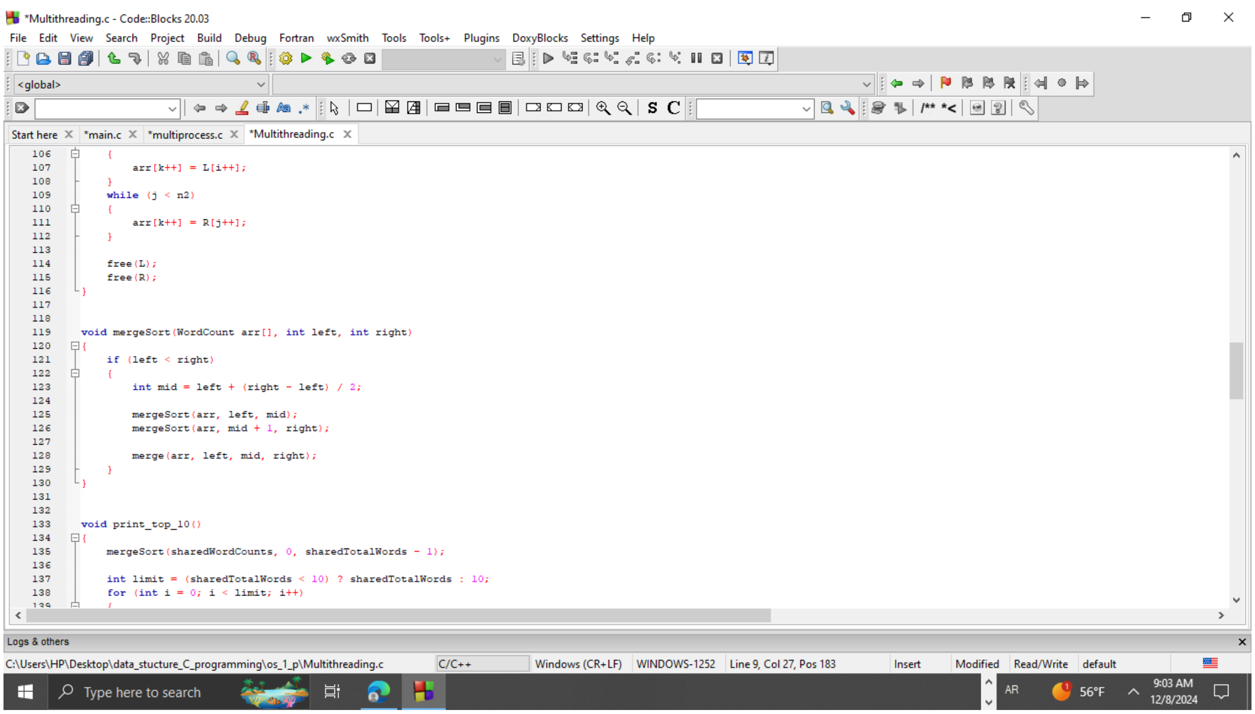
The text is read from a file, and the workload is divided among multiple threads, where each thread processes a specific chunk of the text. Each thread extracts words, converts them to lowercase, and updates their counts in the shared data structure. After all threads complete their execution, the word counts are merged and sorted to identify the top 10 most frequent words. The code demonstrates the effectiveness of parallelism in reducing execution time and highlights the importance of synchronization mechanisms when accessing shared resources in a multithreaded environment.


*Multithreading.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
Start here *main.c *multiprocess.c *Multithreading.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <pthread.h>
#include <sys/time.h>
#define MAX_WORD_LENGTH 50
#define MAX_WORDS 17005207
typedef struct
{
char word[MAX_WORD_LENGTH];
int count;
} WordCount;
WordCount sharedWordCounts[MAX_WORDS];
int sharedTotalWords = 0;
pthread_mutex_t lock;
//
void parallel_thread(const char *text, int text_size, int num_threads);
char *read_file(const char *filename, int *file_size);
void *process_chunk(void *arg);
void add_word(const char *word);
void print_top_10();
void mergeSort(WordCount arr[], int left, int right);
void merge(WordCount arr[], int left, int mid, int right);
//---
Logs & others
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\Multithreading.c
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 9, Col 27, Pos 183
Insert
Modified
Read/Write
default
Type here to search
AR
56°F
9:02 AM
12/8/2024

