K Most Popular Words
Programming Assignment 1

COEN 242 Big Data (Spring 2023)

Instructor Prof. Kishore Pusukuri
Santa Clara University

Shweta Deshmukh (1650082) Xiaoxiao Jiang (1651657)

The Problem Statement:

Design and Implement an efficient java/python/scala code (or any language you prefer) to determine Top K most frequent/repeated words in a given dataset (example: K = 10). The objective here is to obtain the result with the least possible execution Time (or with the best performance on your computer).

Computer Configurations:

Macbook - M1 - Pro, 16gb ram, 8 cores

Programming language:

Python

Result (By Multi Processes):

DataSet	Result (K = 10)	Running time (sec) (average of 10 runs)
50MB	[('would', 22320), ('us', 18579), ('new', 15675), ('economic', 15624), ('one', 14374), ('countries', 13784), ('political', 12853), ('even', 12698), ('also', 12270), ('global', 11976)]	1.17
300MB	[('european', 318743), ('mr', 210690), ('would', 181912), ('also', 180118), ('commission', 172783), ('must', 156856), ('president', 152134), ('union', 130344), ('states', 130270), ('member', 126301)]	4.53
2.5GB	[('said', 2616266), ('one', 949555), ('would', 917212), ('new', 852822), ('also', 727934), ('last', 700735), ('people', 688828), ('us', 670794), ('mr', 659562), ('year', 654244)]	34.23
16GB	[('said', 16983038), ('would', 5829109), ('one', 5827854), ('new', 5619251), ('also', 4618231), ('us', 4602724), ('people', 4302078), ('last', 4096906), ('year', 4011803), ('two', 3964144)]	224.62

Title: Analyzing the Efficiency of Different Approaches to Count Top K Word Frequencies in Large Text Files

Introduction:
The task is to create a Python program to count the top k word frequencies in a given text, which might be too large to fit in memory. In this report, we will analyze three different methods for tackling this problem, namely: brute force, multithreading, and multiprocessing. We will then discuss their complexities, advantages, and disadvantages, including the Python's Counter implementation, the Global Interpreter Lock (GIL), and the overhead of thread and process switching.

- Python's **Counter Class**: The Counter class in Python is a container that holds the count of each unique element present in a sequence, such as a list or a string. It is part of the collections module and is useful in a variety of applications, such as data analysis and processing, text mining, and more.

- **most_common()** is a method available in Python's collections.Counter class, which returns a list of the k most common elements and their counts from the counter object, where k is a positive integer argument.

- **pool.map()** is a method from the multiprocessing module in Python that allows you to apply a function to a list of arguments in parallel, using a process pool. **map_file()** is the function that you want to apply to the list of arguments.

- In **map_file()** we are calculating the word count by processing the text to lowercase and then incrementing the counter if it is not a stop_word.

- **merge_counts()** is implemented to merge the counter objects in a single Counter object.

Methodology:

Brute Force: This method utilizes Python's built-in Counter class to count word frequencies. To avoid memory issues, we read the text in fixed-size chunks (512 MB) and update the word counter accordingly. The Counter class in Python is implemented using a dictionary, which has an average case complexity of O(1) for insertion and retrieval operations. The Counter class was chosen for its simplicity and efficient implementation.

Multithreading: This method divides the text into smaller files and employs the divide-and-conquer (MapReduce) strategy. A thread pool is used to spawn multiple threads, each responsible for counting word frequencies in its designated file. Each thread holds a separate counter, and the results are merged in the end. We have implemented divide conquer strategy

Multiprocessing: Similar to the multithreading approach, this method also adopts the MapReduce strategy. However, instead of using threads, it utilizes separate processes to handle the counting tasks.

