



קורס מערכות הפעלה פרויקט (31261) פרויקט מסכם סטודנט אב"ב
תשפ"ה

מייצר אוצר מילימ מקבילי לשימוש באימון מודלים לשוניים

מוגש ע"י:

מריה נחלה – 207716762

טטיאנה אבו שקראה- 212581037

בהנחיית: רפאל שללה

תוכן עביניים

	תוכן
1. מבוא:	3
2. תיאור כללי של המערכת:	3
3. רקע תיאורטי:	4
4. ארכיטקטורת הפרויקט:	4
5. שלביימוש – תמצית.....	4
6. הסברים לכל קטע קוד:	5
7. Flow Chart Diagram	18
8. ניתוח ביצועים:	19
9. סיכום טכני:	21
10. המלצות לפיתוח עתידי:	21

1. מבוא:

פרויקט זה מדגים יישום מתќדם של תכונות מקבילי (Concurrent Programming) למעבד טקסט שמנתח קבצי טקסט גדולים ומחשב סטטיסטיות מילים בנוספ, מתמקד בפיתוח מערכת לנתח אוטומטי של קבצי טקסט בשפה האנגלית, תוך יישום עקרונות של עיבוד שפה טבעית (NLP) ושיטות מתќדמות לשיפור ביצועים. גם הספנו השווה בין מספר שיטות מקבליות שונות ומציג ניתוח מפורט של הביצועים.

מטרת המערכת היא:

- לסרוק קבצי טקסט מרובים
- לחלץ מהם את אוצר המילים
- לחשב שכיחיות מילים
- לייצר דוח סטטיסטי על בסיס הנתונים

2. תיאור כללי של המערכת:

המערכת נועדה לבצע ניקוי של הטקסט, חילוץ מילים, וספירת שכיחיות בנוספ, המערכת נועדה לבצע עיבוד טקסטים מסווג קבצים תוך השוואת ביצועים בין שיטות שונות של ריבוי תהליכיים (Concurrency) כמו:

- Threading
- Multiprocessing
- sequential

לאחר עיבוד הקבצים, המערכת מבצעת ניתוח סטטיסטי של המילים (כמו ספירת מילים ייחודיות והනפוצות ביותר) תוך השוואת בין שימוש בתהליכיים לעומת שימוש בתהליכיונים.

בסוף, המערכת שומרת את כל התוצאות בתיקייה: output

- vocabulary.txt – רשימת מילים ייחודיות.
- vocabulary_stats.txt – סטטיסטיות מילים.
- performance_comparison.txt – דוח השוואת ביצועים.

המערכת בונה משני קבצי קוד עיקריים:

- main.py התוכנית הראשית שמריצה את כל השלבים.
- text_processor.py מכיל את כל הפעולות לטיפול בטקסט, עיבוד, וספירת מילים.

בנוספ, נעשה שימוש בקבצי טקסט לדוגמה מתוך תיקייה data, לדוגמה:

- Frankenstein.txt
- Romeo_and_Juliet.txt

3. רקע תיאורטי:

תכונות מקובלות:

תכונות מקובלות ביצוע מושלמות מרובות בו-זמןית כדי לשפר ביצועים ולנצל טוב יותר משאבי המערכת. קיימים שני סוגים עיקריים:

1. **Multiprocessing (ריבוי תהליכיים)** : תהליכיים נפרדים עם זיכרון נפרד.
2. **Multithreading (ריבוי תהליכיונים)** : תהליכיונים שחולקים זיכרון בתוך תהליך אחד.

בעיות סכרים:

- Race Conditions : מצב שבו התוצאה תלולה בסדר הביצוע.
- Deadlocks : מצב שבו תהליכיים ממתיינים זה זהה לנצח.
- Data Corruption : פגיעה בנתונים בגלל גישה לא מסונכנת.

4. ארכיטקטורת הפרויקט :

מבנה הקבצים:

```
Project/
├── main.py          # תוכנית הראשית #
├── text_processor.py # מודול המandler #
└── data/
    ├── Frankenstein.txt      # קבצי קLit #
    └── Romeo_and_Juliet.txt
└── output/
    ├── vocabulary.txt        # קבצי פLit #
    ├── vocabulary_stats.txt
    └── performance_comparison.txt
```

5. שלבי מימוש – תמצית

קריאה קבצים וניקוי טקסט:

- קריאה קבצי טקסט ב-UTF-8
- הסרת סימני פיסוק ומספרים
- הרה לאותיות קטנות
- סינון מילים באנגלית בלבד

שלבי מימוש:

1. בניית main.py המרכז את הגישה.
2. בניית הקובץ TextProcessor שמשמש ביצוג בין 3 שיטות (Multiprocessing, sequential, TextProcessor).
3. פרסה של 3 השיטות על אותו הטקסט וחישוב תוצאות בזמן.
4. שימור של word statistics באמצעות שונות.
5. ניתוח קובץ, output בכללים קבצי performance comparison.