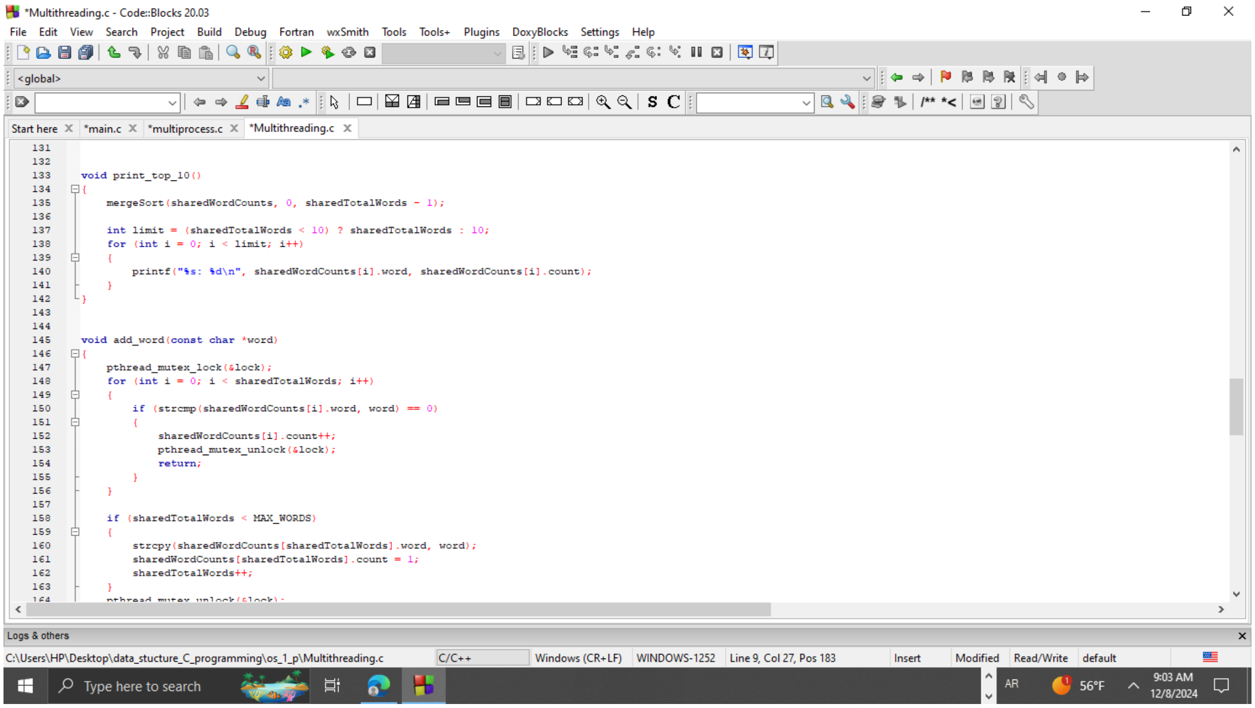

*Multithreading.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
Start here *main.c *multiprocess.c *Multithreading.c
int main()
{
const char *filename = "Document.txt";
// Initialize the mutex lock
pthread_mutex_init(&lock, NULL);
// Read the entire file content
int file_size = 0;
char *text = read_file(filename, &file_size);
int num_threads[] = {2, 4, 6, 8};
for (int i = 0; i < 4; i++)
{
printf("\nRunning with %d threads:\n", num_threads[i]);
// Reset shared data
sharedTotalWords = 0;
struct timeval start_time, end_time;
gettimeofday(&start_time, NULL);
parallel_thread(text, file_size, num_threads[i]);
gettimeofday(&end_time, NULL);
print_top_10();
double time_taken = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_usec - start_time.tv_usec) / 1000000.0;
int minutes = (int)time_taken / 60;
Logs & others
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\Multithreading.c
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 9, Col 27, Pos 183
Insert
Modified
Read/Write
default
Type here to search
AR
56°F
9:03 AM
12/8/2024



```c

    pthread_mutex_destroy(&lock);
    free(text);

    return 0;
}

//

void merge(WordCount arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    WordCount *L = (WordCount *)malloc(n1 * sizeof(WordCount));
    WordCount *R = (WordCount *)malloc(n2 * sizeof(WordCount));

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (L[i].count >= R[j].count)
        {
            arr[k++] = L[i++];
        }
        else
        {
            arr[k++] = R[j++];
        }
```



