An Approach to Parallelization of Naive String Matching Algorithm using Java Functional Parallelism Frameworks

[1] Vijay Kumar, [2] Alka Singh,
[1]M.Tech Student, [2]Assistant Professor,
[1] Department of Computer Science,
[1] Kamla Nehru Institute of Technology, Sultanpur, Uttar Pradesh, India.

Abstract : String matching is one of the significant classes of problems in computer science. Searching smaller text in larger data in a reasonable time is a difficult task for an algorithm. The Naive string matching algorithm is the basic linear single pattern algorithm. After the survey of the Naive string matching algorithm, we found that the performance of the algorithm is better than some advanced algorithm (Boyer-Moore) when the pattern exists at the starting of the larger data text. In this work, we are applying the Naive string matching algorithm using the proposed concept of parallelization to enhance the performance. The concept of parallelization is based on SIMD architecture where data divided and distributed for parallel processing. Using different JAVA functional parallelism frameworks the results are generated and compared. The comparison suggests that the performance of one framework is proved to be better than another framework.

Index Terms - **String Matching, Naïve Algorithm, SIMD, Parallel Naïve.**

I. Introduction

In this era of enormous data searching a piece of information in extremely huge data in a sequential manner is quite old-fashioned. Huge data requires a parallel approach to deal with. Researchers had been doing research to improve the efficiency of string matching algorithms [1, 2, 3]. Information of greater importance in an enormous amount of data is needed to be retrieved, so the means of retrieval of information should be efficient enough to extract a piece of information within a given time. The availability of information at a particular point in time has a greater impact. The role of information retrieval is significant for real-time applications. String matching algorithms have a wide range of usage in computer science. With the concept of parallelism better performance of the Naive algorithm is achieved.

II. Related Work

Ubaid S. Alzoabia et al.[4] proposed parallel KMP algorithm. In their research they proposed a strategy for data division to distribute data over the individual processors. They provided with the detailed description of the KMP algorithm and about the performance of the algorithm using parallel approach. They had divided the data with the number of available processors/core and the remainder is distributed among all processors.

Rasool et al.[5] proposed implementation of KMP string matching in parallel using different SIMD architecture-multicore and GPGPUS. They designed the algorithm that work on SIMD architecture and concluded that their work is proved better then multithreaded implementation.

S.V. Raju et al. [6] in their research used grid computing approach to implement string matching in parallel using grif MPI. Their approach for is based on Single Program Multiple Data method, data is divided into parts and then executed in parallel simultaneously. Grid computing is usually used in finding solution to various complex problems.

III. Parallel Processing Overview

The goal of parallelism is to efficiently utilize available resources concurrently to solve a problem. Resources such as processing units, graphical processing units, memory units, and other required resources local or distributed. To speed up the processing faster processor, faster memory, faster communication network, etc. An economical solution to a problem is desired by using concept of parallel processing. While using concept of parallelism it should be kept in minds that which:

- The type of application under consideration.
- Parallel computer model to be used.
- Parallel algorithm-design techniques to be used.
- Which parallel algorithm model is best?

IV. Naive String Matching Algorithm

Given a text string str[0..n] and a pattern ptr[0..m], write a function search(char ptr[], char str[]) that prints all indexes of occurrence of ptr[] in str[]. It is assumed that 'n' being greater than 'm'. Time-complexity for the algorithm comes out to be O (n-m+1) m) i.e. is O (nm).

4.1 NAÏVE ALGORITHM WITH PROPOSED PARALLELIZATION

The algorithm has been applied, to work on parallel architecture supporting strings of larger sizes. The motive of using the concept of parallelization is to enhance the performance of the algorithm. Parallelization requires large size string to be divided into parts irrespective of pattern size. The same pattern is executed on different parts of the string in parallel, thereby reducing the time complexity of the algorithm. Talking about memory and processors, much reliable multiple executions can be achieved in parallel. It is possible to apply the same notion of Naive Search to match the pattern in the strings divided into various parts and performed in parallel. Here we illustrate the data division technique in figure 4.1.2 along with further changes. Suppose there are four processors available. So we divide the text into four parts and shared memory keeps the patterns and four different parts are processed by four different processors. In this parallelization, the architecture used is SIMD architecture. Here the Naive algorithm is executed on individual data for parallel.