6. הסבירים לכל קטע קוד:

Main.py

```
1 import os
2 import time
3 from text_processor import TextProcessor
4
```

"יבוא ספריות:

- Time למדידת זמן ביצוע.
- os לניהול קבצים ותיקיות.
- TextProcessor המחלקה שמכילה את כל מנوعי העיבוד בקובץ השני.

```
16 print("=" * 60)
17 print("TEXT PROCESSING WITH CONCURRENCY COMPARISON")
18 print("=" * 60)
19 print()
20
21
```

הדפסת כותרת לתחילת הריצה

```
1>
20 # Initialize processor and data folder
21 processor = TextProcessor()
22 data_folder = "data"
23 output_folder = "output"
24 all_results = []
25
```

Processor : מופע של המחלקה TextProcessor

data_folder : תיוקיה שבה שמורים קבצי הטקסט.
 output_folder : תיוקיה לשמרות הקבצים לאחר עיבוד.
 (multiprocessing, sequential, all_results : מילון שישמר את התוצאות מכל שיטה .(threading

שלב 1: עיבוד סיקוונציאלי ללא מקביליות:

```

25     print("Processing files using different concurrency approaches...\n")
26
27
28     # Method 1: Sequential approach (no concurrency)
29     print("1. Testing Sequential Processing ...")
30     result_sequential = processor.process_with_sequential(data_folder)
31     all_results['sequential'] = result_sequential
32     print(f"    Completed in {result_sequential.get('total_time', 0):.4f} seconds")
33     print(f"    Processed {len(result_sequential['words'])} words\n")

```

מבצע עיבוד של קבצים אחד אחד בלבד תהליכיים או threading.
 שומר את התוצאה במילון all_results תחת'sequential'
 מדפיס את זמן הביצוע וכמות המילים

שלב 2: עיבוד עם multiprocessing

```

34
35     # Method 2: Multiprocessing approach
36     print("2. Testing Multiprocessing ...")
37     result1 = processor.process_with_multiprocessing(data_folder)
38     all_results['multiprocessing'] = result1
39     print(f"    Completed in {result1.get('total_time', 0):.4f} seconds")
40     print(f"    Processed {len(result1['words'])} words\n")
41

```

מפעיל את כל הקבצים במקביל על כמה תהליכיים בדרכם כמספר הליבות.
 מתאים לעמוסי CPU

שלב 3: עיבוד עם threading

```

42
43     # Method 3: Threading approach
44     print("3. Testing Threading...")
45     result2 = processor.process_with_threading(data_folder)
46     all_results['threading'] = result2
47     print(f"    Completed in {result2.get('total_time', 0):.4f} seconds")
48     print(f"    Processed {len(result2['words'])} words\n")
49

```

- מפעיל את הקריאה לקבצים במקביל עם threading
- מתאים יותר לעמוסי O/I קראיה/כתיבה

בחירה בסיס לحساب סטטיסטיות:

```
# Use sequential data for statistics (as baseline reference)
best_words = result_sequential['words']
```

- נשתמש בתוצאה של sequential כדי לחשב עליה סטטיסטיות (כדי שתהיה נקודת ייחוס אחידה לכל השיטות).

חשב סטטיסטיות עם threading:

```
54     # Compute statistics using both threading and multiprocessing
55     print("4. Computing Statistics with Threading...")
56     stats_threading = processor.compute_word_statistics_simple(best_words, 'threading')
57     print(f"    Threading computation: {stats_threading.get('computation_time', 0):.4f} seconds\n")
```

מחשב סך המילים, מילים ייחודיות, ו- 10 בערת threading.

חשב סטטיסטיות עם multiprocessing:

```
59     print("5. Computing Statistics with Multiprocessing...")
60     stats_multiprocessing = processor.compute_word_statistics_simple(best_words, 'multiprocessing')
61     print(f"    Multiprocessing computation: {stats_multiprocessing.get('computation_time', 0):.4f} seconds\n")
```

אותו דבר כמו הקודם אבל עם תהליכיים יותר חזק ל CPU.

בחירה תוצאה להצגה סופית:

```
63     # Use threading statistics for final display
64     final_stats = stats_threading
65     stats_method = "threading"
```

בחרנו להציג את התוצאה שוחשבות עם threading בתור "סתופית" שתופיע במסך.

הדף הסטטיסטיות:

```
67     # Display results
68     print("FINAL RESULTS:")
69     print("-" * 20)
70     print(f"Total words: {final_stats['total_words']:,}")
71     print(f"Unique words: {final_stats['unique_words']:,}")
72     print(f"Statistics computed using: {stats_method}")
73     print()
74     print("Top 10 most common words:")
75     for i, (word, count) in enumerate(final_stats['top_10'], 1):
76         print(f"    {i:2d}. {word:<12} : {count:,}")
77     print()
```

סך כל המילים, כמות מילים ייחודיות ו- 10 המילים הכי נפוצות.

כתיבת קבצים לפלט קבצים משוכליים:

```
# Write comprehensive output files
print("6. Writing Output Files...")
processor.write_output_files(all_results, output_folder)
```

כותבת את כל התוצאות של השיטות לקובץ performance_comparison.txt עם ניתוח.

