Log Analysis using Distributed Sub-string Matching

Shivangi Agarwal (2020113011) | Bhumika Joshi (2022121006)

April 22, 2024

1 Introduction

In this project we have explored substring-matching in a distributed setting. Though our implementation accommodates use-cases other than log analysis, our main goal was to use distributed substring-matching algorithms for log analysis.

Companies often generate large volumes of log data that need to be analyzed for monitoring, debugging and security auditing purposes. Though logs are traditionally not too large to warrant the use of distributed computing, companies and organisations are generating millions of logs, which are stored in a distributed manner. When searching for a particular log, a distributed sub-string matching algorithm would come in handy.

The codes for Naive, Rabin-Karp, Knuth Morris Pratt, and Aho-Corasick are implemented in a way that accounts for the size of the actual pattern being larger than the chunk size. For log analysis, this will usually not be necessary. Most keywords are short while the amount of text to look through is huge.

1.1 Map-Reduce

Map-Reduce is a framework for processing large chunks of data on clusters. Apache develops the open-source implementation of map-reduce we will be using on top of the Hadoop File System. A Map-Reduce system is usually composed of 3 independent operations

1. Map: Each worker node applies the map function to the local data and writes the output to temporary storage. A master node ensures that only one copy of the redundant input data is processed.

2. Shuffle: Worker nodes redistribute data based on the output keys (produced by the map function), so all data belonging to one key is located on the same worker node.

3. Reduce: Worker nodes now process each group of output data, per key, in parallel.

1.2 Solution

There are two major steps for the implementation of these algorithms. The basic intuition behind distributing the computation remains the same and is discussed in details.

1.2.1 Pre-processing

The input file is divided into chunks of data. Similarly, the pattern to match is divided into chunks. For m text chunks and n pattern chunks, each m is passed through the mapper paired with each n. This takes O(nm) time.

1.2.2 Map-Reduce

Given each pair, (text_chunk, pattern_chunk), the mapper runs the given algorithm. If a match is found, it would emit a key, value pair where the key is index of the text_chunk at which the pattern_chunk would have to start for an overlap. The value will be i, where pattern_chunk is the i^{th} chunk of the whole pattern.

1.2.3 Basic Intuition

when preparing the text_chunks, a certain part of the previous chunk is included in the next one. This is because patterns can span across the boundaries of chunks. Without overlapping chunks, matches that occur at the dividing point between two chunks might be missed. Overlapping ensures that patterns crossing chunk boundaries are not split and can be fully detected.

2 Algorithms

In this section we have discussed the basic ideas behind the algorithms we implemented.

2.1 Naive Method

The mapper runs the actual sub-string matching for each pair of text_chunk and pattern_chunk. It compares character by character, from index i to i + pattern_chunk_size. In case of no match, the next comparison starts from the $(i+1)^{th}$ index.
Time Complexity: $O(n^2)$
Space Complexity: O(m)
It is generally not recommended for large datasets due to its time complexity,

2.2 Knuth Morris Pratt (KMP) Algorithm

The mapper runs the actual sub-string matching for each pair of text_chunk and pattern_chunk. The partial match table, i.e. pi table is built for each pair of text chunk and pattern chunk. The key idea behind KMP is that after each mismatch, we know enough information that would allow us to skip the characters we know would match. This requires a level of pre-processing when comparing from any index i. The number of indices to skipped is maintained separately.
Time Complexity: O(m + n)
Linear time complexity translates to reduced total time in a distributed system, though efficiency decreases if the pattern is present across two text_chunks.

2.3 Boyer Moore Algorithm

The Boyer Moore Algorithm is a combination of *Bad Character Heuristics* and *Good suffix Heuristics*. It is similar to KMP, but maintains two conditions that can independently decide the number of indices to skip. Boyer-Moore takes the maximum of the two. It starts matching from the last character of the pattern and stores the last occurrence of each alphabet to *reposition* the pattern_chunk.
Time Complexity: O(m + mn)
It is extremely efficient for short patterns and when matches are infrequent, though efficiency decreases if the pattern is present across two text_chunks. In the worst case, especially with many repetitive patterns, it can approach O(mn) complexity.

