Performance Comparison of Distributed Pattern Matching Algorithms on Hadoop MapReduce Framework

C. P. Sona(✉) and Jaison Paul Mulerikkal

Śúnya Labs, Rajagiri School of Engineering and Technology, Kochi, India
sonacp@rajagiritech.edu.in

Abstract. Creating meaning out of the growing Big Data is an insurmountable challenge data scientists face and pattern matching algorithms are great means to create such meaning from heaps of data. However, the available pattern matching algorithms are mostly tested with linear programming models whose adaptability and efficiency are not tested in distributed programming models such as Hadoop MapReduce, which supports Big Data. This paper explains an experience of parallelizing three of such pattern matching algorithms, namely - Knuth Morris Pratt Algorithm (KMP), Boyer Moore Algorithm (BM) and a lesser known Franek Jennings Smyth (FJS) Algorithm and porting them to Hadoop MapReduce framework. All the three algorithms are converted to MapReduce programs using key value pairs and experimented on single node as well as cluster Hadoop environment. The result analysis with the Project Gutenberg data-set has shown all the three parallel algorithms scale well on Hadoop environment as the data size increases. The experimental results prove that KMP algorithm gives higher performance for shorter patterns over BM, and BM algorithm gives higher performance than KMP for longer patterns. However, FJS algorithm, which is a hybrid of KMP and Boyer horspool algorithm which is advanced version of BM, outperforms both KMP and BM for shorter and longer patterns, and emerges as the most suitable algorithm for pattern matching in a Hadoop environment.

Keywords: Pattern matching · Hadoop · MapReduce · Big Data
Knuth Morris Pratt Algorithm · Boyer Moore Algorithm
Franek Jennings Smyth Algorithm

1 Introduction

Big Data is defined as a large collection of data-sets that grow exponentially with time. It is generally understood that the four characteristics of Big Data are volume, velocity, variety and veracity [13]. This varied large volume of data can be from applications like the Large Hardon Collider of CERN or from the human genome project. It could also be the data generated by jet engines, which

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
J. Hu et al. (Eds.): MONAMI 2017, LNICST 235, pp. 45–55, 2018.
https://doi.org/10.1007/978-3-319-90775-8_4

can generate up to 10 terabytes of data in 30 min of flight time [5]. Another example is the New York Stock Exchange which generates about 1 terabyte of new trade data per day [14].

The Big Data can be structured, semi-structured or unstructured. The data which is structured and semi-structured can be addressed using the traditional data management tools but unstructured data still remains unsolved using traditional methods. The efficient processing tool that can deal with Big Data is Hadoop MapReduce framework developed by Doug Cutting [17]. It is systematically designed to process Big Data in a scalable way through distributed processing. The Hadoop Distributed File System (HDFS) creates the distributive environment that is required for the parallel processing of data. The Mapper and Reducer functions help to split and parallelize the work for faster processing of data.

Pattern matching algorithms are good candidates to decipher insights from Big Data. They are also known as string matching algorithms. These are essential classes of string algorithms which help discover one or all existences of the string within an enormous group of text [3]. It is a solution for many real world problems. Many applications such as twitter analysis, information retrieval, sentimental analysis and DNA analysis use pattern matching algorithms at different stages of processing. For example, protein link prediction using the DNA genes requires a stage where a good pattern matching algorithm is required to match the DNA pattern and take a count of matched DNA for further processing. Normal prediction requires high execution time due to processing of more than fifty thousands of DNA pattern. This execution time can be improved using effective pattern matching algorithm which works well on distributed environment [12]. The efficiency varies with applications as well as different parameters likes pattern length, data-set etc.

Certain pattern matching algorithms such as Knuth Morris Pratt Algorithm (KMP), Brute Force Algorithm and Boyer Moore Algorithm (BM) have proven to be some of the optimal solutions for such applications [7,18]. There were several attempts to parallelize KMP, BM and Brute Force algorithms for small sets of data. However those efforts found it difficult to use these parallel algorithms for large sets of data by distributing Big Data among many nodes. Hadoop becomes a natural candidate to overcome this difficulty.