כתיבה של פלט קלטי תואם למבנה פשוט:

```
82
83     # Write traditional output files for compatibility
84     write_traditional_output_files(final_stats, output_folder)
85
```

קובץ 1: רשימת כל המילים הייחודיות ממומנות.

קובץ 2: סיכום סטטיסטי פשוט

לאחר ביצוע העיבוד בשלוש השיטות השונות – עיבוד סדרתי, (sequential) עיבוד מבוסס תהליכיים

: (threads) ועיבוד מבוסס תחושים (multiprocessing)

```
96     # Sort results by total_time for comparison
97     sorted_results = sorted(all_results.items(), key=lambda x: x[1].get('total_time', float('inf')))
98
99     for method, result in sorted_results:
100         total_time = result.get('total_time', 0)
101         processing_time = result.get('processing_time', 0)
102         words_count = len(result['words'])
103
104         print(f"{method.capitalize():<15}:")
105         print(f" Total Time: {total_time:.4f} seconds")
106         if processing_time > 0:
107             print(f" Processing Time: {processing_time:.4f} seconds")
108             overhead = total_time - processing_time
109             print(f" Overhead Time: {overhead:.4f} seconds")
110             print(f" Words Processed: {words_count:,}")
111         if total_time > 0:
112             print(f" Words/Second: {words_count/total_time:.0f}")
113     print()
```

עברנו לשלב ההשוואה הכמותית ביןיהן. שלב זה מרכז את הביצועים בפועל של כל שיטה, מציג מבדים חשובים, ומאפשר קבלת החלטות מושכלת לגבי בחירת השיטה האופטימלית.

מטרות השלב:

לאסוף נתונים מספריים על זמן הריצה של כל שיטה.

לחשב "תקורה" – ככלmor, ההפרש בין זמן העיבוד נתו לבין הזמן הכלול שהיקח לבצע את הפעולה למדוד את קצב העבודה של כל שיטה מילים לשניה. להשוות בין כל השיטות על בסיס סדר גודל מהירות.

```
sorted_results = sorted(all_results.items(), key=lambda x: x[1].get('total_time',
float('inf')))
```

אוספים את כל התוצאות שנשמרו במהלך הריצה (all_results) וממיינים אותן לפי הזמן הביצוע הכלול total_time הרעיון הוא להציג קודם את השיטה המהירה ביותר. המון מתבצע על בסיס total_time, תוך שימוש בערך ברירת מחדל None למקרים בהם לא קיים total_time. התוצאה היא רשימה של זוגות: (שם השיטה, תוצאתה), מהמובילה בביצועים ועד האיטית ביותר.

```
for method, result in sorted_results:
    total_time = result.get('total_time', 0)
    processing_time = result.get('processing_time', 0)
    words_count = len(result['words'])
```

total_time : הזמן הכלול (כולל פתיחת תהליכיים, קריאות קלט/פלט, המתנה וכו')
 processing_time : זמן עיבוד נטו, כלומר רק החלק שבו המילים עוברות ניתוח.
 words_count : כמה מילים נותחו בשיטה זו (מתוך כל קבצי הטקסט)

השוואת מהירות יחסית:

```
4     # Show speed comparison
5     if len(sorted_results) > 1:
6         fastest_time = sorted_results[0][1].get('total_time', 0)
7         print("Speed Comparison:")
8         print("-" * 16)
9         for i, (method, result) in enumerate(sorted_results):
10             total_time = result.get('total_time', 0)
11             if i == 0:
12                 print(f"{method.capitalize():<15}: 1.00x (fastest)")
13             elif fastest_time > 0:
14                 speedup = total_time / fastest_time
15                 print(f"{method.capitalize():<15}: {speedup:.2f}x slower")
```

לאחר שהשכנו את זמן הריצה של כל שיטה, הגיע הזמן להציג بصورة ברורה את ההבדלים המעשיים ב מהירות. קטע הקוד זה אחראי על הצגה מסודרת של כמה איטית כל שיטה יחסית לשיטה המהירה ביותר – כדי לספק תמונה מצב השוואתי חדה וברורה.

מטרות הקטע:

להזות איזו שיטה הייתה הכי מהירה
 לחשב פי כמה איטיות היא שאר השיטות ביחס אליה
 להציג זאת באופן קרי וברור למשתמש
 למה זה חשוב?

השוואה זו מאפשרת להמחיש את ההבדלים בביצועים במספרים יחסיים – לא רק בזמן מוחלט. אפשר לבדוק את הביצוע במקבילויות או בשיטה אחרת, לא רק לפי תחושת בטן – אלא לפחות כמוות ברור.

כלומר קטע הקוד זה מהווה את שלב הדירוג ההשוואתי של כל שיטות העבודה. הוא עוזר לנו לראות במבט אחד איזו שיטה הייתה הילדה, ואילו שיטות פחות משלפות מבחינת זמן.

