1. The Problem Statement

The objective of this report is to design and implement an efficient Python code to determine the top K most frequent/repeated words in each dataset (example: K = 10) and present a detailed analysis of the performance through different metrics such as running time, speedup, CPU utilization, memory usage, etc. The report covers the analysis of the code execution on each of the three input data files separately.
The primary focus of this project is to obtain the result with the least possible execution time (or with the best performance on your computer) by skipping the "stop" words such as 'the' - provided in the stop words list available at stop_words_link.
The result should preserve case sensitivity, meaning that words like "Title" and "title" are considered two different words. The input dataset contains only English alphabets, white spaces, and hyphen-separated words, i.e., "a-z," "A-Z," "\s."
Initially, the code was tested on a smaller dataset provided at small_dataset. Then, the code was executed on three larger datasets of different sizes provided in a zip file available at zip_file_link. The zip file consists of three files with only English words, i.e., data_300MB.txt - a text file of size 300MB, data_2.5GB.txt - a text file of size 2.5GB, and data_16GB.txt - a text file of size 16GB.
This report presents a detailed analysis of why a particular algorithm or data structure has been used to solve this problem. The report also covers the presentation of the results obtained from the execution of the code and the performance metrics calculated.

2. System Configuration

- Processor: M2 processor used.
- Cores: 10 cores.
- Memory: 16 GB RAM available on the system.
- Storage Type: SSD storage with 512 GB storage space
- Storage size available: Available disk space.
- Operating System: macOS 16.1
- Programming Language: Python3
- Compiler and Runtime Environment: Python 3 interpreter

3. Experimental Process:

Our experiment aimed to design and implement an efficient algorithm to find the top K most frequent words in a dataset. We experimented with various approaches using different data structures and algorithms and evaluated their performance on three input data files of different sizes. We implemented our approaches in Python and measured different metrics such as running time, speedup, CPU utilization, and memory usage to analyze their performance.

3.1. Approach 1: Hashmap File Singlethread

This approach reads the whole dataset file using a single thread and stores the count of each word in a hashmap. Stop words are removed, and remaining words' frequency is sorted in reverse to return the top K words. Time complexity is O(nlogn) for sorting, and space complexity is O(n). This approach is efficient for smaller datasets, but not for larger ones due to sorting time complexity. Hashmap provides constant time complexity for accessing and updating key values.

3.2. Approach 2: Counter Chunk Singlethread

This approach reads data in chunks, counts the occurrence of words using regular expressions, and stores the word count in a Counter object. The most_common() function finds the top K words. This approach is memory-efficient and eliminates the need for sorting the whole dataset. Time complexity is O(n), and the code includes functions to read stop words and print top K words and performance statistics. Reading data in chunks reduces memory usage, making this approach more efficient for large datasets. Using a Counter object allows for fast and memory-efficient word counting.

3.3. Approach 3: Counter Heapq Singlethread

This approach uses Counter to count word frequency and a predefined set of stop words. The program reads data line by line, lowercases, and splits each line into words using regular expressions. Frequency of each non-stop word is incremented in the Counter object. The top k words with the highest frequency are found using heapq.nlargest, which has a time complexity of O(nlogk), and printed with performance statistics. This approach is similar to Approach 2 but may be more efficient for large k values.

3.4. Approach 4: Defaultdict Chunks Heapq Singlethread

We read the data in chunks in a single thread and stored the word count in the defaultdict. We used the heapq.nlargest function to return the top K words. This approach was similar to the previous one, but we used defaultdict instead of the counter. In this approach, we read the data in chunks in a single thread and stored the word count in the defaultdict. We used the heapq.nlargest function to find the top K words. This approach is similar to the previous one, but we used defaultdict instead of the counter. The advantage of defaultdict is that it automatically initializes the value of a key to a default value, which is useful when we need to know the exact keys beforehand.