Time Comparison (sec)

	Brute Force	Multi Thread	Multi Processes
50MB	3.52	3.67	1.17
300MB	19.57	19.17	4.53
2.5GB	93.53	131.32	34.23
16GB	835.23	997.53	224.62

Results and Analysis:

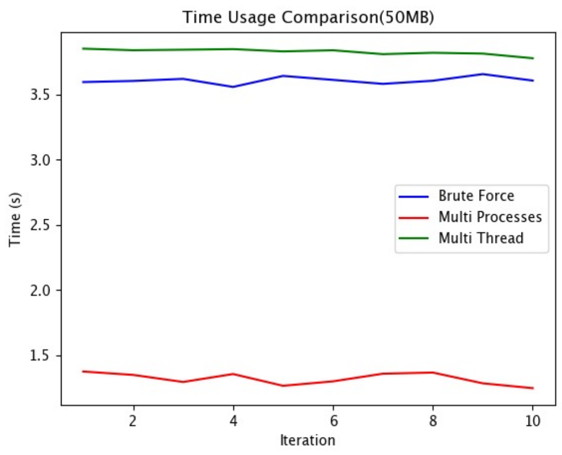
For smaller text files, both multithreading and multiprocessing approaches were slower than the brute force method due to the overhead of thread and process switching. However, when tested on larger text files (e.g., 300 MB), the multiprocessing approach outperformed the brute force method, while the multithreading approach performed the worst.

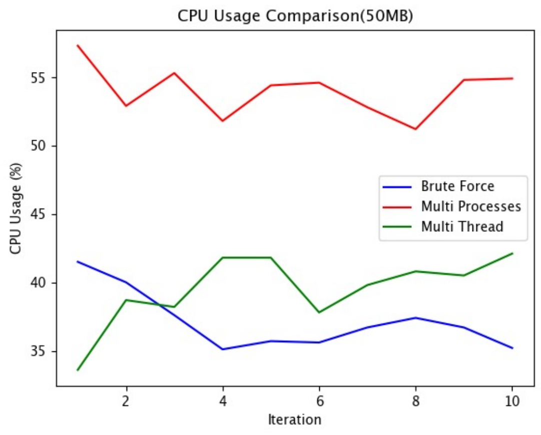
This performance discrepancy can be attributed to Python's **Global Interpreter Lock (GIL)**, which prevents multiple threads from executing Python bytecodes simultaneously in the same process. As a result, multithreading in Python can be inefficient for CPU-bound tasks, such as counting word frequencies.

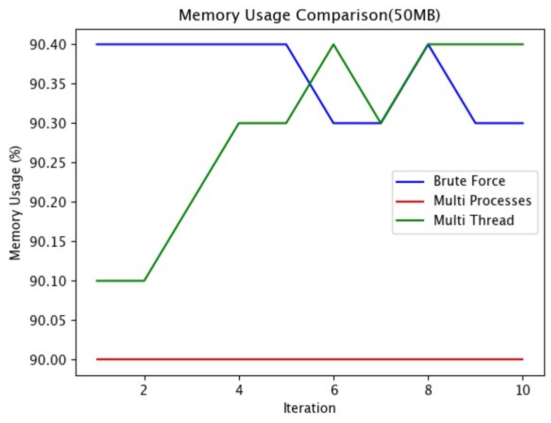
The multiprocessing approach, on the other hand, avoids the GIL limitation by using separate processes, each having its own interpreter and memory space. This allows for better parallelism and improved performance on large text files.

Graphs:

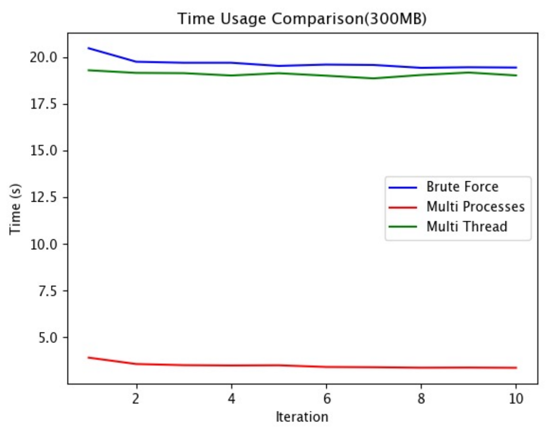
- For 50MB datafile :