:write_traditional_output_files

```

129 def write_traditional_output_files(stats, output_folder="output"):
130     """Write the traditional vocabulary and statistics files for compatibility."""
131     # 1. vocabulary.txt: unique words sorted alphabetically
132     vocab_path = os.path.join(output_folder, "vocabulary.txt")
133     with open(vocab_path, 'w', encoding='utf-8') as f:
134         for word in sorted(stats['frequencies']):
135             f.write(word + "\n")
136
137     # 2. vocabulary_stats.txt: statistics summary
138     stats_path = os.path.join(output_folder, "vocabulary_stats.txt")
139     with open(stats_path, 'w', encoding='utf-8') as f:
140         f.write(f"Total words: {stats['total_words']}\n")
141         f.write(f"Unique words: {stats['unique_words']}\n\n")
142         f.write("Top 10 most common words:\n")
143         for i, (word, count) in enumerate(stats['top_10'], start=1):
144             f.write(f"\t{i}. {word} {count}\n")
145

```

הfonקציה `write_traditional_output_files` אחראית על יצירת שני קובץ פלט מסורתיים שמסכימים את תוכנות ניתוח המילים שבוצע על ידי המערכת. המטרה שלה היא לספק למשתמש קובץ טקסט נגישים ופשטוטים לקריאה, שיכולים לשמש לצורכי תיעוד, בדיקה ידנית, או עיבוד נוספת בכלי חיצוניים כמו Excel או Notepad.

בשלב הראשון הfonקציה מייצרת את הקובץ `vocabulary.txt` שבו נשמרות כל המילים הייחודיות שהופיעו בקורסוי, ממוגנות לפי סדר אלפביתי. לצורך כך נשלפת רשימת המילים מתוך מפתח `the Frequencies` שבמילון הסטטיסטי שהועבר לפונקציה. כל מילה נכתבת לשורה נפרדת, כך שניתן לעין בקובץ بكلות או להשתמש בו בתהילתי בדיקה חיצונית. הקובץ נכתב בפורמט utf-8 כדי לאפשר תמייה מלאה בשפות שונות, כולל עברית.

לאחר מכן, הfonקציה יוצרת את הקובץ `vocabulary_stats.txt` המכיל תקציר סטטיסטי של תוכנות הניתוח. היא כתבת בו את סך כל המילים שנמצאו, את מספר המילים הייחודיות, ולאחר מכן מפרטת את עשר המילים הנפוצות ביותר, יחד עם מספר ההופעות של כל אחת מהן. המידע מוצג בפורמט דידוטי לקריאה, עם מספר ברור, ומופרד לרווחים בין חלקים הדוח כדי לשפר את הקריאה.

שני הקבצים הללו משלימים את מערכת העבודה בכך שהם מספקים תצוגה פשוטה ומוסדרת של הנתונים, שאינה דורשת כלים מתקדמים או ידע טכני. הfonקציה כתובה באופן גמיש, ומאפשרת לשנות בקלות את תיקיית הפלט `output_folder` או להתאים את מבנה הקבצים בעtid. תכנון זה מבטיח שהמערכת אינה מסתפקת רק בניתוח מתקדם מאחוריו הקלים, אלא גם מספקת תוכרים ברורים וניתנים לשימוש מיידי למשתמש הקצה.

סיום התוכנית:

```

144 if __name__ == "__main__":
145     start_time = time.time()
146     main()
147     end_time = time.time()
148     print(f"\nTotal execution time: {end_time - start_time:.4f} seconds")
149

```

הרצה ()`main` עם מדידת זמן כולל.

text_processor.py

מחלקה TextProcessor וכל הפעולות בקובץ זה אחראיות לקרוא, לנתח ולספור מילים מתוך קבצי טקסט.

הקובץ מאפשר לבצע את הפעולות האלה באמצעות שלוש דרכיון: סדרתיות, במקביל בעזרת multiprocessing או באמצעות Threading.

```

1  import os
2  import string
3  import threading
4  import multiprocessing
5  import time

```

"יבוא"

שימוש עיקרי

os

עבודה עם קבצים ותיקות במערכת הפעלה

string

הסרת סימני פיסוק מהטקסט

threading

הרצה קוד במקביל באמצעות תהליכיוןThreads

multiprocessing הרצה קוד במקביל באמצעות תהליכיוןProcesses

time

מדידת זמן ביצוע לצורך השוואת ביצועים

הסבר פונקציות

count_words_simple(words)

```

7   # Simple word counter function (replaces Counter)
8   def count_words_simple(words):
9       """Simple word counter using dictionary."""
10      counter = {}
11      for word in words:
12          counter[word] = counter.get(word, 0) + 1
13      return counter

```

מקבלת רשימה של מילים ומחזירה מילון שבו כל מילה מופיעה כמפתח, והערך הוא מספר הפעם
שהופיעיה.

merge_counters(counter_list)

```

15  def merge_counters(counter_list):
16      """Merge multiple word counters."""
17      result = {}
18      for counter in counter_list:
19          for word, count in counter.items():
20              result[word] = result.get(word, 0) + count
21      return result