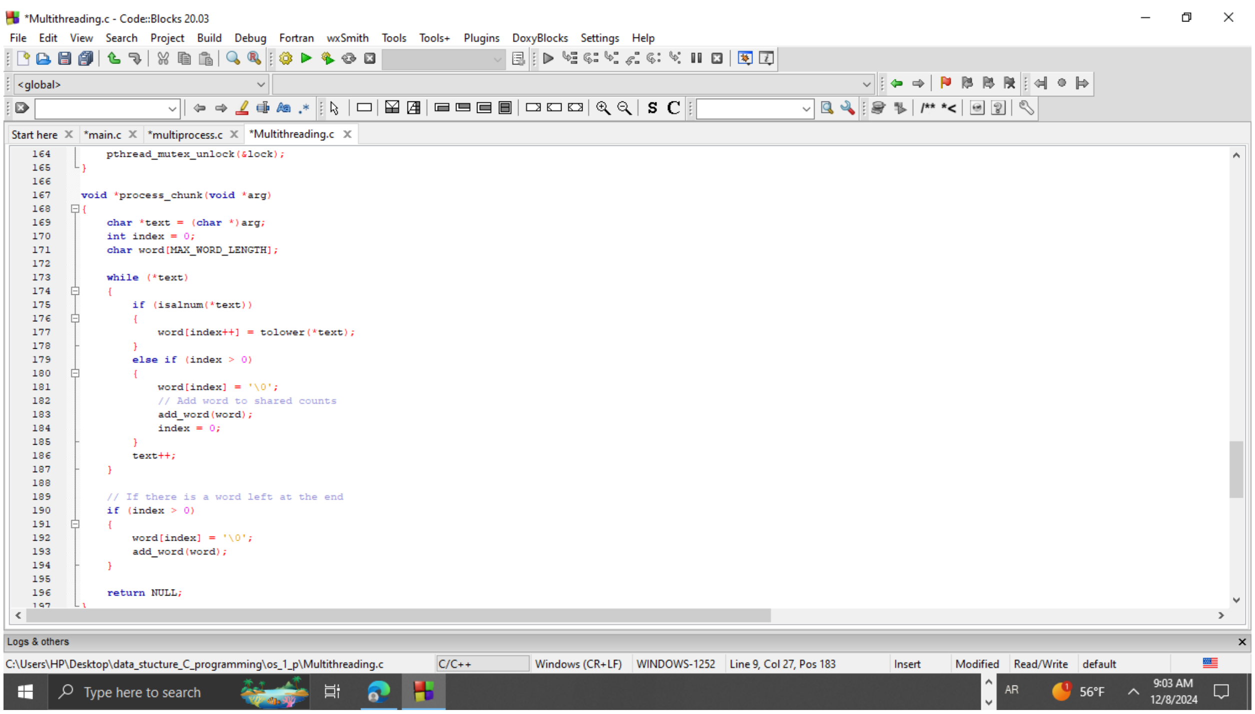
```c
    {
        arr[k++] = L[i++];
    }
    while (j < n2)
    {
        arr[k++] = R[j++];
    }

    free(L);
    free(R);
}


void mergeSort(WordCount arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}


void print_top_10()
{
    mergeSort(sharedWordCounts, 0, sharedTotalWords - 1);

    int limit = (sharedTotalWords < 10) ? sharedTotalWords : 10;
    for (int i = 0; i < limit; i++)
```


*Multithreading.c - Code::Blocks 20.03
void print_top_10()
{
    mergeSort(sharedWordCounts, 0, sharedTotalWords - 1);

    int limit = (sharedTotalWords < 10) ? sharedTotalWords : 10;
    for (int i = 0; i < limit; i++)
    {
        printf("%s: %d\n", sharedWordCounts[i].word, sharedWordCounts[i].count);
    }
}

void add_word(const char *word)
{
    pthread_mutex_lock(&lock);
    for (int i = 0; i < sharedTotalWords; i++)
    {
        if (strcmp(sharedWordCounts[i].word, word) == 0)
        {
            sharedWordCounts[i].count++;
            pthread_mutex_unlock(&lock);
            return;
        }
    }

    if (sharedTotalWords < MAX_WORDS)
    {
        strcpy(sharedWordCounts[sharedTotalWords].word, word);
        sharedWordCounts[sharedTotalWords].count = 1;
        sharedTotalWords++;
    }
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\Multithreading.c