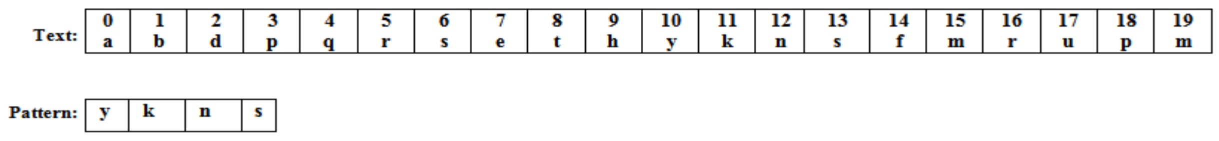
Text:
0 a 1 b 2 d 3 p 4 q 5 r 6 s 7 e 8 t 9 h 10 y 11 k 12 n 13 s 14 f 15 m 16 r 17 u 18 p 19 m
Pattern:
y k n s

Figure 4.1: Data before division

The Figure 4.1 shows sample text and pattern.
We have divided the data depending on:

- Number of Processors core - (k)
- Length of the Pattern (pat) - (L_P)
- Length of Data String (S) - (L_D)

Assume that ($L_P < L_D$)
For k = 4 (Number of processors/Cores)
Let us assume Processors (cores) P1, P2, P3, and P4
For k = 4, data should be divided into 4 Sub parts S_{P1}, S_{P2}, S_{P3} and S_{P4}

Initially,
[Length of sub_part (L_{SP}) = L_D / k]
[Length (S_{P1}) = Length (S_{P2}) = Length (S_{P3}) = Length (S_{P4}) = L_{SP}]

Redefining sub parts length (S_{Pi}) depending upon the pattern occurring at division points:

```
int dp1 = $L_P$ -1;
int dp2 = $L_P$ -1;
int dp3 = $L_P$ -1;
for (int y=1; y<=3; y++) {
for (int z=1; z<=( $L_P$ -1); z++) {
If (t.charAt ((y* $L_{SP}$)-z)! = p.charAt (0)) {
If (y==1)
dp1 = dp1-1;
 if (y==2)
dp2 = dp2-1;
 if(y==3)
dp3 = dp3-1;
}
Else {
 break;
 }
 }
 }
```

Finally,
Length (S_{P1}) = L_{SP}+dp1, length (S_{P2}) = L_{SP}+dp2, length (S_{P3}) = L_{SP}+dp3 and length (S_{P4}) = L_{SP}

Index for sub_part (S_{P1}) for processor P1: [0, ((L_{SP} + (dp1))]
Index for sub_part (S_{P2}) for processor P2: [L_{SP} , ((2*L_{SP}) + (dp2))]
Index for sub_part (S_{P3}) for processor P3: [(2*L_{SP}) , ((3*Lsp) + (dp3))]
Index for sub_part (S_{P4}) for processor P4: [(3*Lsp) , L_D] //Fourth part (kth part) length will remain unchanged.

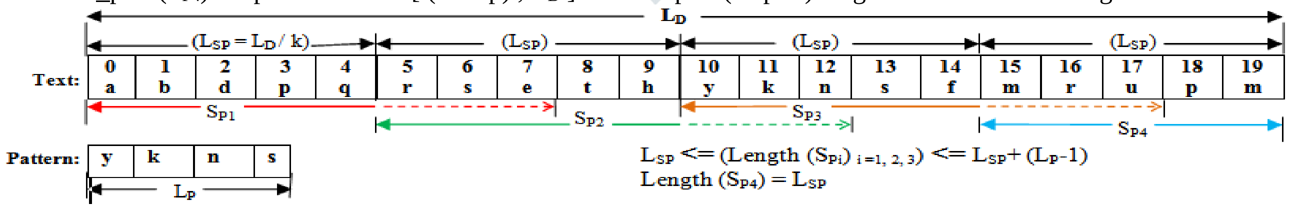
L_D
($L_{SP} = L_D / k$)
(L_{SP})
(L_{SP})
(L_{SP})
Text:
0 a 1 b 2 d 3 p 4 q 5 r 6 s 7 e 8 t 9 h 10 y 11 k 12 n 13 s 14 f 15 m 16 r 17 u 18 p 19 m
S_{P1}
S_{P2}
S_{P3}
S_{P4}
Pattern:
y k n s
L_P
L_{SP} <= (Length (S_{Pi}) $_{i=1, 2, 3}$) <= L_{SP}+ (L_P-1)
Length (S_{P4}) = L_{SP}