3.5. Approach 5: <u>Defaultdict Chunks Multiprocess</u>

This approach uses defaultdict and heapq to find top K frequent words in a file, read in chunks in a single thread. The chunks are processed using multiprocessing Pool and process_chunk function to return a dictionary of word counts. The dictionaries are merged using defaultdict, and heapq.nlargest returns top K frequent words. The code first reads stop words, creates a list of chunks, and applies the process_chunk function to each chunk in parallel. The time complexity is O(N log K), where N is the number of words in the file, and space complexity is O(N). Performance statistics are printed using log.get_top_words_heapq and log.print_statistics functions. This approach is suitable for processing large datasets and finding the most frequent words efficiently.

3.6. Approach 6: <u>Counter Chunk Multithread</u>

This approach uses multithreading to read and process data in chunks. The file is divided into chunks, and each chunk is read by a separate thread, where the word count is stored in a counter. The most_common function retrieves the top K words, and stop words are removed during processing. Time complexity is O(n/p), where n is the input file size, and p is the number of threads. The space complexity is proportional to the number of unique words. Functions read_stop_words, read_chunk, and process_chunk read and process data. The process_data function prints the top K words and performance statistics, creating a list of chunks and processing each in a separate thread. Main function sets the number of top words to find and calls the process_data function for different chunk sizes. This approach is suitable for efficiently processing large datasets on multi-core CPUs.

3.7. Approach 7: <u>Counter Lines Multithread</u>

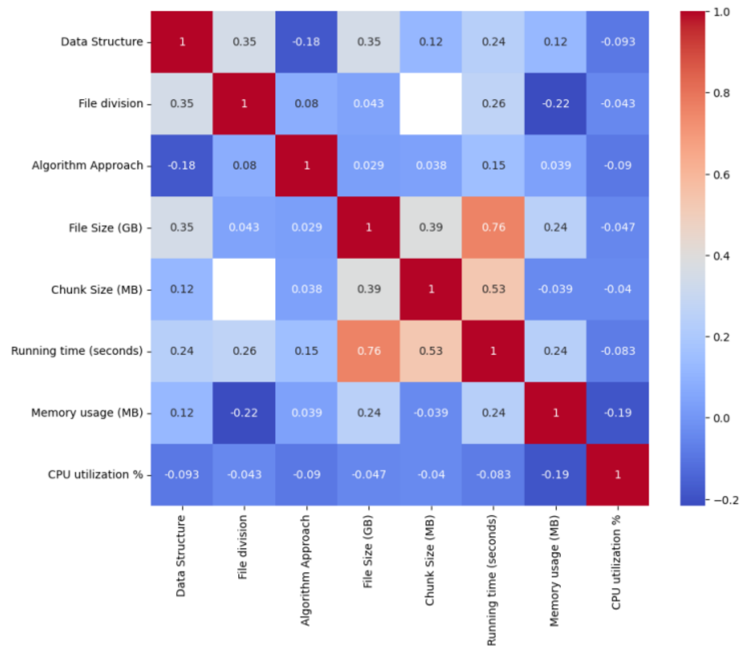
This approach creates a separate thread for each line in the data file and aggregates the local counters of all threads to count the words. The most common words are selected and printed. Time complexity is O(n), and space complexity is O(k+m), where k is the number of top words to find, and m is the number of unique words. However, creating and managing too many threads result in high CPU utilization and slow performance, making this approach less efficient for larger datasets. This approach aims to optimize the multithreading approach used in Approach 6 but could be more efficient with further improvements.

4. Evaluation

To evaluate the performance of these approaches, we measured various metrics such as running time, speedup, CPU utilization, and memory usage and maintained detailed <u>logs</u> of each execution. The <u>logs.csv</u> file contains a log of the experiments conducted to determine the most efficient approach for finding the top k most popular words in a given dataset. Each row in the file represents a particular configuration of the algorithm approach, data structure, file division, and input file size used to run the experiment. The file includes various performance metrics for each configuration tested, such as running time, memory usage, and CPU utilization.