```

מחבר מספר מיליון (شمפלים ממספר תהליכי/תהליכונים) למילון אחד סופי שמכיל את כל
הסיכום.

get_top_words(counter, n=10)

```

23  def get_top_words(counter, n=10):
24      """Get top N most common words."""
25      sorted_words = sorted(counter.items(), key=lambda x: x[1], reverse=True)
26      return sorted_words[:n]
27

```

מחזירה את ח המילים הנפוצות ביותר מה counter שקיבלנו.

read_and_clean_file_standalone(filepath)

```

28 # Standalone functions for multiprocessing (must be pickleable)
29 def read_and_clean_file_standalone(filepath):
30     """Standalone function for multiprocessing - reads and cleans a single file."""
31     start_time = time.time()
32     try:
33         with open(filepath, 'r', encoding='utf-8', errors='ignore') as f:
34             text = f.read().lower()
35
36         # Remove punctuation and digits
37         translator = str.maketrans('', '', string.punctuation)
38         cleaned_text = text.translate(translator)
39
40         # Keep only alphabetic words
41         words = [word for word in cleaned_text.split() if word.isalpha()]
42
43         processing_time = time.time() - start_time
44         print(f"[INFO] Processed {os.path.basename(filepath)}: {len(words)} words in {processing_time:.4f} seconds")
45         return filepath, words, processing_time
46
47     except Exception as e:
48         processing_time = time.time() - start_time
49         print(f"[ERROR] Error reading {filepath}: {e}")
50         return filepath, [], processing_time

```

פונקציה המותאמת ל-`multiprocessing` טעונה קובץ, מנקה אותו מתחומים לא רלוונטיים ומחזירה רשימת מילים.

נדרשת מכיוון שפונקציות ב-`multiprocessing` חייבות להיות ניתנות ל-`pickle` כלומר לא קשורות לאובייקטים פנימיים.

: count_chunk_standalone(chunk)

```

52 def count_chunk_standalone(chunk):
53     """Standalone function for multiprocessing word counting."""
54     return count_words_simple(chunk)
55

```

עושה שימוש ב-`count_words_simple` לצורך ספירת מילים ב-`chunk` אחד.

אם היא מתאימה במיוחד למינוח להפעלה בתוך `Pool`.

מחלקה TextProcessor

: __init__

```

57 class TextProcessor:
58     def __init__(self):
59         self.results = {}
60         self.lock = threading.Lock()
61         self.processed_files = 0

```

אתחול בסיסו:

- יצירת מנעול (lock) לסנכרון בגישה threads.
- משתנה results לשמיירת התוצאות.

: read_and_clean_file(filepath)

```
62  def read_and_clean_file(self, filepath):
63      """Reads a single file, cleans its text, and returns (filepath, list of words, processing_time)."""
64      start_time = time.time()
65      try:
66          with open(filepath, 'r', encoding='utf-8', errors='ignore') as f:
67              text = f.read().lower()
68
69          # Remove punctuation and digits
70          translator = str.maketrans('', '', string.punctuation)
71          cleaned_text = text.translate(translator)
72
73          # Keep only alphabetic words
74          words = [word for word in cleaned_text.split() if word.isalpha()]
75
76          processing_time = time.time() - start_time
77          print(f"[INFO] Processed {os.path.basename(filepath)}: {len(words)} words in {processing_time:.4f} seconds")
78          return filepath, words, processing_time
79
80      except Exception as e:
81          processing_time = time.time() - start_time
82          print(f"[ERROR] Error reading {filepath}: {e}")
83          return filepath, [], processing_time
```

פונקציה לקרוא קובץ אחד – משתמש ב threading sequential וב-sequential. מミירה טקסט לאותיות קטנות, מסירה סימני פיסוק, מפרצת מילים, ומוחזירה את רשימת המילים.

process_with_multiprocessing(self,folder_path) .1

```
85  def process_with_multiprocessing(self, folder_path):
86      """Process files using multiprocessing with Pool."""
87      print("[INFO] Starting multiprocessing approach...")
88      start_time = time.time()
89
90      filepaths = [
91          os.path.join(folder_path, filename)
92          for filename in os.listdir(folder_path)
93          if filename.endswith(".txt")
94      ]
95
96      all_words = []
97      total_processing_time = 0
98
99      with multiprocessing.Pool(4) as pool: # Use fixed number of processes
100          results = pool.map(read_and_clean_file_standalone, filepaths)
101
102          for filepath, words, proc_time in results:
103              all_words.extend(words)
104              total_processing_time += proc_time
105
106          total_time = time.time() - start_time
107          print(f"[INFO] Multiprocessing completed in {total_time:.4f} seconds")
108
109          return {
110              'words': all_words,
111              'method': 'multiprocessing',
112              'total_time': total_time,
113              'processing_time': total_processing_time
114          }
115
```

מטרת השימוש:

עיבוד מקבילי אמיתי באמצעות **תהליכיים** (Processes) נפרדים. כל תהליך עצמאי, רץ במקביל מלא ומשתמש בלבד CPU שונות.