Figure 4.1.2: Data after division (dotted portion of the arrow shows overlapping portion ((dpi <=L_p-1) $_{i = 1, 2, 3,}$) of 'S_{Pi}' which is varying in length)

The purpose of redefining the length of first 'k-1' parts after division is to avoid unnecessary overlapping at the 'k-1' division points. The reason behind the emphasis at the division point is that when the pattern size is comparatively large say half the size of the data string the, in that case, the overlapping part will be larger, so by doing few comparisons we can redefine the size of the subparts and avoid overlapping.

Proposed Parallelize Simd Based Naïve Algorithm:
//Naïve Algorithm Function
NaiveSearch (s, pat)
s - Data string in which pattern is searched
pat - String to be search.

ParallelNaive:

```
Step 1: Start
Step 2: Input S(data string) or Data_File.txt, pat(pattern), k = no. of processor;
Step 3: $L_D$ = S.length ();
Step 4: $L_P$ = pat.length ();
Step 5: length of sub part ($L_{SP}$) = $L_D$/k; // C is no. of available Processor/Cores
Step 6: For all processors/cores $p_i$ where $1 \leq i \leq k$ // k is equal to no. of processors
{String Spi = [(S.substring (0, ($L_{SP}$ + (dp1))); // $S_{p1}$  Sub parts for threads 1),
(S.substring ($L_{SP}$, (2* $L_{SP}$ + (dp2))); // $Sp_2$ Sub parts for threads 2),
(S.substring ((2* $L_{SP}$), (3*$L_{SP}$ + (dp3))); //$Sp_3$ Sub parts for threads 3),
 (S.substring ((3* $L_{SP}$), (k*$L_{SP}$ = $L_D$)); //$Sp_4$ Sub parts for threads 4),
.
(S.substring (((k-1)* $L_{SP}$), (k*$L_{SP}$)); // $S_{pk}$ Sub parts for threads 'kth')]}
Step 7: Result $R_i$ from all threads where $1 \leq i \leq k$
{int $R_i$ = [(NaiveSearch ($S_{pi}$, pat) //$R_1$ result produced b from thread 1),
(NaiveSearch ($S_{pi}$, pat) //$R_2$ result produced from thread 2),
(NaiveSearch ($S_{pi}$, pat) //$R_3$ result produced from thread 3),
.
(NaiveSearch ($S_{pk}$, pat) //$R_k$th result produced from thread 'kth')]}
Step 8: Int Global Result = $R_1 + R_2 + R_3 + \ldots + R_k$; //Global Result
Step 9: Print Final Result
Step 10: Stop
```

