# TriePDC: Parallel Text Pattern Matching

Krunal Murlidhar Kumbhare
*School of Computing Science and Engineering*
*Vellore Institute of Technology, Chennai*
*India*
krunalmurlidhar.k2019@vitstudent.ac.in

***Abstract* – The process of pattern matching at line speeds is a memory and computation intensive task. It requires dedicated hardware algorithms. To overcome this project I have made an application which would efficiently search and match words in a directory by indexing the files using multiprocessing and searching the words using Trie data structure.**

***Keywords – pattern, trie, text files, directory, multiprocessing***

## I. INTRODUCTION

In computer science, pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact: "either it will or will not be a match." The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace) [1].

Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as backtracking. Tree patterns are used in some programming languages as a general tool to process data based on its structure [1]. Parsing algorithms often rely on pattern matching to transform strings into syntax trees [2][3].

There are already many string matching and searching algorithm available. They are Knuth-Morris-Pratt ( KMP) algorithm, Boyer Moore algorithm, Rabin Karp algorithm and many more. Their comparison in previous researches related to string matching algorithm in documenting content comparing Brute Force algorithm, Knuth-Morris-Pratt algorithm ( KMP), Boyer Moore algorithm (BM), and Rabin Karp algorithm. Using the KMP algorithm has a better speed than the others [4]. Document search algorithm shows that the KMP algorithm has the best performance [5]. Full permutation pattern-matching research with BM, Harspool, Aho-Corasick and KMP algorithms concludes that the KMP algorithm has the fastest permutation pattern-matching performance [6]. However, a comparison of the performance of the KMP, Naïve, and BM algorithms in processing various text sizes concluded that the BM algorithm has the best and most efficient performance for large text sizes [7].

## II. PRELIMINARIES

### *A. Definitions*

*Multiprocessing:* The running of two or more programs or sequences of instructions simultaneously by a computer with more than one central processor.

*Indexing:* An index is a list of data, such as group of files or database entries. It is typically saved in a plain text format that can be quickly scanned by a search algorithm. This significantly speeds up searching and sorting operations on data referenced by the index. Indexes often include information about each item in the list, such as metadata or keywords, that allows the data to be searched via the index instead of reading through each file individually.

*Trie:* a trie, also called digital tree or prefix tree, is a type of search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In order to access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

*JSON:* JavaScript Object Notation is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. The data is stored as key-value pairs.

### *B. Abbreviations and Acronyms*

| Acronyms | Full |
|---|---|
| ICISS | International Conference on Intelligent Sustainable Systems |
| KMP | Knuth-Morris-Pratt |
| BM | Boyer Moore |
| JSON | JavaScript Object Notation |

Table 1: Abbreviations and acronyms used

## III. Related Work

In International Conference on Intelligent Sustainable Systems ( ICISS), 2017 a methodology was proposed by Sanu and Nidhi which had separate modules for database creation, indexing and analysis and, searching. The parsers parsed the content from the files, contents were processed and analyzed. Appropriate indexing was applied for better performance of algorithms. Based on the query the search algorithm would get the word from indexed files and display it to user [8]. Results of the paper showed that it took 14ms to search for a word in a file with pptx format having size of about 1000000 bytes and 1250ms for searching a word in docx file with size of 80000 bytes [8].

It must be observed that the program is executing serially. The performance could be further increased if multiple processes are executed at same time. This would ensure complete processor utilization as well as reduce the time required to get the word. I have proposed a system where I would be solving this problem by using multiprocessing.

## IV. Proposed System

I have designed and developed a system where the word would be efficiently be searched and matched in the text files available in the directory. Here the system is divided to two modules. They are

i. indexing the text files and their contents
ii. searching and matching the prefix

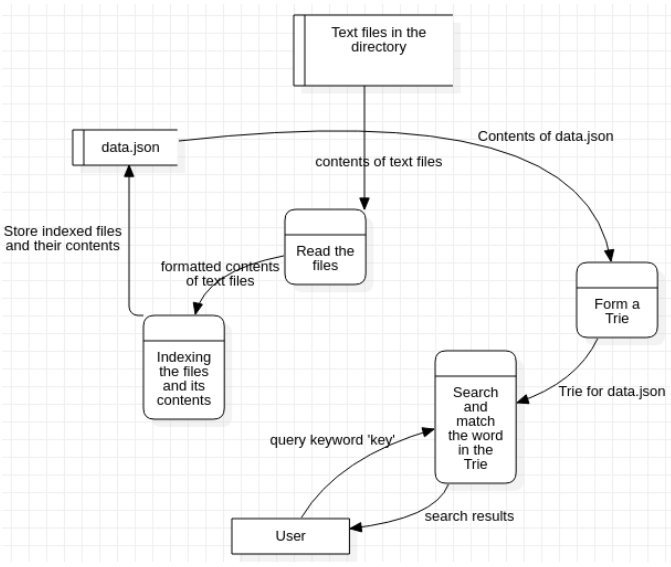


Fig.1: Data flow diagram of TriePDC.