This study is focused on creating parallelized versions of these three algorithms - viz., KMP, BM and FJS - to work with Hadoop MapReduce framework, so as to scale well with Big Data to produce increased performance. The experiments have been carried out on single node as well as Hadoop MapReduce setups with different sizes of data-sets and lengths of patterns using Project Gutenberg textual data-set. FJS algorithm proves to be the most efficient algorithm on Hadoop MapReduce framework for shorter as well as longer patterns. Other inferences are explained in detail in the result analysis section.

In this paper, Sect. 2 will discuss background studies and related works. Section 3 gives a brief description about pattern matching and algorithms used.

Section 4 describes the experimental setup and followed by result analysis at Sect. 4.6. Section 5 gives the conclusion and future scope.

2 Related Works

In the past there were many attempts [9,10,16] to parallelize pattern matching algorithms in distributed environments. Many attempts succeeded in dealing with small sets of data but either never tried with large sets of data or were confronted with road blocks when chose to deal with it.

Diwate and Alaspurkar [11] conducted linear experiments on different pattern matching algorithms which gave the conclusion that the KMP algorithm is more effective than the BM and Brute Force algorithm. They found out that time performance of exact string pattern matching can be greatly improved if KMP algorithm is used.

Alzoabi et al. [16] proposed a parallel KMP algorithm using MPI programming which give better improvement in execution time. However they could not find an efficient way to split the data when it became large. Cao and Wu [10] have also parallelized the KMP algorithm using MPI programming but it started to show communication errors when the number of processes exceeded 50 or so.

Kofahi and Abusalama [9] have proposed a framework for distributed pattern matching based on multi-threading using java programming. The framework addresses the problems in splitting the texual data sets, but it is not efficient when large number of smaller text files are processed.

Sardjono and Al Kindhi [15] proposed that performance measurement of pattern matching on large sets of Hepatitis C Virus Sequence DNA data showed that Boyer Moore is efficient when comes to minimum shift technique whereas Brute Force algorithm has higher accuracy for pattern matching or finding a match. The study also proved that either KMP or BM algorithm can be chosen as appropriate algorithm according to the pattern length. The disadvantage was that they did not conduct the study in a distributed environment.

Ramya and Sivasankar [2] explains the efforts to port KMP algorithm to Hadoop MapReduce framework and proved that it is possible. However, they have done experiments for only single occurrence of pattern.

Franek et al. have introduced a new linear pattern matching algorithm, which is a hybrid version of KMP and BM. It uses the good features of both KMP and BM for execution and it is explained in [19]. This paper proves that their algorithm, which is generally know as FJS algorithm, has better execution time than most other pattern matching algorithms available, including KMP, BM and Brute Force.

It is in this context, that experiments focus to find an optimal algorithm that can work well with Hadoop MapReduce framework which will be helpful in dealing with Big Data applications. Drawing inspiration from the above literature survey, KMP, BM and FJS algorithms were chosen to test on Hadoop MapReduce framework and to compare the efficiency of pattern matching algorithms on a distributed environment.

Algorithm 1. Knuth Morris Pratt algorithm

```
class Mapper
method Initialize
H = new AssociativeArray
method Map(docid id,doc d)
for all term t in doc d
If t statifies KMP.Prefix(i) do
KMP.SearchPattern(t,p,Prefix(i))
H[t] = H[t] + 1
Emit(term t,count 1)
for all term t in H do
Emit(term t, count H[t])
class KMP
method Initialize
p ← Pattern
method ComputePrefix(p,i,j)
return Prefix(i)
method SearchPattern(t,p,Prefix(i))
return pattern(t,id)
class Reducer
method Reduce(term t, counts [c1, c2,...])
sum = 0
for all count c in [c1, c2,...] do
sum = sum + c
Emit(term t, count sum)
```

3 KMP, BM and FJS Algorithms and Their MapReduce Versions

Pattern or string matching can include single pattern algorithms, algorithms using a finite set of patterns and algorithms using infinite number of patterns. Single pattern algorithms used here includes KMP algorithm, BM algorithm and some improved algorithms which includes FJS algorithm. BM algorithm is considered as the bench mark algorithm for pattern matching [6]. KMP is considered as first linear time string-matching algorithm So It was important to check its efficiency in distributed environment. FJS algorithm which is the hybrid combination of KMP and BM has also chosen for experiments since it was proved to be efficient in linear implementations as reported in the previous section.