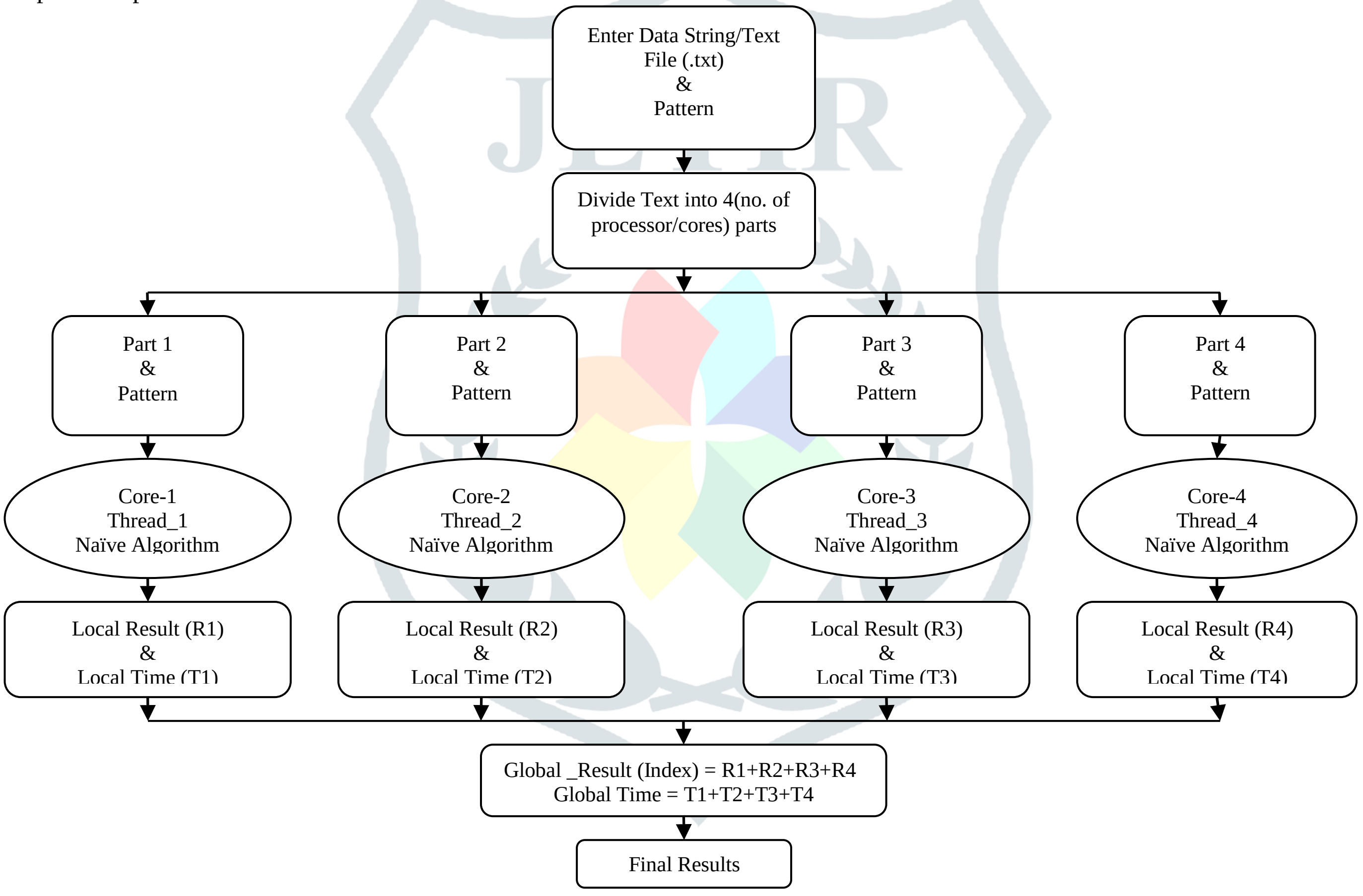
Enter Data String/Text File (.txt) & Pattern
Divide Text into 4(no. of processor/cores) parts
Part 1 & Pattern
Part 2 & Pattern
Part 3 & Pattern
Part 4 & Pattern
Core-1 Thread_1 Naïve Algorithm
Core-2 Thread_2 Naïve Algorithm
Core-3 Thread_3 Naïve Algorithm
Core-4 Thread_4 Naïve Algorithm
Local Result (R1) & Local Time (T1)
Local Result (R2) & Local Time (T2)
Local Result (R3) & Local Time (T3)
Local Result (R4) & Local Time (T4)
Global _Result (Index) = R1+R2+R3+R4 Global Time = T1+T2+T3+T4
Final Results

Figure 4.1.3: Proposed Model

V. ALGORITHM ANALYSIS

This method very much improves the performance of the algorithms. The time complexity of the string matching algorithm of worst case scenario is O (nm), 'n' is the size of text in which pattern of size 'm' is to be searched. Algorithm utilizes the available processors by make the number of string divisions equal to available number of processors k and approximately with same number of characters. Time complexity of our parallel algorithm is $O(\frac{nm}{k} + m - 1)$.The algorithm process all part in parallel in one stage without dependency between them.