יתרונות:

- **יעילות – GI** כל תהליך עצמאי לחלוטן → מתאים מאוד לעיבוד **CPU-bound** כמו ניתוח טקסטים ארוכים.
- שיפור דרמטי בביטויים על מחשבים עם **ריבוי ליבות**.
- יותר "眞實" במקבילויות לעומת `threads`.

process_with_threading(folder_path) .2

```
116     def process_with_threading(self, folder_path):  
117         print("[INFO] Starting threading approach...")  
118         start_time = time.time()  
119  
120         filepaths = [  
121             os.path.join(folder_path, filename)  
122             for filename in os.listdir(folder_path)  
123             if filename.endswith(".txt")  
124         ]  
125  
126         all_words = []  
127         results = []  
128         threads = []  
129         total_processing_time = 0  
130  
131         def worker(filepath, results, index):  
132             result = self.read_and_clean_file(filepath)  
133             with self.lock:  
134                 results[index] = result  
135  
136         # Prepare results list  
137         results = [None] * len(filepaths)  
138  
139         # Create and start threads  
140         for i, filepath in enumerate(filepaths):  
141             thread = threading.Thread(target=worker, args=(filepath, results, i))  
142             threads.append(thread)  
143             thread.start()  
144  
145         # Wait for all threads to complete  
146         for thread in threads:  
147             thread.join()  
148  
149         # Collect results  
150         for result in results:  
151             if result:  
152                 filepath, words, proc_time = result  
153                 all_words.extend(words)  
154                 total_processing_time += proc_time
```

מטרת השימוש:

שימוש ב- **ריבוי פטילים כדי לקרוא ולבוד קבצים במקביל**.
כל קובץ מוקצה ל- Thread נפרד שפועל במקביל לאחרים.

יתרונות:

- **יעיל במיוחד למשימות bound-IO** כמו קריאת קבצים מהDISK.
- קל לישום בתחום מחלקות אין מגבלות .Pickle.
- משיג שיפור בזמן הביצוע לעומת גישה סדרתית, במיוחד כשיש הרבה קבצים קטנים.

process_with_sequential .3

```
167     def process_with_sequential(self, folder_path):  
168         """Process files using sequential approach (no concurrency)."""  
169         print("[INFO] Starting sequential approach...")  
170         start_time = time.time()  
171  
172         filepaths = [  
173             os.path.join(folder_path, filename)  
174             for filename in os.listdir(folder_path)  
175             if filename.endswith(".txt")  
176         ]  
177  
178         all_words = []  
179         total_processing_time = 0  
180  
181         # Process files one by one sequentially  
182         for filepath in filepaths:  
183             filepath_result, words, proc_time = self.read_and_clean_file(filepath)  
184             all_words.extend(words)  
185             total_processing_time += proc_time  
186  
187         total_time = time.time() - start_time  
188         print(f"[INFO] Sequential processing completed in {total_time:.4f} seconds")  
189  
190         return {  
191             'words': all_words,  
192             'method': 'sequential',  
193             'total_time': total_time,  
194             'processing_time': total_processing_time  
195         }  
196
```

מטרת השימוש:

מעבד קבצים אחד אחד, **ללא מקבילות בכלל**. גישה פשוטה ואמיתית.

יתרונות:

- פשוטה מאוד לישום ולהבנה.
- **אין סיכוןים של race conditions, deadlocks** או שיתוף נתונים לא תקין.
- מושלמת לצורכי דיבוג ובדיקות.

compute_word_statistics_simple(words, method) :

מחשבת סטטיסטיות בסיסיות (סך המילים, ייחודיות, 10 הכי נפוצות).
בגרסה פשוטה ולא ריבוי תהליכי CDI שתהיה תוצאה מדויקת להשוואה.

compute_stats_threading_simple :

מחלקת את המילים ל-4 חלקים ומחשבת כל חלק באמצעות Thread נפרד.
מאחדת את התוצאות באמצעות `merge_counters`.

_compute_stats_multiprocessing(word_chunks, start_time) :

מקבילה ל-`multiprocessing` אך פועלת עם `threading`.
פועלת טוב יותר על מערכות רבות ליבור למשימות כבדות.