4.1. Correlation Matrix

A correlation matrix is a table that shows the correlation coefficients between many variables. It is used to study the relationship between variables, which can help to identify patterns, trends, and associations among the data. In a correlation matrix, each cell shows the correlation between two variables, with values ranging from -1 to 1. A value of 1 indicates a perfect positive correlation, a value of -1 indicates a perfect negative correlation, and a value of 0 indicates no correlation between the variables.



Observation:

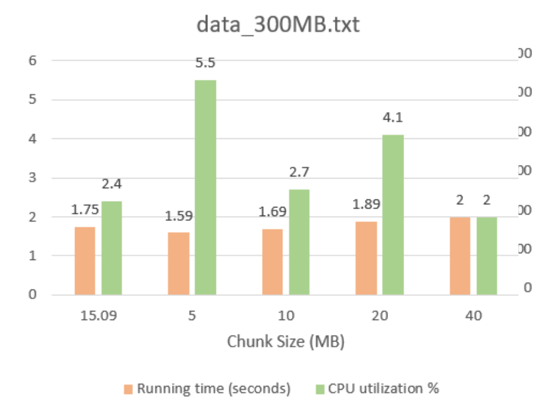
- File Size (GB) and Running time (seconds) have a strong positive correlation of 0.76, meaning that as the file size increases, the running time tends to increase as well.
- Memory usage (MB) and CPU utilization % have a strong negative correlation of -0.19, meaning that as memory usage increases, CPU utilization tends to decrease.
- We can see that Memory usage (MB) has a weak positive correlation with File Size (GB) and Running time (seconds), which suggests that as the file size or running time increases, the memory usage may also increase slightly.
- Similarly, CPU utilization % has a weak negative correlation with Algorithm Approach, suggesting that more complex algorithms may result in lower CPU utilization.

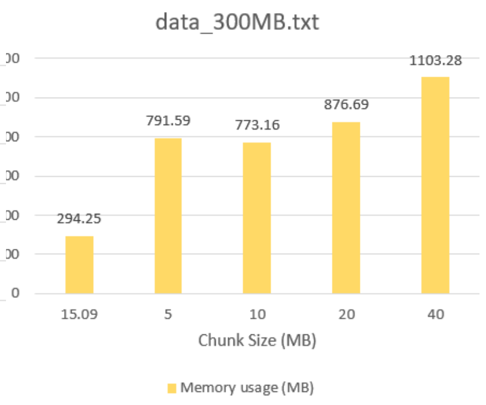
4.2. Performance Analysis

4.2.1. 300 MB Dataset

Smaller input data set competes the efficient algorithm to be clear winner whereas, large dataset aim to optimize the read / write operation.

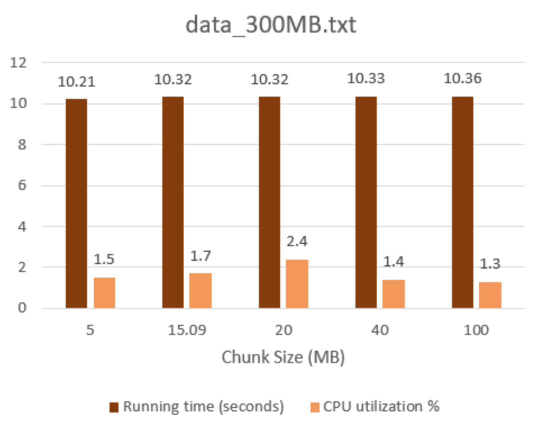
4.2.1.1. Single Thread + Default Dict + heapQ

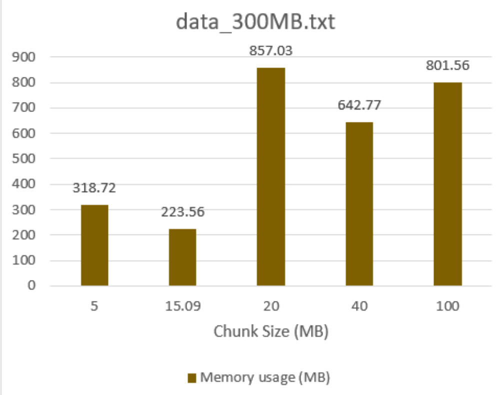




Analysis: As mentioned, the most algorithm results performing best in every algorithm due to smaller dataset however much smaller dataset of 5 MB surpassed CPU use but consumed higher memory as well.