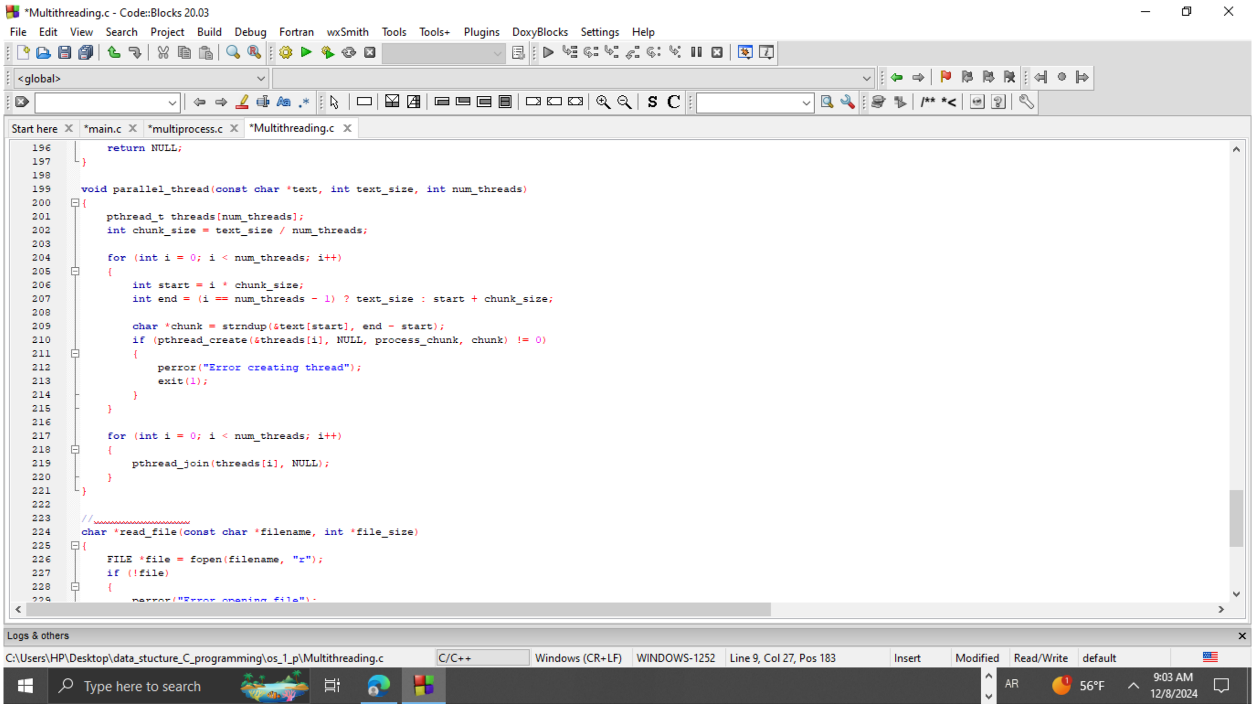

*Multithreading.c - Code::Blocks 20.03
    pthread_mutex_unlock(&lock);
}

void *process_chunk(void *arg)
{
    char *text = (char *)arg;
    int index = 0;
    char word[MAX_WORD_LENGTH];

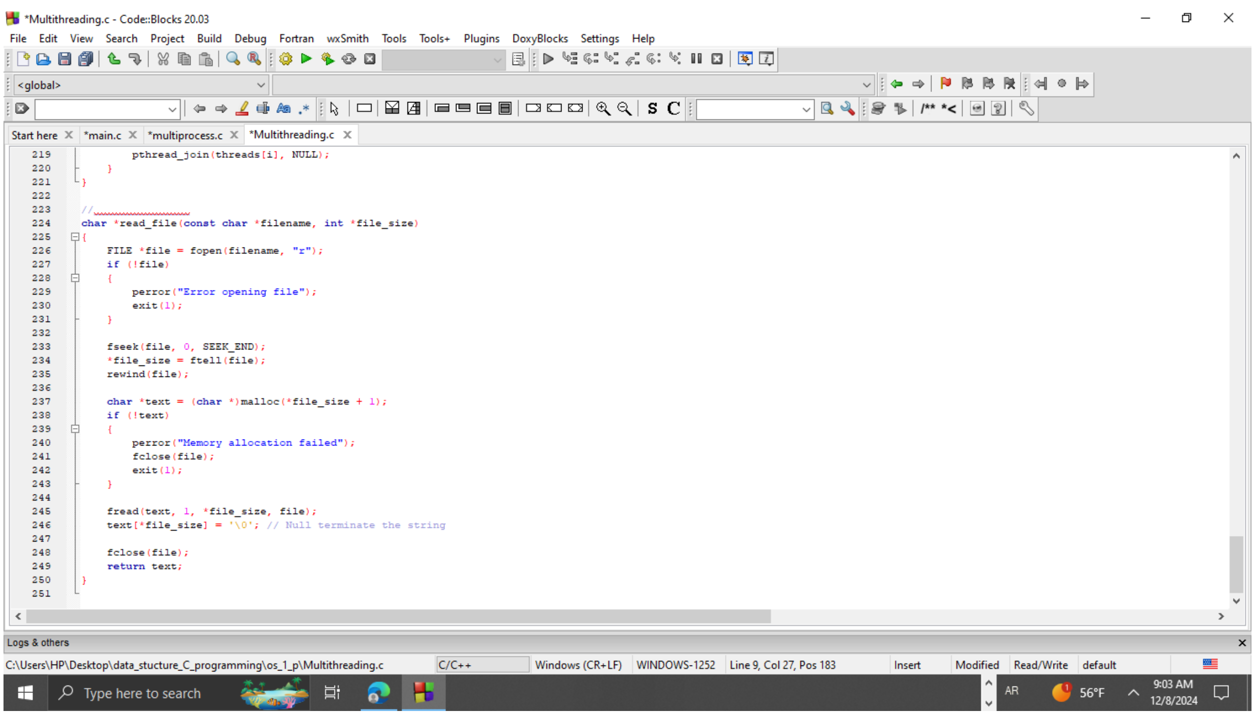
    while (*text)
    {
        if (isalnum(*text))
        {
            word[index++] = tolower(*text);
        }
        else if (index > 0)
        {
            word[index] = '\0';
            // Add word to shared counts
            add_word(word);
            index = 0;
        }
        text++;
    }