3.1 Knuth Morris Pratt Algorithm

The KMP algorithm was developed by Knuth and Pratt, and independently by Morris. The KMP algorithm uses prefix table for string matching to avoid back-tracking on string for redundant checks. It has been reported that it works well on shorter patterns [7]. The KMP algorithm is explained in [8] and its MapReduce version is given at Algorithm 1. The KMP matcher performs the shifts

Algorithm 2. Boyer Moore algorithm

```
class Mapper
method Initialize
H = new AssociativeArray
method Map(docid id,doc d)
for all term t in doc d
If t statifies BM.Search() do
H[t] = H[t] + 1
else BM.Badrule()
Emit(term t,count 1)
for all term t in H do
Emit(term t, count H[t])
class BM
method Initialize
p ← Pattern
method Search(p,i,j)
while ( P.charAt(j) == T.charAt(i0+j) )
j—
return P with corresponding term t
method BM.Badcrule
return i0 = i0 + j - lastOcc[T.charAt(i0+j)]
class Reducer
method Reduce(term t, counts [c1, c2,...])
for all count c in [c1, c2,...] do
sum = sum + c
Emit(term t, count sum)
```

while performing string matching. While porting this algorithm to MapReduce programming paradigm, the document ID and document contents are selected as key, value pairs. The result of which emits the term of document contents containing the required pattern with number of occurrences.

3.2 Boyer Moore Algorithm

In BM algorithm [4], the string check is done from right end of the string. It uses bad character shift table and good suffix table. This is generally used for DNA analysis. As reported in [4] BM algorithm works well for long pattern lengths.

The MapReduce adaptation of this algorithm is given at 2. Here in the mapper phase the document ID and contents of document is selected as key value pairs. The mapper phase emits the terms in documents which satisfies the BM.Search() where it tries to match from the end of the string and if match position is 0 then jump ahead characters. The search continues based on for each character perform right-to-left scan. The bad character rule for each character for rightmost occurrence of character in pattern p is assigned to be zero if character does not occur in p. The terms satisfying the predicate are recorded

Algorithm 3. FJS algorithm

```
1: class Mapper
2: method Initialize
3: H = new AssociativeArray
4: method Map(docid id,doc d)
5: for all term t in doc d
6: If t statifies FJS.Search() do
7: class FJS
8: method Initialize
9: p ← p(1..x)
10: t ← t(1..y)
11: method Search(p,t,doc d)
12: if x < i then return
13:     i' ← x
14: end if
15: if i' < n then
16:     x' ← x-1
17: end if
18: sundayshift.
19: x[i'] ← p[m].
20: i ← x-1.
21: KMP-Match(x,t)
22: return Pattern
23: class Reducer
24: method Reduce(term t, counts [c1, c2,...])
25: sum = 0
26: for all count c in [c1, c2,...] do
27: sum = sum + c
28: Emit(term t, count sum)
```

in a associative array. At reducer phase the term containing the pattern and its number of occurrences are emitted as list of key-value pairs.

3.3 Franek Jennings Smyth Algorithm

The FJS algorithm is a hybrid algorithm of KMP and BM. It uses KMP algorithm if it finds a partial match, else it uses the simplified version of BM method with help of sunday-shift. The algorithm only use bad-character shift for computing. Pre-processing phase prepare the bad character shift value and that table used during the searching phase of algorithm. The algorithm is explained at [19] and its MapReduce version is presented at Algorithm 3. The mapper function reads the contents from the documents emits the key-value pair containing the terms and count which satisfies the FJS predicate. The reducer function outputs the terms its total number of occurrences.