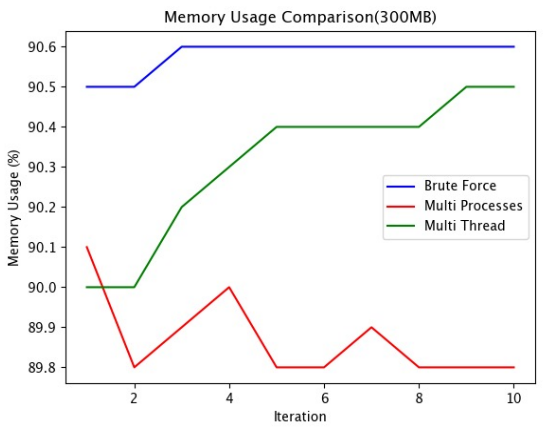


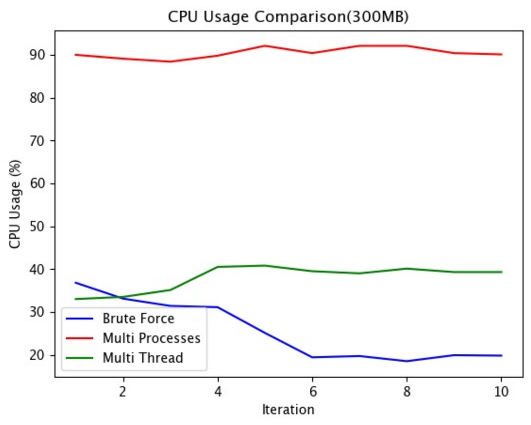


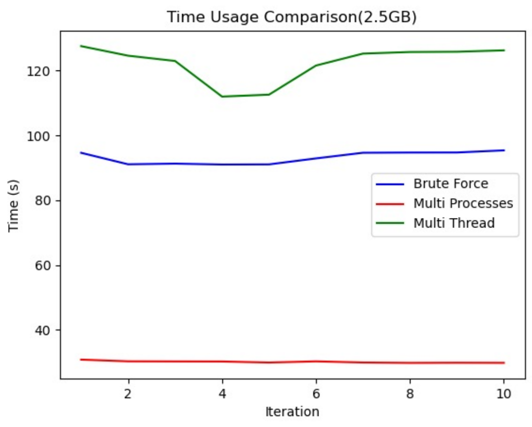


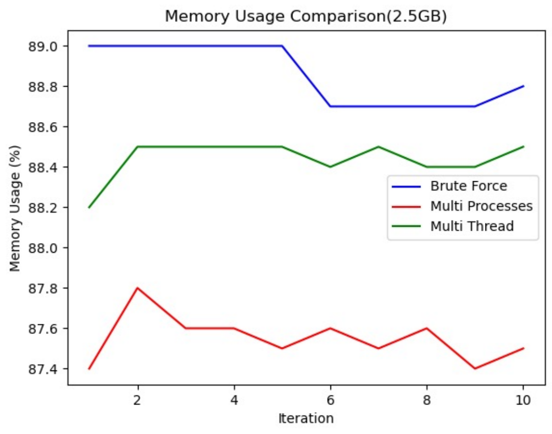
Here, as you can see, Brute Force approach of algorithm is efficient as compared to the Multi Process.

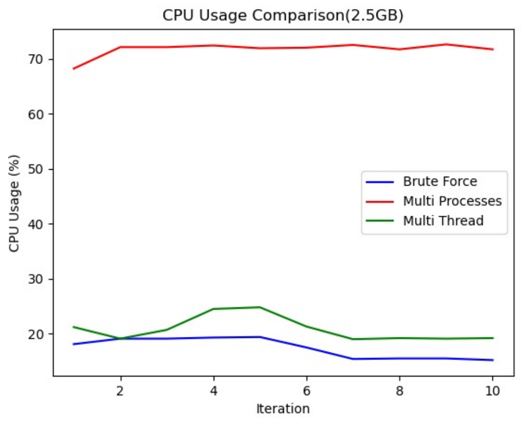












For a 300MB dataset, the brute force method and the multi thread method already take about the same amount of time. The performance of the multi processes method is still far better than the other two methods. For a 2.5GB dataset, multi thread takes more time than brute force. We speculate that this is due to the presence of the aforementioned GIL in python