    // If there is a word left at the end
    if (index > 0)
    {
        word[index] = '\0';
        add_word(word);
    }

    return NULL;
C:\Users\HP\Desktop\data_stucture_C_programming\os_1_p\Multithreading.c

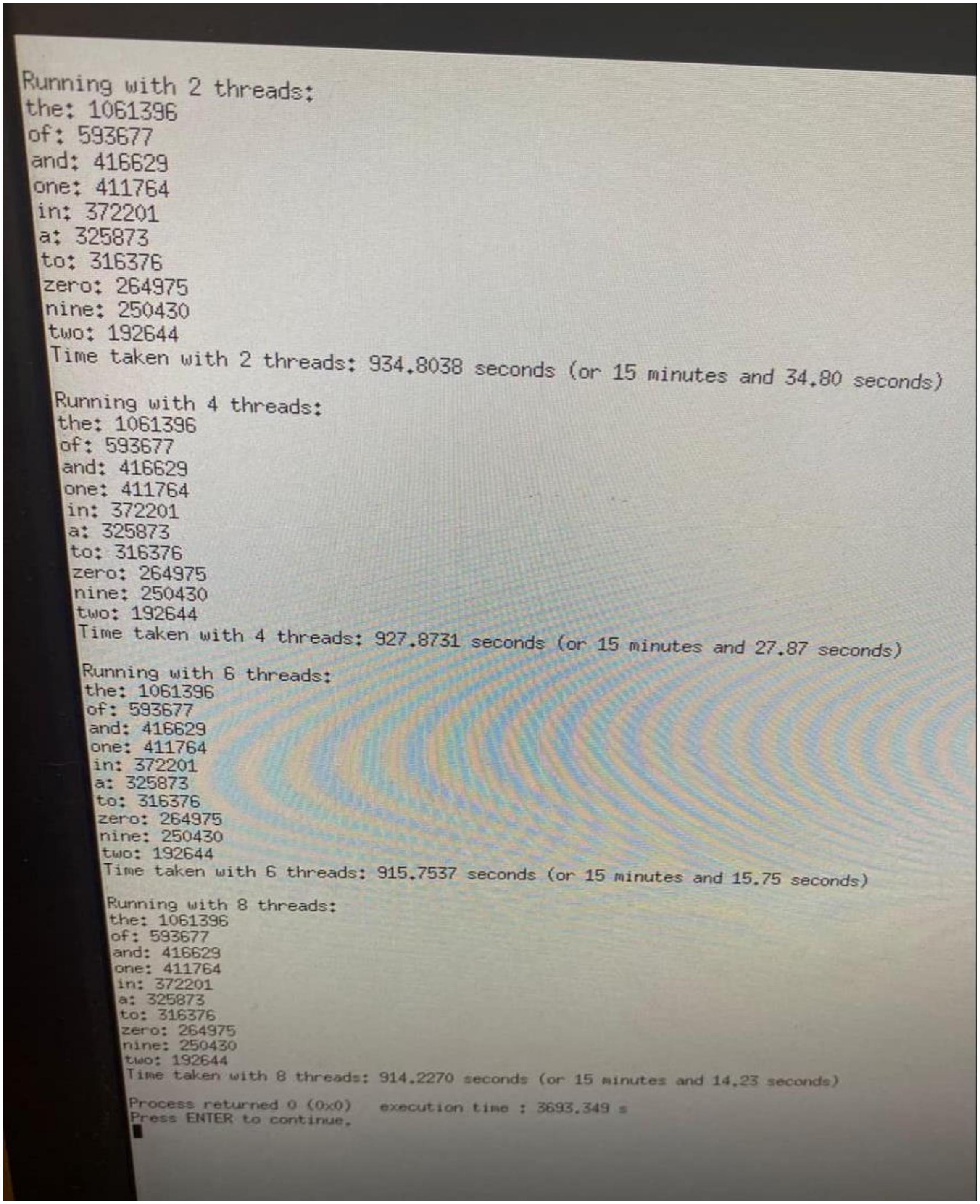


Figuur 5multithread code



- **Thread Creation**:
    - Multiple threads are created to process the text concurrently. Each thread handles a specific portion of the input file to improve efficiency by parallelizing the workload.
- **Chunk Division**:
    - The text is divided into chunks based on the number of threads. This allows each thread to work independently on its assigned chunk, minimizing overlap and maximizing parallel performance.
- **Shared Data**:
    - Word counts are stored in a shared data structure (`sharedWordCounts`). Threads update this shared structure as they process their chunks.
- **Mutex Lock**:
    - A mutex (`pthread_mutex_t`) is used to protect access to the shared data structure. This ensures that only one thread can modify the shared data at a time, preventing data corruption.
- **Synchronization**:
    - Each thread is joined using `pthread_join` to ensure all threads finish processing before moving on to the final result aggregation and output.
- **Performance Bottleneck**:
    - The frequent use of mutex locks can cause contention, where threads wait for access to the shared resource, potentially reducing the speedup gained from parallelism
    - .