2.4 Rabin Karp Algorithm

The mapper runs the actual sub-string matching for each pair of text_chunk and pattern_chunk. It calculates the hash value for the pattern_chunk and then computes the hash values of length, *pattern_chunk_size*, from text_chunk. If the hashes match, it performs a direct string comparison to verify the match [done because of hash collisions].
Time Complexity: O(m + mn) [O(m + n) average]
After hashing, checks for potential matches can be done in constant time for each text window. Implementing a rolling hash function in a distributed setting is complicated.

2.5 Aho-Corasick Algorithm

The mapper runs the actual sub-string matching for each pair of text_chunk and pattern_chunk. A trie is created of all the words in the input array. It can handle several words at once. Aho-Corasick is a kind of dictionary-matching algorithm. We traverse through the text input using the trie, and follow failure transitions when there's no match. When a match is found, we record it and continue.

Time Complexity: O(m + n + z)
(where z is no. of occurrences of the sub-string and q is the length of the alphabet)
The complexity of the algorithm is linear in the length of the text plus the total length of all patterns. It requires considerable pre-processing to create the trie and failure functions which increases in a distributed setting.

3 Code Structure

The Github repository contains the following files and folders, along with the README.

3.1 make_input_ready.py

This is the file used to pre-process that data as explained in 1.2.1

3.2 generate_inputs.py

This is used to generate a large input file containing logs

3.3 finalize_output.py

This file makes the output presentable, removing irrelevant data

3.4 Algorithm Folders

Each algorithm is implemented in the folder with the corresponding name.

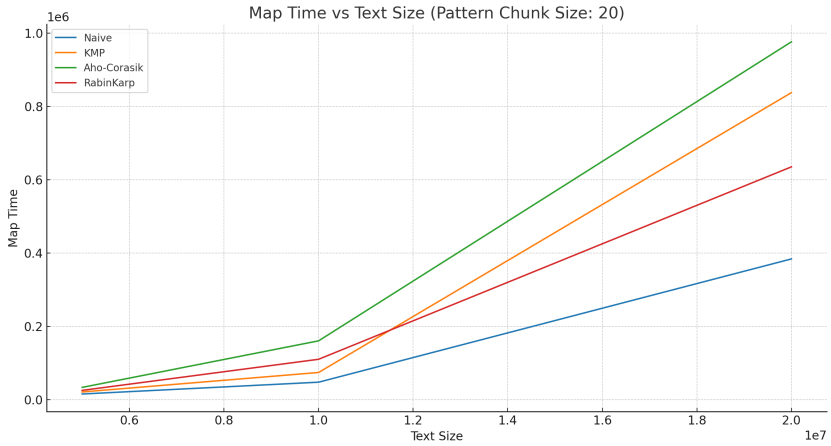
3.5 input_logs

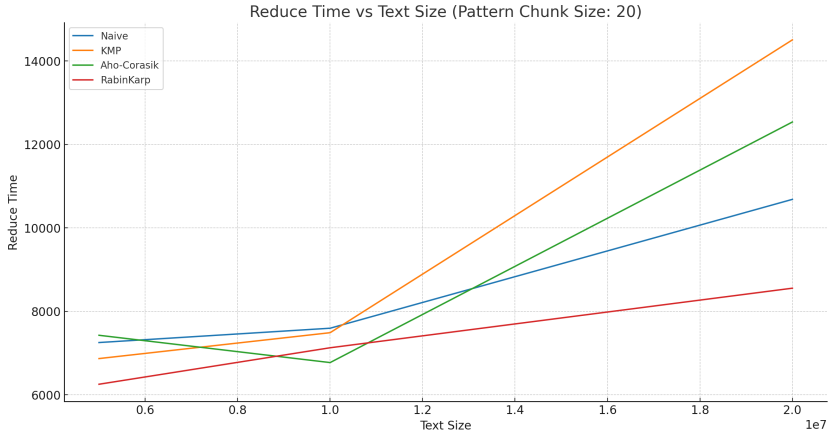
This folder contains some test cases ranging from 50 logs to 2k logs.