The text files ( ends with .txt) are present in the directory. The files and their contents must be indexed so that the data would be in proper structure for efficient searching. For this the contents of the text files are read and parsed. This data is then indexed using the algorithm defined in section V part A.

Now the indexed data is ready in the form of JSON in data.json.

Now the user's keyword is taken as input. A Trie is created for the contents of data.json. Now the keyword which was entered by user is used to search and match it with the Trie. For searching the algorithm defined in section V part B is used. The results of the search is displayed to the user.

## V. Algorithms

### *A. Algorithm for indexing text files and its contents*

i. From a process, read the text from a text file in the current directory and stored in a variable 'wordInfo'.
ii. Characters like '.', '"', ',', ';', etc. are removed from the texts.
iii. The cleaned text from step ii. is stored back in the variable 'wordInfo'.
iv. The data from the variable 'wordInfo' is parsed and converted to JSON format and appended & stored in variable 'data'.
v. Repeat step i – v for all the text files in the directory using multiprocessing.
vi. Store the JSON contents of the variable 'data' in 'data.json'.

The structure of the JSON would be as follows:

```
{ word: {
      "File": {
            "filename1.txt": {
                  "Line": [ i1, i2, i3, ..., in]
                  },

            "filename2.txt": {
                  "Line": [ j1, j2, j3, ..., jn]
                  },
            }
      }
}
```

### *B. Algorithm for pattern matching and searching*

For 'key' to be searched and matched in the Trie.

i. Read data from data.json
ii. Create a Trie from the data collected from step i. Follow the below steps to create the Trie;
    - Create a node 'temp' and assign the keyword to it. This keyword would act as an index of children.

- If the present node in the Trie already has a reference to the present letter, set the present node to that ‘temp’ node. Otherwise, create a new node, set the value to be equal to the present keyword, and even start the present node with this new node.
- Repeat steps 1 – 3 for all the words in data.json

iii. Initialize node = root, found = True, temp_word = ‘’
iv. for eachKey in list of keys:
- if eachKey is not in node’s children
  set found = False
  break
- set temp_word += eachKey
- assign node = node.children[eachKey]

v. if temp_word != key
- set found = False

vi. if found return []
vii. else append the words and their information in a list L and return L.

## VI. Implementation

### *A. Overview of software*

All the softwares and framework used in implementing TriePDC application are OS independent, i.e., ,they work on any operating system. They can be easily downloaded and installed from any browser or OS centric store.

### *B. Software Requirements*

Following languages/ libraries/ packages must be installed in the computer to run the program.

1. Language:
   - Python (version 3)
2. Packages in python3:
   - json
   - os
   - multiprocessing
   - sys

Visual Studio Code is recommended IDE for development. But users can use other IDEs as par their convenience. Above all any PC with minimal RAM, few MBs of storage would be good for running TriePDC.

### *C. Modules*

As said in section IV TriePDC is divided in two modules. They are -

i. indexing the text files and their contents
ii. searching and matching the prefix

Module i. has files and functions, etc. which would help in indexing the text files. Also separate function is written to structure the JSON. Multiprocessing concept is used to index the files. Algorithm defined in section V part A is used.

Module ii. is dedicated to operations, methods, etc. related to searching the keyword in data.json. Here, a separate class “Trie” is defined with methods performing required operations for/ on the Trie. This makes the class more cohesive. Main methods of the class are constructors, searching function, formatting, autocomplete/ autosuggestion function. The main algorithm used here is defined in section V part B.