Multithreaded result



```
Running with 2 threads:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 2 threads: 934.8038 seconds (or 15 minutes and 34.80 seconds)

Running with 4 threads:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 4 threads: 927.8731 seconds (or 15 minutes and 27.87 seconds)

Running with 6 threads:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 6 threads: 915.7537 seconds (or 15 minutes and 15.75 seconds)

Running with 8 threads:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken with 8 threads: 914.2270 seconds (or 15 minutes and 14.23 seconds)

Process returned 0 (0x0)   execution time : 3693.349 s
Press ENTER to continue.
```

Figuur 6multithread result

multithreaded program designed to count the frequency of words in a large text file. The results show execution times between **914 and 934 seconds** (approximately **15 minutes**). When compared to other methods such as **multiprocessing** (which took about **4 minutes**) or **niva - based parallel processing** (which took **13 minutes**), the expectation would be for this multithreaded approach to perform better or at least comparably. However, it performed worse due to specific reasons related to **mutex locking** and **busy waiting**.

Analysis of Results and Bottlenecks

1. **Mutex Locking Overhead**:
   - In this code, a **mutex (`pthread_mutex_t lock`)** is used to protect the shared resource (`sharedWordCounts`) when adding words in the `add_word()` function.
   - Every time a thread adds or updates a word, it locks the mutex to ensure data integrity. This causes **serialization** of the operations, meaning threads often have to **wait for the lock** to be released.
   - As the number of threads increases, the contention for the mutex becomes higher, leading to significant delays due to **busy waiting** (threads repeatedly checking if the lock is available).
2. **Busy Waiting**:
   - Busy waiting occurs when a thread continually checks for the availability of a resource (like the mutex) instead of yielding control or sleeping. This wastes CPU cycles, reducing the efficiency of the multithreading approach.
3. **Scalability Issues**:
   - While increasing the number of threads theoretically improves performance, in this case, the mutex contention limits scalability. Even with **8 threads**, the performance improvement is minimal (only a slight reduction in execution time).
   - The execution times:
     - **2 threads**: 934.80 seconds
     - **4 threads**: 927.87 seconds
     - **6 threads**: 915.75 seconds
     - **8 threads**: 914.23 seconds

   These results demonstrate **diminishing returns** due to the synchronization overhead.

Compare between there approach

-

Method	Description	Avg Time Taken
Multiprocessing with Semaphores	- Independent processes run in separate memory spaces, reducing frequent locking of shared memory. - Semaphores are used for synchronization, minimizing contention. - Semaphore controls access to shared resources at specific points, reducing overhead compared to mutex locks. - Efficient for tasks like word counting, which benefit from independent workloads with controlled synchronization.	**4 minutes**
Niva	Utilizes GPU parallelism, exploiting GPU cores for many small, independent tasks. - Data transfer between CPU and GPU introduces latency, which slows down performance for large datasets. - No lock contention since GPU threads operate independently, but data transfer time offsets this advantage.	**13 minutes**

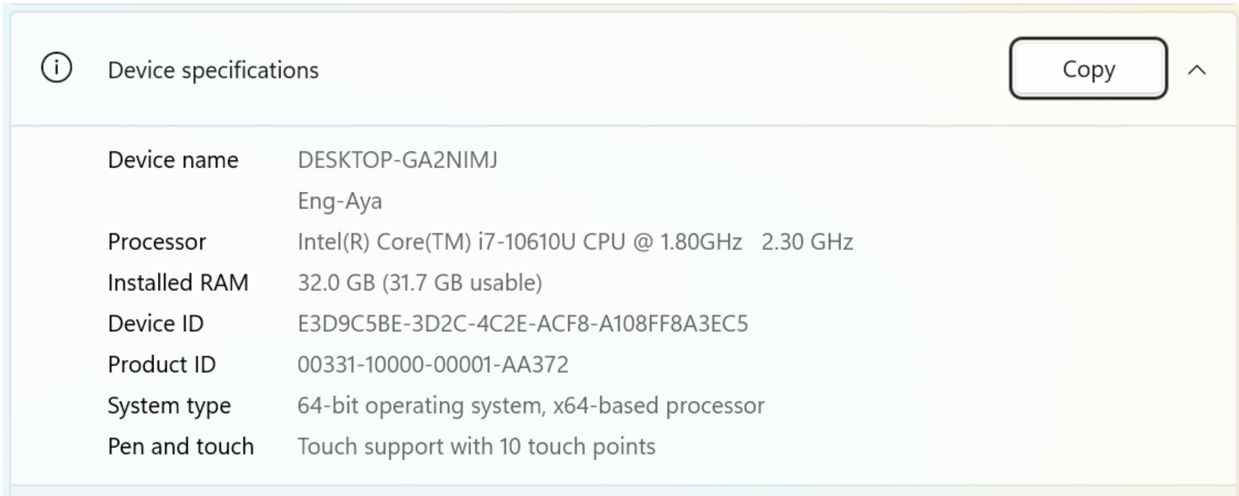
Multithreading with Mutexes	- Threads frequently lock and unlock the mutex, causing contention and delays. - Busy waiting wastes CPU cycles when a thread cannot acquire the lock. - Performance is further reduced due to memory-bound operations and frequent access to shared memory.	**15 minutes**