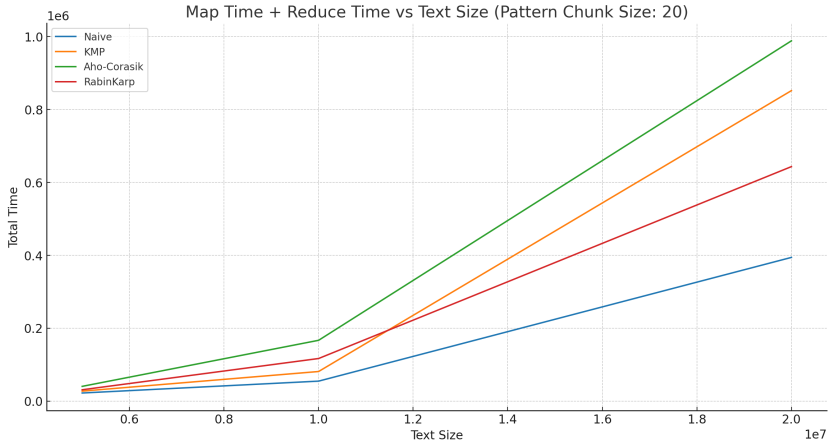
4 Analysis

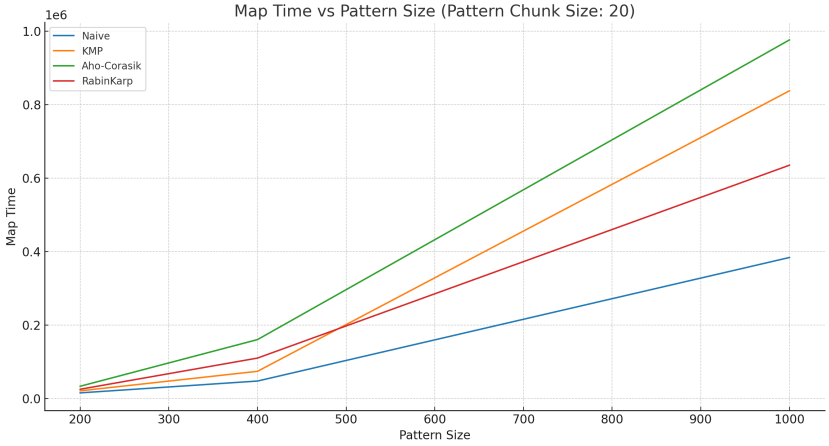
In this section, we have included some plots for varying input conditions.

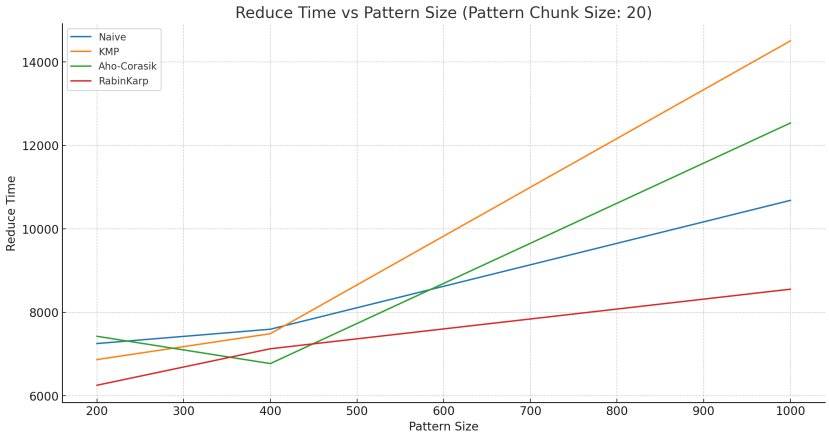
4.1 Text Chunk Size =100, Pattern Size =20