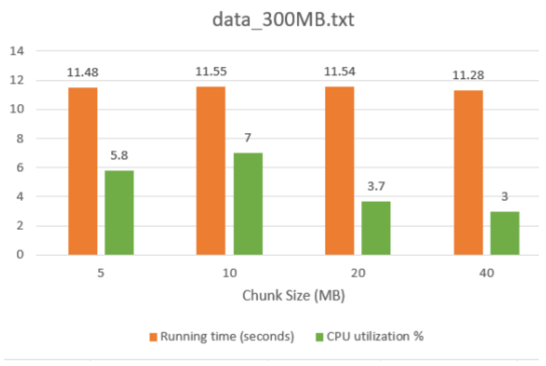
4.2.1.2. Single Thread + Default Dict + Counter

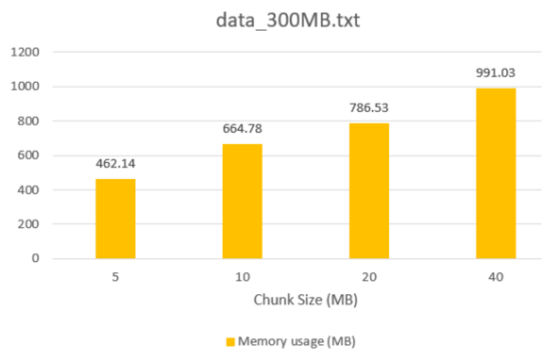




Analysis: This algorithm result was consistent across runtime and CPU use, but application memory usage varied distinctly. Surprisingly, 15 MB chunk size performed the best but 10MB and 20MB chunk size resulted in worst performance.

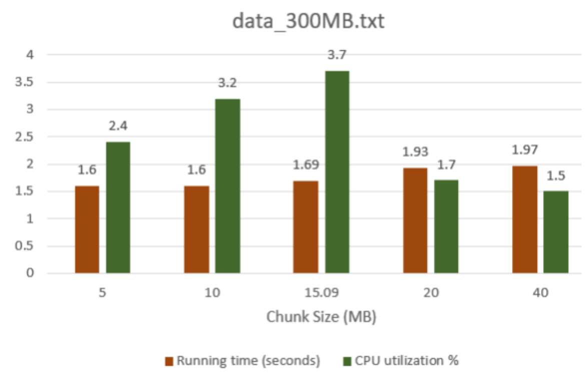
4.2.1.3. Multithread + Counter

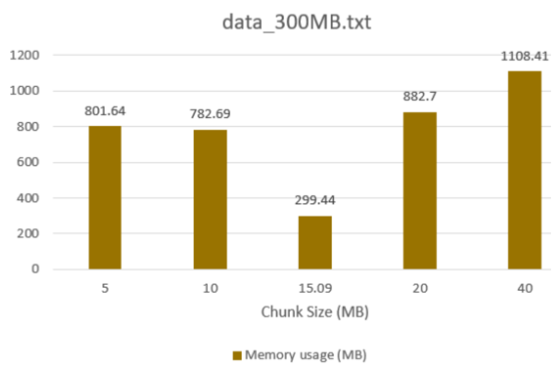




Analysis: Converting the above algorithm from single thread to multi thread produces predicted result of increase in memory usage as chunk size increases. This due to face that each thread of smaller chunk size consumes less heap memory than the bigger chunk sized thread.