VI. EXPERIMENTAL RESULT AND ANALYSIS

Supplying divided data string to individual thread and applying Naïve String Matching Algorithm on each thread in parallel and executing algorithm on individual data string on each thread. After execution on each thread, result of each thread is combined and global result is produced.

Naïve String matching algorithm is implemented using two Java Functional Parallelism frameworks which are:

- Executor Service Framework.
- ForkJoinPool Framework.

Using the above two frameworks we can implement string matching algorithm using parallel model and analyze the performance of the two frameworks and find out which framework is efficient in execution. We will take character strings of increasing length and record the time of execution and analyzing the performance.

6.1 Experimental Environment:

System Configuration Implemented on:

- Processor: i5(Intel® Core™ i5-3210M CPU @ 2.5 GHz processor(Dual Core)
- RAM: 6GB
- OS: Windows 7(64 Bit) operating system
- Language: Java SE runs on Eclipse 2018-12(4.10.0)
- Language (parallel Implementation): Java ExecutorService and ForkJoin Framework.

To implement the algorithm we used Eclipse IDE 2018-12(4.10.0) along with Java 8.0/ JDK 1.8.

6.2 Experimental Data for String Matching

Data String: Text of size of 10000000 -120000000 characters.
Text File: Text files containing data string (File size: 9.53 MB to 114 MB)
Pattern File: Pattern of size 10 character.
Here we are taking 4 threads, executing one thread on each core in multi-core CPU. The Implementation is done on 10 texts sample with different length n from 10000000 to 120000000 characters and each next text is grater by 10000000 characters. In Executor Service string is distributed to the individual threads depending upon the distribution strategy and the executing is carried out whereas in ForkJoin after the distribution of the string to the individual threads further internal breakdown is done until simpler instances are achieved under this framework.

6.3 EXPERIMENT

Naïve String matching algorithm is implemented in three different ways:

- Serial
- Multicore CPU using Executor Service Framework
- Multicore CPU using ForkJoin Framework

The experimental outcomes are shown in tables 6.3; the comparison is shown in the graphs figure 6.3 below. In Executor Service Framework implementation a speedup of 1.02 is gained, in ForkJoin Framework implementation a speedup of 1.16 is achieved in comparison to the serial implementation. Speedup also depends upon machine producing result by parallel processing.

Table 6.3 Table of execution time for comparative analysis of Sequential Naïve String Matching algorithm, Parallel Naïve String matching using Java Executor Service Framework and Parallel Naïve String matching using Java ForkJoin Framework.

S.No.	**Length of Data String (in Number of Characters)**	**Execution time of Sequential Naïve String Matching Algorithm (in Milliseconds)**	**Execution time of Parallelized (using Java Executor Service framework) Naïve String Matching Algorithm (in Milliseconds)**	**Execution time of Parallelized (using Java ForkJoin framework) Naïve String Matching Algorithm (in Milliseconds)**
1	1 Crore	107	166	103
2	2 Crore	182	214	182
3	3 Crore	258	301	257
4	4 Crore	344	395	326
5	5 Crore	420	468	392
6	6 Crore	548	484	421
7	7 Crore	623	578	530
8	8 Crore	700	641	592
9	9 Crore	796	747	676
10	10 Crore	854	811	741
11	12 Crore	1018	927	810