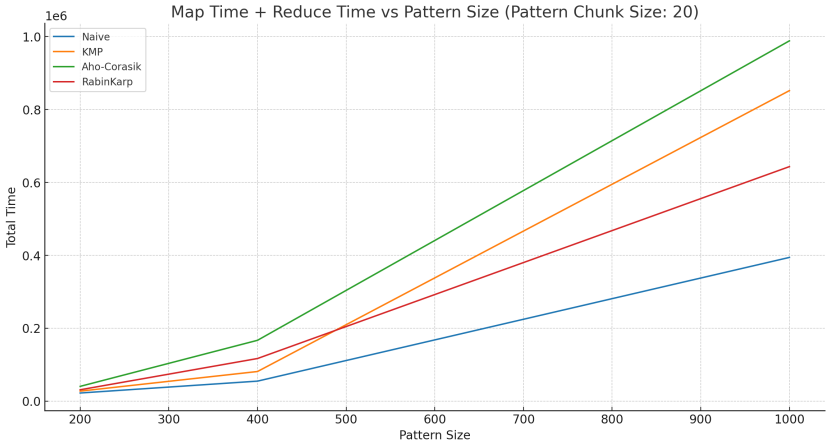




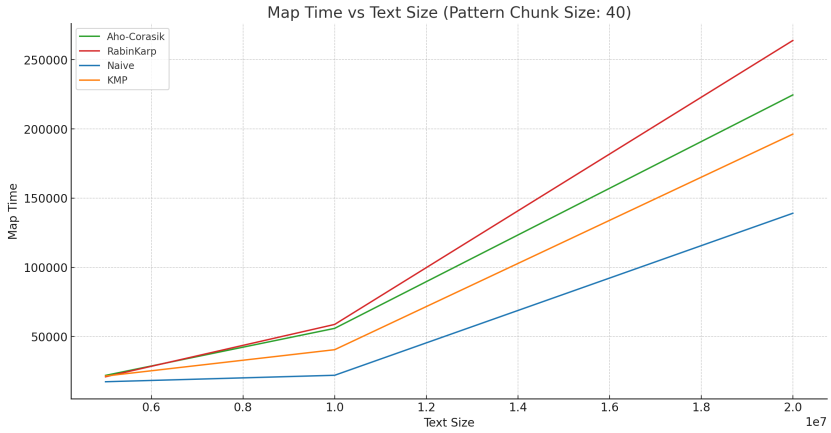


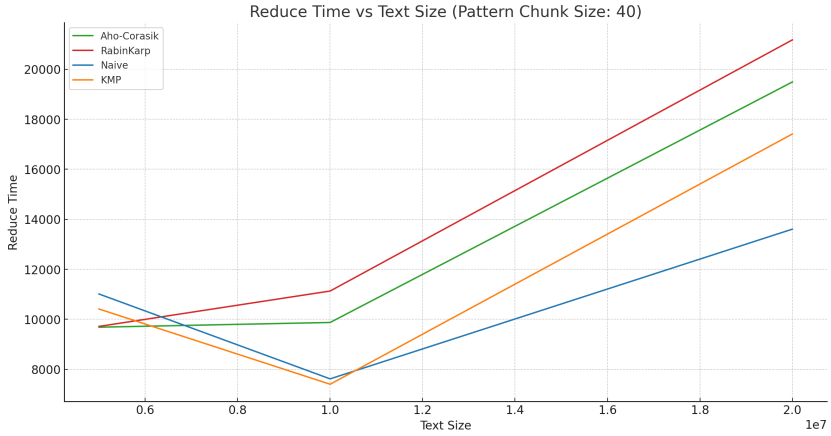

Map Time vs Pattern Size (Pattern Chunk Size: 20)
1e6
Naive
KMP
Aho-Corasik
RabinKarp
Map Time
Pattern Size

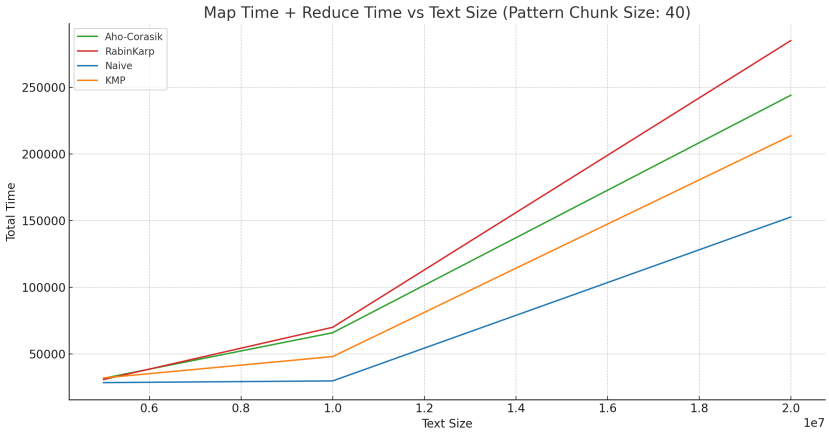

Reduce Time vs Pattern Size (Pattern Chunk Size: 20)
Naive
KMP
Aho-Corasik
RabinKarp
Reduce Time
Pattern Size