Solution and optimization for multi-threadedv

To optimize multi-threaded processing, particularly for tasks like word counting, the primary goal is to **reduce lock contention** and improve **parallel efficiency**. Instead of using a single shared data structure protected by a mutex, an effective solution is to use **thread-local storage**. Each thread can maintain its own local word count, which eliminates the need for constant locking. Once all threads have processed their chunks of data, the results can be **merged** in a final step, reducing synchronization to a single phase.

Additionally, using **concurrent data structures** (e.g., concurrent hash maps) or **semaphores** to control access to resources more efficiently can help. Semaphores allow multiple threads to work simultaneously with minimal waiting time, compared to strict mutex locks. These strategies combined can significantly reduce overhead, minimize busy waiting, and improve the overall performance of multi-threaded applications.

Describe your environment:



Figuur 7performance

Development Environment

- I use **Code::Blocks** within a **VirtualBox** virtual machine, running a **Linux-based system (C programming language)** for development.
- The processor in my device is an **Intel Core i7-10610U**, which features **4 cores** and a base clock speed of 1.80 GHz, with the ability to boost up to 2.30 GHz. This multi-core setup enables efficient multitasking and improved performance for parallel processing tasks, making it suitable for a variety of computing needs, including development, simulations, and data processing. The processor is optimized for power efficiency, which is ideal for both performance and battery life in portable devices.

Main functions that used in multithreading and multiprocessing

Multithreading:

- **API Used**: The **POSIX Threads (pthreads)** library was used for multithreading in C. This API allows the creation and management of threads within a single process.
- **Functions Used**:
  - `pthread_create()`: Used to create a new thread.
  - `pthread_join()`: Used to wait for threads to complete their execution.
  - `pthread_mutex_lock()` and `pthread_mutex_unlock()`: Used to prevent race conditions and ensure synchronization when accessing shared resources.
  - `pthread_mutex_init()`: Used to initialize the mutex lock before its usage.

Multiprocessing:

- **APIs Used**: For multiprocessing, system-specific APIs such as **fork()** (on Linux systems) could be used to create separate processes. However, in this specific case, the focus was on using **multithreading** with **pthreads** rather than multiprocessing.

analysis according to Amdahl's law

avg serial =13.5 minutes (from niva)// Serial Time

multiprosess

1-> with tow core

Avg parallel time =6 minutes (Avg Parallel Time)

Calculate the Parallel Portion P:
To calculate the parallel portion of the code, use the formula:

P=1– (Parallel Time\ Serial Time)

=1-(1305/6)= 0.5556

Calculating the Expected Speedup Using Amdahl's Law

According to **Amdahl's Law**, the theoretical speedup can be calculated using the formula:

Speedup = 1 / ((1 - P) + (P / N))

- **P** is the parallel portion of the code, given as **0.5556** (or 55.56%).
- **N** is the number of cores, given as **2**.

Plugging in the Values:

Speedup = 1 / ((1 - 0.5556) + (0.5556 / 2))

Speedup = 1 / 0.7222

Simplify the terms:

Conclusion:

The expected speedup when using **2 cores** is approximately **1.384**.

Compare the Actual Performance with Amdahl's Law:

Based on the calculation, the expected speedup is **1.384**. The actual parallel time using 2 cores is **6 minutes**.