4.2.1.4. Multiprocess + Default Dict



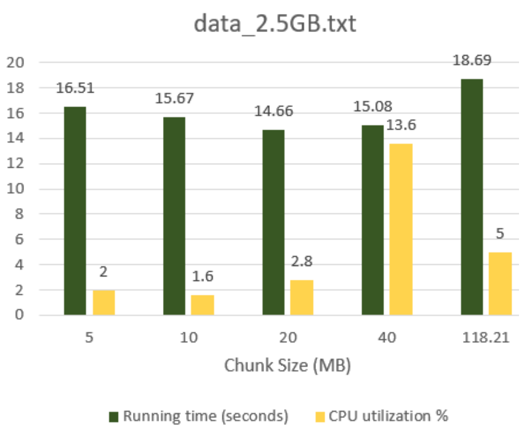


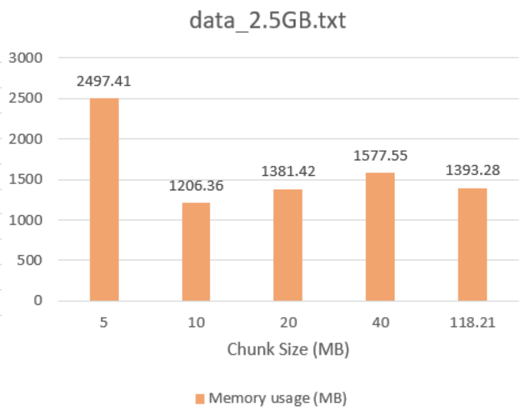
Analysis: This is one of the interesting graphs where using the efficient read and process algorithm lets chunk sized chosen by the program to be more efficient than the predefined ones. One of the reasons could be that letting program choose to read and process size is better since the inbuilt algorithm complexity for smaller dataset can be optimized by program irrespective of available main memory or CPU

4.2.2. 2.5 GB Dataset

As the input data size changes, the running time across different algorithms is near about same making application memory and CPU utilization main priority.

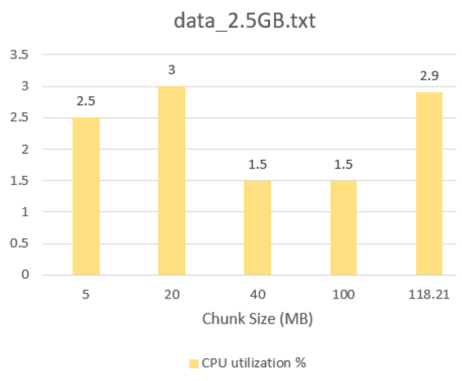
4.2.2.1. Single Thread + Default Dict + heapQ

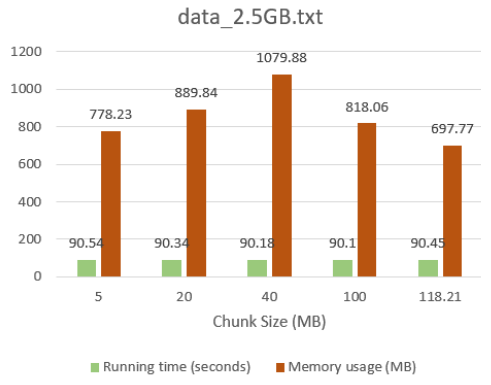




Analysis: 10 MB chunk size uses least memory but has poor CPU utilization. Thus aiming for resource use, 40 MB chunk suits best.

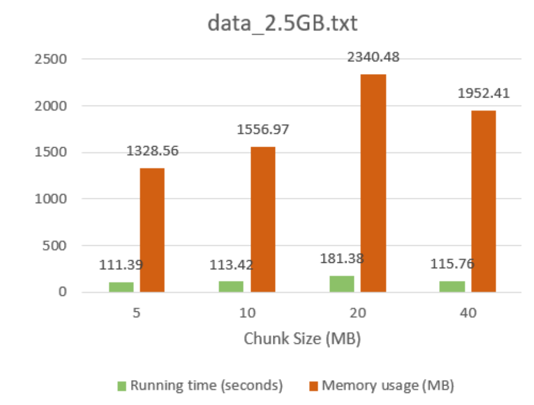
4.2.2.2. Single Thread + Default Dict + Counter