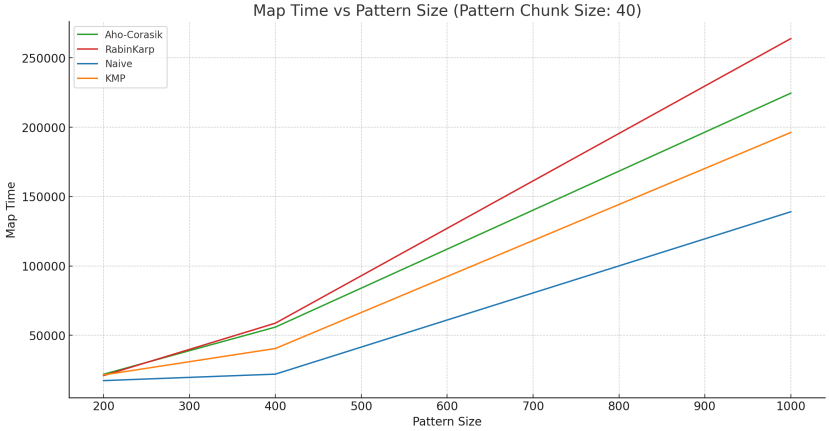

Map Time + Reduce Time vs Pattern Size (Pattern Chunk Size: 20)
1e6
Naive
KMP
Aho-Corasik
RabinKarp
Total Time
Pattern Size

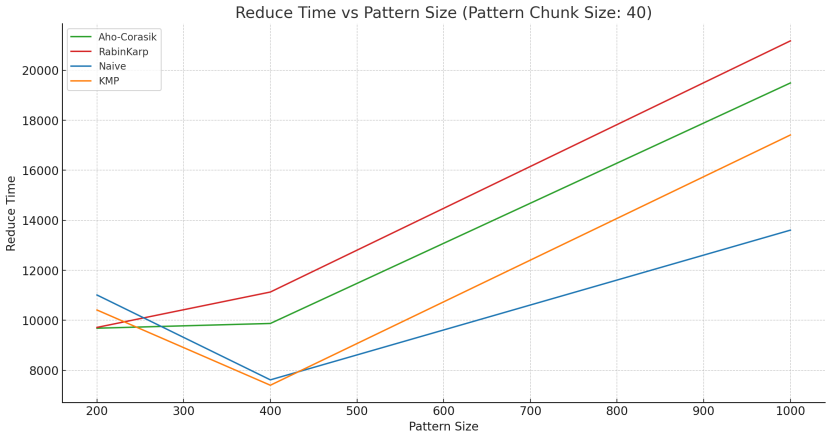
4.2 Text Chunk Size =400, Pattern Size =40

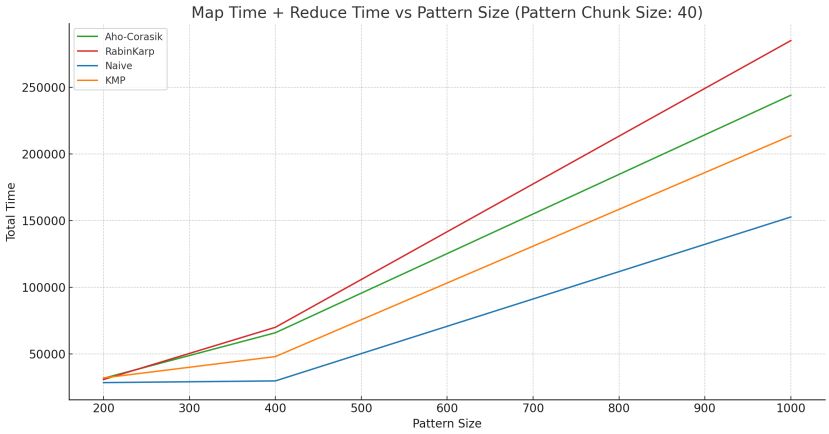





Map Time + Reduce Time vs Text Size (Pattern Chunk Size: 40)
Aho-Corasik
RabinKarp
Naive
KMP
Total Time
250000
200000
150000
100000
50000
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
Text Size
1e7