### *D. Setting up the project*

1. Execute the following command to download the project.

*git clone https://github.com/krunalmk/ParallelPatternMatchingTrie.git*

2. Extract the zip.
3. Download and install the libraries, language, packages, etc. mentioned in section VI part B.

### *E. Execution of project*

1. Open terminal in the extracted folder
2. Execute the following
   - for indexing the text files

     *python3 reindexthefiles.py*
   - to get parallel prefix match for your input

     *python3 main.py <your word>*

     Eg.:

     *python3 main.py guten*

## VII. Results

Following the steps written in section VI part E we get following output for the keyword “proclaim”

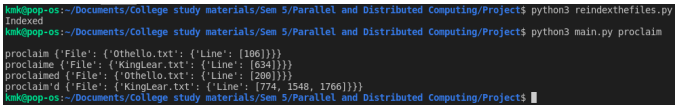


Fig. 2: Indexing the files, searching for information related to “proclaim”

In fig. 2, we can see that we have got four words with their information as results. The words are “proclaim”, “proclaime”, “proclaimed” and, “proclaim’d”. It must be noted that we got these suggestions because we are just focusing on prefix of the words. Also the location of the words are indicated against the words. For e.g., the word

"proclaim" is in the text file "Othello.txt" at line 106+1 = 107.

I have tested TriePDC application. The words searched are "house", "proclaim", "hid", "father" and "jump". The results are shown in Table 2.

| Word | Time ( in milliseconds) |
|---|---|
| house | 0.058349609375 |
| proclaim | 0.01708984375 |
| hid | 0.022705078125 |
| father | 0.030517578125 |
| jump | 0.02197265625 |

Table 2: Test results showing the words and time required to search them using TriePDC.

The directory during testing had four text files namely "shakespeare.txt", "Othello.txt", "KingLear.txt" and "SampleTextFile.txt". They had 124457, 465, 1943 and 130 lines of texts respectively.

Now I have searched the same keywords in the directory using general brute force algorithm which uses re.finditer() method in python. The results are shown in table 3.

| Word | Time ( in milliseconds) |
|---|---|
| house | 12.940185546875 |
| proclaim | 7.140625 |
| hid | 10.55322265625 |
| father | 10.63720703125 |
| jump | 10.3134765625 |

Table 3: Test results showing the words and time required to search them using re.finditer().

The results of the tests show that TriePDC is more than 200 times faster than normal brute force algorithm using re.finditer().

## VIII. Conclusion

In this paper, a new method of searching keyword in text files present in a directory is proposed which uses multiprocessing and Trie data structure. Results show that it is more than 200 times faster than algorithm which uses re.finditer() method. So we can say that TriePDC application which uses multiprocessing for indexing the text files in the directory and Trie for searching is efficient and better in performance compared to the basic method of searching.

## IX. Future Work

TriePDC can be extended further by using distributing the indexing task using distributed computing. And GPUs could be used for searching and matching the keywords. In this way we would get benefits of both distributed computing and as well efficiently use multiple cores of GPU. This would further reduce the total cost in terms of time.

## Acknowledgment

The author would like to thank Prof. Harini S. and Vellore Institute of Technology, Chennai for providing the required guidance and support for writing this paper.

## References

[1] https://en.wikipedia.org/wiki/Pattern_matching
[2] Warth, Alessandro, and Ian Piumarta. "OMeta: an object-oriented language for pattern matching." Proceedings of the 2007 symposium on Dynamic languages. ACM, 2007.
[3] Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." SIAM journal on computing 6.2 (1977): 323-350.
[4] R. Janani and S. Vijayarani, "An Efficient Text Pattern Matching Algorithm for Retrieving Information from Desktop," Indian J. Sci. Technol., vol. 9, no. 43, Nov. 2016, doi: 10.17485/ijst/2016/v9i43/95454.
[5] M. B. Sri, R. Bhavsar, and P. Narooka, "String Matching Algorithms," Int. J. Eng. Comput. Sci., vol. 7, no. 3, 2018.
[6] Diptarama, R. Yoshinaka, and A. Shinohara, "Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings," in Proceedings of the Prague Stringology Conference 2016, 2016, pp. 7–21.
[7] P. Chettri and C. Kar, "Comparative Study between Various Pattern Matching Algorithms," in International Conference on Computing & Communication, 2016.
[8] S. Lakhara and N. Mishra, "Design and implementation of desktop full-text searching system," 2017 International Conference on Intelligent Sustainable Systems (ICISS), 2017, pp. 480-485, doi: 10.1109/ISS1.2017.8389458.
[9] M. Yadav, A. Venkatachaliah and P. D. Franzon, "Hardware Architecture of a Parallel Pattern Matching Engine," 2007 IEEE International Symposium on Circuits and Systems, 2007, pp. 1369-1372, doi: 10.1109/ISCAS.2007.378482.

## Appendix

[1] The source code for this project is available on this link. https://github.com/krunalmk/TriePDC .Feel free to contribute to the project.
[2] For more information on Trie data structure visit https://en.wikipedia.org/wiki/Trie