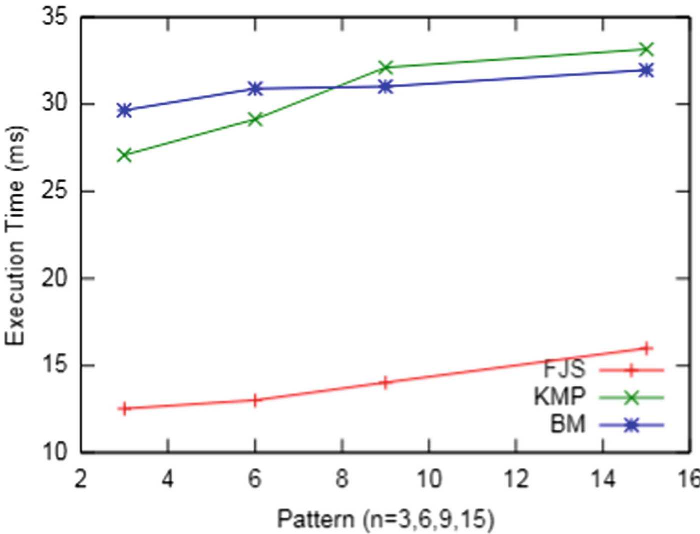

Fig. 1. Execution time of pattern matching of different lengths of patterns on 1 GB

4 Performance Analysis

4.1 Experimental Setup

The experiments are conducted on three different configurations as explained at Sects. 4.2, 4.3 and 4.4. The configurations include experiment on single node as well as cluster Hadoop implementations. The initial studies were conducted on a single node commodity machine installed with Hadoop for single node. It was then extended to a single node server machine with Hadoop for single node installation. Final experiments were conducted on a commodity cluster installed with Hadoop cluster version. Hadoop version 2.2.0 and Eclipse IDE are used. In all the cases (Sects. 4.2, 4.3 and 4.4), the data-set used was the open data available at Project Gutenberg [1]. An average of execution time is taken from 3 consecutive runs of the program for all cases.

4.2 Single Node on Commodity Machine

The Single node Hadoop configuration was first tested on Intel Core i3-2120 machine with 8 GB RAM. The machines ran Ubuntu 16.04 OS and used Eclipse IDE as programming framework. Pattern matching experiment on dataset size of 1 GB were carried out. Each algorithm was evaluated with an increase in pattern length from 3 term to 15 term which was performed on 1 GB dataset.

4.3 Single Node on Server Machine

The single node Hadoop implementation was done on an Intel Xeon E5-2650 V3 server machine with a RAM of 32 GB. The OS used was Debian 8 (Jessie).

Pattern matching experiments of varying sizes of data were carried out on this configuration.

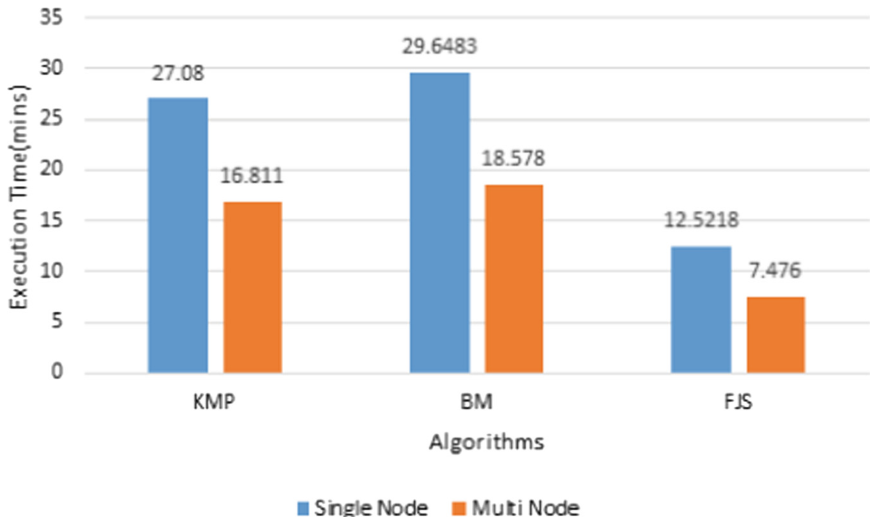

Fig. 2. Execution time analysis of algorithms on single node setup configuration of Sect. 4.2 and multi-node Sect. 4.4

4.4 Multi-node on Commodity Machine

This configuration consists of three Intel Core i3-2120 machines (similar to the machine at Sect. 4.3) each with 8 GB RAM. The multi-node Hadoop system was introduced by configuring one of the systems as master node. The same system also runs a slave instance. The other two machines run one each slave instances. So, the multi-node setup consists of one master and three slave instances. All nodes run Ubuntu 16.04 OS and Eclipse IDE is used for programming development. Pattern matching experiments on data-set size of 1 GB were carried out on this multi-node configuration.