Map Time vs Pattern Size (Pattern Chunk Size: 40)
Aho-Corasik
RabinKarp
Naive
KMP
Map Time
250000
200000
150000
100000
50000
200
300
400
500
600
700
800
900
1000
Pattern Size


Reduce Time vs Pattern Size (Pattern Chunk Size: 40)
Aho-Corasik
RabinKarp
Naive
KMP
Reduce Time
20000
18000
16000
14000
12000
10000
8000
200
300
400
500
600
700
800
900
1000
Pattern Size



Analysis of time vs Text Size

- As Text Size increases, Map Time generally increases for all algorithms.
- The Naive algorithm tends to have a steeper slope, indicating less scalability with respect to Text Size.
- Reduce Time also increases with Text Size, but the rate of increase is less steep than Map Time.
- The Aho-Corasik algorithm shows a relatively flat profile, suggesting it is less affected by the increase in Text Size compared to other algorithms.
- Total Time combines both Map and Reduce Times and follows the trend of Map Time as it is the dominant factor.
- Naive algorithm appears to be the least efficient, with its Total Time increasing significantly with Text Size.

Analysis of Time vs Pattern Size

- Map Time tends to increase with Pattern Size, but the increase is not as pronounced as with Text Size.
- The Rabin Karp algorithm shows a particularly notable increase in Map Time with Pattern Size, which may be due to its computational complexity that involves hashing.
- The impact of Pattern Size on Reduce Time is minimal for all algorithms, suggesting that the Reduce phase is more influenced by the distribution and aggregation of data rather than the complexity of the pattern itself.
- Total Time increases with Pattern Size, closely following the trend of Map Time since it dominates the Total Time.

Certain scenarios and which algorith is best suited is listed as below:

1. **Search by Error Code**: Distributed Aho-Corasick is the best-suited algorithm for this case as it allows for efficient multi-pattern searches. Logs often contain various error codes, and the ability to search for multiple codes in a single pass is advantageous.

2. **Search by Team ID**: For most corporations/organisations, team IDs have a fixed format and do not vary much in length. Hence, a distributed KMP or Boyer Moore will be best suited for this purpose.

3. **Search by Employee ID**: Distributed Rabin-Karp is a good fit for searching by employee ID. The use of hashing can quickly match employee IDs across multiple log entries without much chance of collisions.

4. **Search by Date**: Distributed Boyer-Moore could be effective for date searches because dates usually have fixed patterns and the Boyer-Moore algorithm's heuristics can skip over many characters that do not match the pattern.

5. **Search by a Entire Log**: Distributed Brute Force might actually be useful if the log is not too large, and the search patterns are not very repetitive. It doesn't require any pre-processing or any memory overhead, which is useful here.

** **NOTE:** In terms of just the amount of time taken, Naive substring-matching seems to be performing the best, i.e. is fastest. But this is because, the memory overhead is less. As chunk size increases (both text and pattern), the performance of the Naive substring-matching method falls. This is because it is an O(mn) algorithm.

** **NOTE:** All results are based on a single run. To get better, and more accurate results, one should run the code multiple times and take an average.

5 Conclusion

For large Text Sizes, algorithms like Aho-Corasik, which show a less steep increase in computation time, may be preferred. Conversely, for tasks with larger Pattern Sizes, algorithms that are less sensitive to an increase in Pattern Size would be more suitable. The Naive algorithm generally shows the least favorable scalability and may only be preferred for smaller datasets or less complex matching requirements.

The choice of the algorithm should be based on the expected frequency of patterns, the size and complexity of the search terms. We need to balance the pre-processing time, memory usage, and actual search efficiency for log analysis.