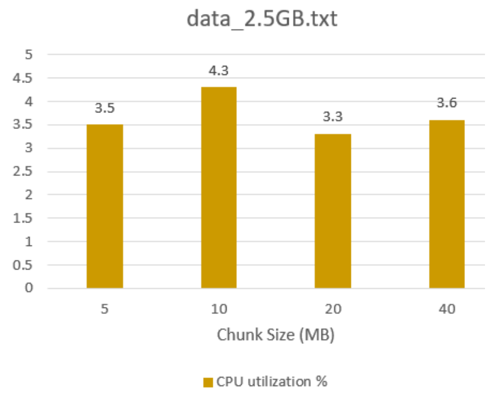




Analysis: Changing the data structure makes 40MB chunk size to produce worst result and letting application decide on chunk size of 118 MB is best performing. All of this is how efficient the data structure can keep results within itself.

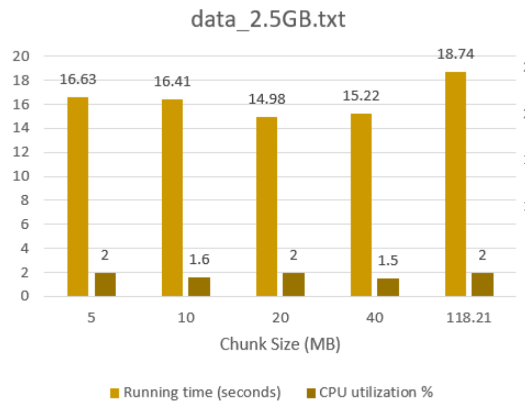
4.2.2.3. Multithread + Counter

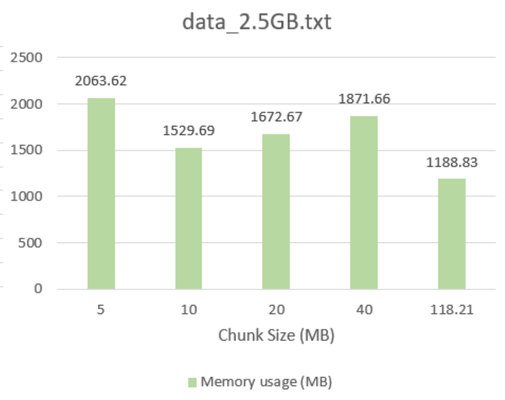




Analysis: Implement multithreading to the earlier algorithm makes smaller chunk size of 5MB best performing due to fact that higher chunk sizes would require more static memory for each thread than smaller ones.

4.2.2.4. Multiprocess + Default Dict

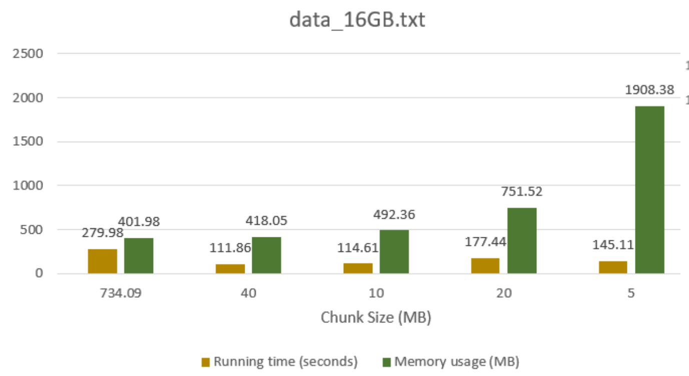


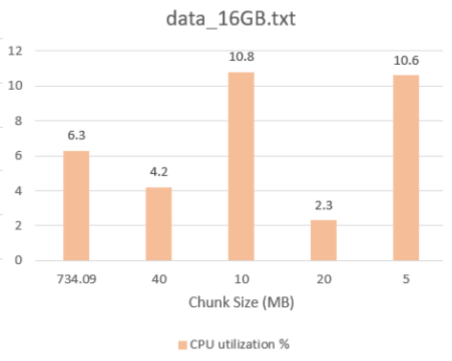


Analysis: Tweaking efficient data structure to utilize the multi-processing CPU to yield 2- Mb chunk size as best performing algorithm.