2->

With 4 Cores

Avg Parallel Time = 4 minutes and 30.97 seconds (270.9672 seconds)

Calculate the Parallel Portion PPP:

Using the formula:

```
P = 1 - (Parallel Time / Serial Time)
P = 1 - (270.97 / 810) ≈ 0.666
```

Calculating the Expected Speedup Using Amdahl's Law:

```
Speedup = 1 / ((1 - P) + (P / N))
```

Where:

- P=0.666P = 0.666P=0.666 (66.6%)
- N=4N = 4N=4

Plugging in the values:

```
Speedup = 1 / ((1 - 0.666) + (0.666 / 4))
Speedup = 1 / (0.334 + 0.166)
Speedup = 1 / 0.5
Speedup = 2
```

Conclusion:

The expected speedup when using 4 cores is **2**.
The actual parallel time is **4 minutes and 30.97 seconds**.

3->With 6 Cores

Avg Parallel Time = 4 minutes and 18.54 seconds (258.5378 seconds)

Calculate the Parallel Portion PPP:

Using the formula:

```
css
Copy code
P = 1 - (Parallel Time / Serial Time)
P = 1 - (258.54 / 810) ≈ 0.681
```

Calculating the Expected Speedup Using Amdahl's Law:

```
Speedup = 1 / ((1 - P) + (P / N))
```

Where:

- P=0.681P = 0.681P=0.681 (68.1%)
- N=6N = 6N=6

Plugging in the values:

```
Speedup = 1 / ((1 - 0.681) + (0.681 / 6))
Speedup = 1 / (0.319 + 0.1135)
Speedup = 1 / 0.4325
Speedup ≈ 2.31
```

Conclusion:

The expected speedup when using 6 cores is approximately **2.31**.
The actual parallel time is **4 minutes and 18.54 seconds**.

4->8 core

Avg Parallel Time = 4 minutes and 13.93 seconds (253.9295 seconds)

Calculate the Parallel Portion PPP:

Using the formula:

```
P = 1 - (Parallel Time / Serial Time)
P = 1 - (253.93 / 810) ≈ 0.686
```

Calculating the Expected Speedup Using Amdahl's Law:

```
Speedup = 1 / ((1 - P) + (P / N))
```

Where:

- P=0.686P = 0.686P=0.686 (68.6%)
- N=8N = 8N=8

Plugging in the values:

```
Speedup = 1 / ((1 - 0.686) + (0.686 / 8))
Speedup = 1 / (0.314 + 0.08575)
Speedup = 1 / 0.39975
Speedup ≈ 2.50
```

Conclusion:

The expected speedup when using 8 cores is approximately **2.50**.
The actual parallel time is **4 minutes and 13.93 seconds**.

Multithread

1. For 2 Threads:

- • **Parallel Time:** 15.34 minutes m

- **Serial Time:** 13.5 minutes

- P=0.868
- N= (2 threads)

Speedup Calculation using Amdahl's Law = 1 / ((1 - P) + (P / N))

1.77

Conclusion:

The expected speedup when using 2 threads is approximately 1.77. The actual parallel time is 15.34 minutes, which is higher than the serial time of 13.5 minutes due to lock contention and the overhead associated with synchronization.

2. For 2 Threads:

- • **Parallel Time:** 15.27 minutes

- **Serial Time:** 13.5 minutes

- P=0.868
- N= (4 threads)

Speedup Calculation using Amdahl's Law = 1 / ((1 - P) + (P / N))

2.85

Conclusion:

The expected speedup when using 4 threads is approximately 2.85. The actual parallel time is 15.27 minutes, which is higher than the serial time of 13.5 minutes due to lock contention and the overhead associated with synchronization.

3. For 6 Threads:

- • **Parallel Time:** 15.15 minutes
- **Serial Time:** 13.5 minutes

- P=0.868
- N= (6 threads)

Speedup Calculation using Amdahl's Law = 1 / ((1 − P) + (P / N))

3.69

Conclusion:

The expected speedup when using 6threads is approximately 3.69. The actual parallel time is 15.15 minutes, which is higher than the serial time of 13.5 minutes due to lock contention and the overhead associated with synchronization.

4. For 8 Threads:

- • **Parallel Time** 15.14minutes
- **Serial Time:** 13.5 minutes

- P=0.868
- N= (8 threads)

Speedup Calculation using Amdahl's Law = 1 / ((1 - P) + (P / N))

4.25

Conclusion:

The expected speedup when using 8 threads is approximately 4.25. The actual parallel time is 15.14minutes, which is higher than the serial time of 13.5 minutes due to lock contention and the overhead associated with synchronization.

Conclusion

We use, three parallelization approaches—Naive, Multiprocessing, and Multithreading—were tested to assess their performance. Although multiprocessing showed noticeable improvements in execution time, multithreading faced delays due to lock contention, which hindered its expected benefits. Despite using Amdahl's Law to predict theoretical speedups, the actual results revealed diminishing returns as the number of threads or processes increased, especially in multithreading. This emphasizes that while parallelization can boost performance, synchronization overhead and resource contention significantly impact efficiency, and these factors should be carefully considered when choosing the right approach for a given ta