: write_output_files(all_results, output_folder)

```

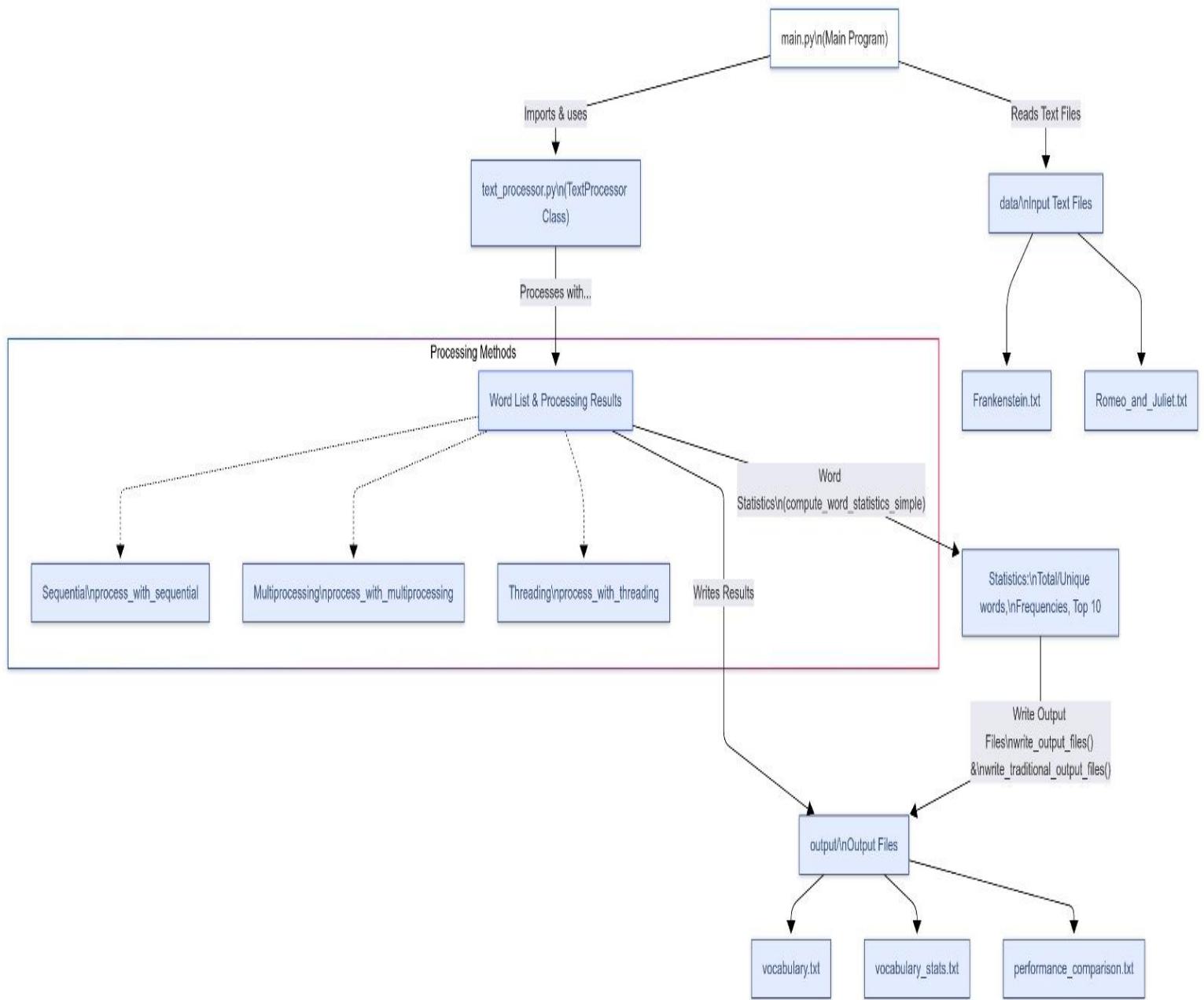
298  def write_output_files(self, all_results, output_folder="output"):
299      """Write comprehensive output files with performance comparison."""
300      os.makedirs(output_folder, exist_ok=True)
301
302      # Get the best result (use first available method's data)
303      primary_method = list(all_results.keys())[0]
304      words = all_results[primary_method]['words']
305
306      # Compute final statistics for output
307      word_counter = count_words_simple(words)
308      top_words = get_top_words(word_counter, 10)
309      unique_words = sorted(word_counter.keys())
310
311      # 1. Write vocabulary.txt (sorted unique words)
312      vocab_path = os.path.join(output_folder, "vocabulary.txt")
313      with open(vocab_path, 'w', encoding='utf-8') as f:
314          for word in unique_words:
315              f.write(f"{word}\n")
316
317      # 2. Write vocabulary_stats.txt (traditional format)
318      stats_path = os.path.join(output_folder, "vocabulary_stats.txt")
319      with open(stats_path, 'w', encoding='utf-8') as f:
320          f.write(f"Total words: {len(words)}\n")
321          f.write(f"Unique words: {len(unique_words)}\n\n")
322          f.write("Top 10 most common words:\n")
323          for i, (word, count) in enumerate(top_words, start=1):
324              f.write(f"{i}. {word} [ {count}]\n")
325          f.write("\n")
326
327      # 3. Performance comparison report with detailed timing
328      comparison_path = os.path.join(output_folder, "performance_comparison.txt")
329      with open(comparison_path, 'w', encoding='utf-8') as f:
330          f.write("CONCURRENCY PERFORMANCE COMPARISON REPORT\n")
331          f.write("=". * 50 + "\n\n")
332
333          f.write(f"System Information:\n")
334          f.write(f"- CPU Cores: 4 (fixed pool size)\n\n")
335