4.2.3. 16 GB Dataset

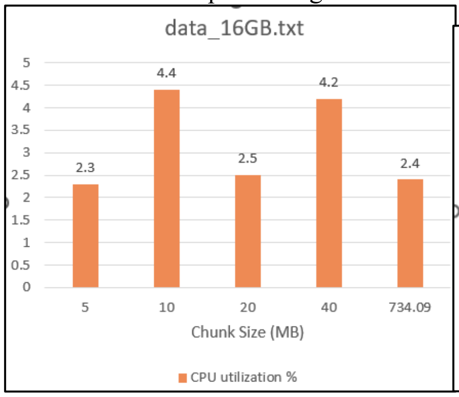
4.2.3.1. Single Thread + Default Dict

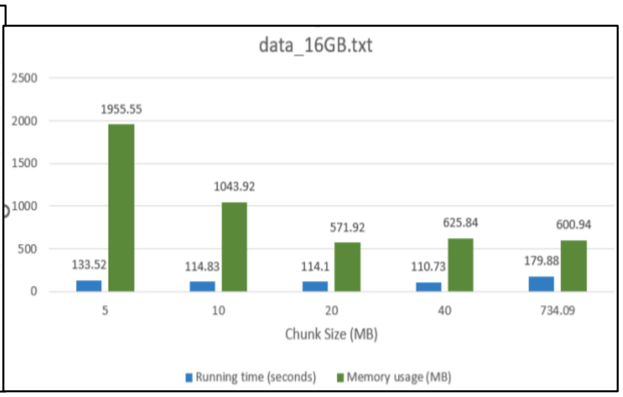




Analysis: Letting program decide on file read chunk size based on actual file size helps to reduce the overall application memory but is 2x slower than choosing 10Mb chunk size which has best running time with 23% more application memory. Moreover, 10Mb chunk utilizes 40% better CPU resources as well.

4.2.3.2. Multiprocessing + Default Dict





Analysis: Just by tweaking the algorithm to multi-process the read operations worsen the 10mb read chunk and makes program chosen chunk size the best performing.

5. Conclusions

Data Structure	Algorithm Approach	Filename	Chunk Size (MB)	Running time (seconds)	Memory usage (MB)	CPU utilization %
defaultdict heapq	singlethread	data_300MB.txt	15.09	1.75	294.25	2.4
defaultdict heapq	singlethread	data_2.5GB.txt	40	15.08	1577.55	13.6
defaultdict heapq	singlethread	data_16GB.txt	10	114.61	492.36	10.8
counter	singlethread	small_50MB_dataset.txt	2.29	1.74	70.22	1.2
defaultdict	multiprocess	data_300MB.txt	15.09	1.69	299.44	3.7
counter	singlethread	data_300MB.txt	20	10.32	857.03	2.4
counter	multithread	data_300MB.txt	5	11.48	462.14	5.8
defaultdict	multiprocess	data_2.5GB.txt	20	14.98	1672.67	2
counter	singlethread	data_2.5GB.txt	118.21	90.45	697.77	2.9
counter	multithread	data_2.5GB.txt	5	111.39	1328.56	3.5
defaultdict	multiprocess	data_16GB.txt	20	114.1	571.92	2.5

Best =
best for 50 MB = 2.29 MB chunk size counter
300 MB default dict heapq single thread chunk size = 15.09 mb
2.5 Gb default dict heapq single thread chunk size 40 Mb
16 gb default dict heapq single thread chunk size 10 Mb

conclusion best algo default dict heapq single thread