4.5 Details of Experiments

The data-set from Project Gutenberg which contains large collection of small text data was pre-processed and was loaded to HDFS. In the first phase of experiment, the pattern matching was performed on Hadoop single node. Each algorithm was tested on data-set of size 1 GB to 3 GB.

All occurrence of patterns with matching index line was stored in an output file. The execution time of three pattern matching algorithms (i.e. KMP, BM and FJS) were noted and average is reported in Fig. 3.

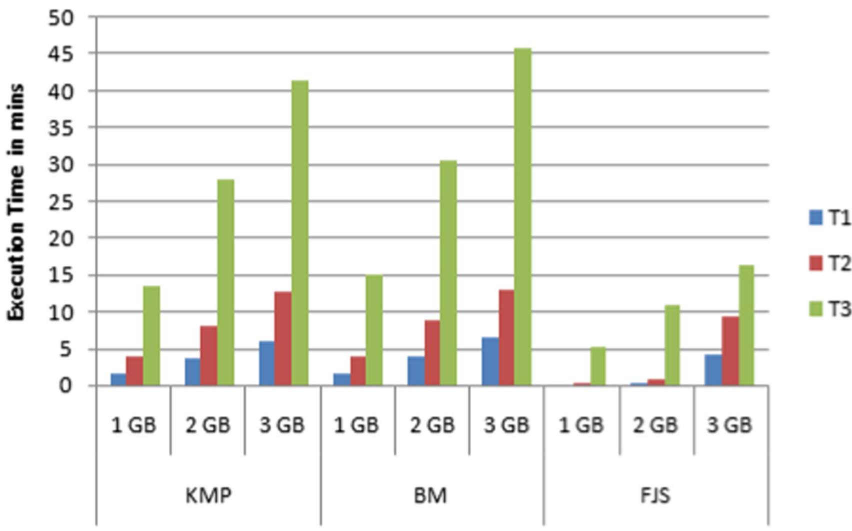

Fig. 3. Execution time analysis of algorithms on single node configuration Sect. 4.3

In the second stage of experimental study, each algorithm is processed on a data-set of 1 GB with different pattern length ranging from 3 term pattern to 15 term. The results are shown in Fig. 1.

In the final sets of experiments, the Hadoop multi-node cluster with 1 master node and 3 slave nodes are used. Pattern matching is performed on 1 GB of data. The results of these experiments are shown in Fig. 2.

4.6 Result Analysis and Performance Comparison

Figure 3 shows the performance of algorithms in terms of mapper time (T1), reducer time (T2) and execution time (T3). Our experiments prove that all the three algorithms under consideration, scale well with regards to increase in data on Hadoop MapReduce framework with almost linear progression for T1, T2 and T3 as shown in Fig. 3 on a single node server configuration as explained in Sect. 4.3. This suggests that these algorithms can be successfully parallelized using Hadoop MapReduce framework to analyze increasing data, or in other words, Big Data.

The results of pattern matching experiments performed on a uniform 1 GB data with different pattern sizes ranging from 3 terms to 15 terms using single node configuration as experimented in Sect. 4.2 are reported in Fig. 1. Figure 1 has corroborated previously reported trends of linear KMP and BM algorithms on a Hadoop MapReduce framework as explained below.

It was reported on [7,18] that BM algorithm shows better performance than KMP algorithm for longer pattern lengths using linear programming models. Figure 1 proves that this is also true with case of Hadoop MapReduce framework

versions of these algorithms. It was also reported in [7] that KMP shows better performance than BM on shorter pattern length using linear algorithms. Our results also prove that this advantage of KMP using shorter pattern length is replicated in a Hadoop MapReduce framework as shown in Fig. 1. However, the most striking insight is that as expected, the FJS algorithm showed much faster pattern matching execution time for both shorter and longer patterns of length on textual data for all different sizes of data comparing KMP and BM using Hadoop MapReduce framework. This proves that FJS algorithm is the optimal solution for pattern matching application for Big Data on Hadoop MapReduce framework.