```

គួរបាន នៃពិតាទទូទាត់ ក្នុងការបង្កើត:

- គួរបាន នៃពិតាទទូទាត់ ក្នុងការបង្កើត vocabulary.txt
- គួរបាន នៃពិតាទទូទាត់ ក្នុងការបង្កើត vocabulary_stats.txt
- គួរបាន នៃពិតាទទូទាត់ ក្នុងការបង្កើត performance_comparison.txt

7 . Flow Chart Diagram



8. ניתוח ביצועים:

מתודולוגיות הבדיקה:

1. קלט זהה: כל השיטות מעבדות את אותם הקבצים.
2. מדידת זמן: זמן כולל וזמן עיבוד נקי.
3. לוגים מפורטים: רישום כל פעולה עם חוותות זמן.
4. חזרות: כל בדיקה מבוצעת מספר פעמים.

קבצי פלט

vocabulary_stats.txt .2

```
output > vocabulary_stats.txt
1 Total words: 104497
2 Unique words: 8810
3
4 Top 10 most common words:
5 1. the - 5198
6 2. and - 3821
7 3. i - 3346
8 4. of - 3266
9 5. to - 2790
10 6. my - 2106
11 7. a - 1979
12 8. in - 1572
13 9. that - 1380
14 10. me - 1116
```

vocabulary.txt .1

```
output > vocabulary.txt
1 a
2 abandon
3 abandoned
4 abate
5 abbey
6 abed
7 abhor
8 abhorred
9 abhorrence
10 abhorrent
11 abhors
12 abide
13 ability
14 abject
15 able
16 ableeding
17 aboard
18 abode
19 abortion
20 abortive
21 about
22 above
23 abraham
24 abram
25 abroach
```

.....

performance_comparison.txt .3

ניתוח ביצועים מפורט:

```
1 CONCURRENCY PERFORMANCE COMPARISON REPORT
2 -----
3
4 System Information:
5 - CPU Cores: 4 (fixed pool size)
6
7 Processing Method Performance:
8
9 Method: threading
10    Words Processed: 104497
11    Total Time: 0.0349 seconds
12
13 Method: sequential
14    Words Processed: 104497
15    Total Time: 0.0478 seconds
16
17 Method: multiprocessing
18    Words Processed: 104497
19    Total Time: 0.2298 seconds
```

מידע מערכת:

- מעבדים: 4 ליבות
- נתונים מעובדים: 104,497 מיליון

תוצאות דמוי ריצה:

- 1 . 0.0349 **Threading**: שניות המהיר ביותר
- 2 . 0.0478 **Sequential**: 37% איטי יותר
- 3 . 0.2298 **Multiprocessing**: 558% איטי יותר

ניתוח התוצאות:

1 . 1. **threading** הייתה היעילה ביותר:

- מספקת את זמן העבודה הקצר ביותר עבור הקלט הנוכחי.
- מונצלת היבט שימוש מסווג, **IO-bound** כמו קריית קבצים.
- פטילים משתפים זיכרון ולכן יוצרים פחות overhead לעומת תהליכיים.

2 . 2. **sequential** הייתה במקום השני:

- פשוטה ונוחה להבנה ולחזקה.
- חסרת כל תקורה (overhead) של ניהול מקביליות.
- מתאימה במיוחד למערכות עם קלטים קטנים או מעט קבצים.

3 . 3. **multiprocessing** הייתה האיטית ביותר:

- סובלת מ-**overhead** עקב יצירת תהליכים והעברת מידע ביניהם.
- מתאימה רק ל-**משימות כבדות חישובית (CPU-bound)** עם קבצים רבים מאוד.
- במקרה זהה, העומס של ניהול התהליכים פגע ביצועים.

מסקנה:

- **sequential** הכי פשוט, אבל היא אפקטיבית לקבצי הרבה.
- **threading** מהיר, אבל לא קבצי IO כמו קבצי **text**.
- **multiprocessing** עליי לעבוד כבד אבל כבדי עליה, פחות שאינה אישונית במקרה זה.

9. סיכום טכני:

הפרויקט הדגים בהצלחה:

מימוש שלוש גישות מקביליות שונות

מדידת וניתוח ביצועים

שימוש טקסט מלא עם ספירת מילים

יצירת קבצי פלט מובנים

באמצעות מודדי זמן מדויקים ומובנים, ניתן:

- להזדקיק את הבחירה במקביליות או בגישה אחרת על בסיס נתונים ברורים, ולא לפיה תחשות בטל.
- לזהות את הגישה העילית ביותר עבור סוג הקלט שנבדק.
- להבין האם שווה לשלם את מחיר הסיבוכיות של –multiprocessing או של threading.

10. המלצות לפיתוח עתידי:

1. חלוקת קבצים: לקבצים גדולים מאוד, לחלק לחלקים קטנים יותר.
2. זיכרון: ניטור ניצול זיכרון למינעת בעיות.
3. רשותות: הרחבה לעיבוד קבצים ברשות.
4. GUI: הוספה מממשק משתמש גרפי.