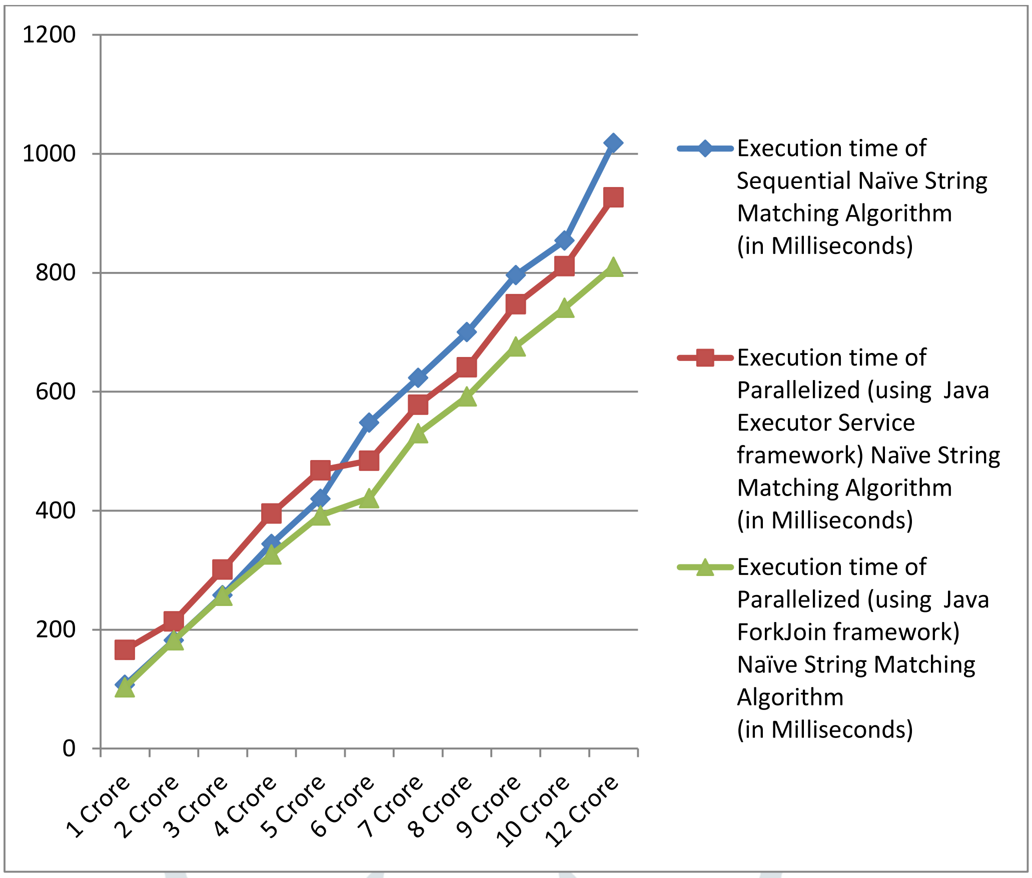

Figure 6.3: Execution time chart for comparative analysis of Sequential Naïve String matching algorithm, Parallel Naïve String matching using Java Executor Service Framework and Parallel Naïve String matching using Java ForkJoin Framework.

VII. CONCLUSIONS

Through this work, we achieved better performance in terms of execution time and also result of this work proves that the ForkJoin Framework qualifies to be the best framework for parallel implementation of this algorithm. The performance of the ForkJoin framework is better than the Executor Service framework. From the result, it can be concluded that the ForkJoin framework may have better or we can say lesser context switching overhead between threads and may also have better load distributed between sub-tasks internally.

REFERENCES

[1] Chettri, Pranit, and Chinmoy Kar. "Comparative Study between Various Pattern Matching Algorithms." *International Journal of Computer Applications* 975: 8887.

[2] Published IEEE papers related to confined topics of KMP and other string matching algorithms, references as , Knuth DE,Morris JH,Pratt V R.Fast pattern in strings[J]. SIAM Journal on Computing,1977,6(2):323-350.

[3] Tang Va-ling. KMP algorithm in the calculation of next array. Computer Technology and Development [J] .2009,19 (6) :98-101.

[4] Alzoabi, Ubaid S., et al. "Parallelization of KMP string matching algorithm." 2013 World Congress on Computer and Information Technology (WCCIT). IEEE, 2013.

[5] Rasool, Akhtar, and Nilay Khare. "Parallelization of KMP string matching algorithm on different SIMD architectures: Multi-core and GPGPU's." International Journal of Computer Applications49.11 (2012).

[6] Rajashekharaiah, K. M. M., Ch MadhuBabu, and S. Viswanadha Raju. "Parallel string matching algorithm using grid." International Journal of Distributed and Parallel Systems3.3 (2012):21.