Figure 2 shows the scaling of all the three algorithms moving from single node Hadoop setup to a multi-node Hadoop setup. This is done using similar machines in single node as well as multi-node configurations as explained in Sects. 4.2 and 4.4 with a standard 3 term pattern. Figure 2 shows around 40% performance improvement for all algorithms from single node (Sect. 4.2) to multi-node (refmulti-com) installation of Hadoop. This result safely proves that scaling of performance in a Hadoop environment is possible as the number of compute nodes increases for the above algorithms using MapReduce programming framework. FJS emerges as the most suitable candidate in this scenario as well.

5 Conclusion and Future Scope

The experiments have clearly proved that Hadoop MapReduce versions of KMP and BM algorithms work efficiently on shorter and longer patterns respectively in a distributed environment. The study shows that FJS is the optimum pattern matching algorithm in a Hadoop distributed environment compared to KMP and BM. It indicates the potential of FJS algorithm to be a solution for many real time Big Data applications like text analytics, information retrieval and DNA pattern matching.

The future scope of this work is an enhancement to FJS algorithm on Hadoop MapReduce framework. For enhancing the FJS method, a Hash Join function can be introduced. The main benefit of using the hash function will be to reduce the number of character comparisons performed by FJS algorithm in each attempt. Thus, it will reduce the required comparison time. The enhanced FJS algorithm can be explored with its feasibility in different system configurations using Hadoop MapReduce framework. The potential of Enhanced FJS algorithm can be further explored using a real-time Big Data application such as twitter data analysis.

Acknowledgement. This work was completed successfully using the infrastructure support provided by Śúnya Labs, Rajagiri School of Engineering and Technology, India.

References

1. Project gutenberg. https://www.gutenberg.org/
2. Ramya, A., Sivasankar, E.: Distributed pattern matching and document analysis on big data using Hadoop MapReduce model. In: International Conference on Parallel and Distributed Grid Computing (2014)
3. Al-Mazroi, A.A., Rashid, N.A.A.: A fast hybrid algorithm for the exact string matching problem. Am. J. Eng. Appl. Sci. **4**(1), 102–107 (2011)
4. Boyer, R.S.: A fast string searching algorithm. Commun. Assoc. Comput. Mach. **20**, 762–772 (1977)
5. Finnegan, M.: Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic. http://www.computerworlduk.com/data/. Accessed 12 Sep 2017
6. Hume, A., Sunday, D.: Fast string searching. Softw.: Pract. Exp. **21**(11), 1221–1248 (1991)
7. Al-Khamaiseh, K., ALShagarin, S.: A survey of string matching algorithms. Int. J. Eng. Res. Appl. **4**, 144–156 (2014). ISSN 2248–9622
8. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977)
9. Kofahi, N., Abusalama, A.: A framework for distributed pattern matching based on multithreading. Int. Arab J. Inf. Technol. **9**(1), 30–38 (2012)
10. Cao, P., Wu, S.: Parallel research on KMP algorithm. In: International Conference on Consumer Electronics, Communications and Networks (CECNet) (2011)
11. Diwate, M.R.B., Alaspurkar, S.J.: Study of different algorithms for pattern matching. Int. J. Adv. Res. Comput. Sci. Softw. Eng. **3**, 1–8 (2013). ISSN 2277 128X
12. Rajesh, S., Prathima, S., Reddy, L.S.S.: Unusual pattern detection in DNA database using KMP algorithm. Int. J. Comput. Appl. **1**(22), 1–5 (2010)
13. Singh, S., Singh, N.: Big data analytics. In: 2012 International Conference on Communication, Information and Computing Technology (ICCICT), 13230053, IEEE, October 2012
14. Singh, A.: New York stock exchange oracle exadata - our journey. http://www.oracle.com/technetwork/database/availability/index.html. Accessed 12 Sep 2017
15. Sardjono, T.A., Al Kindhi, B.: Pattern matching performance comparisons as big data analysis recommendations for hepatitis C virus (HCV) sequence DNA. In: International Conference on Artificial Intelligence, Modelling and Simulation (AIMS) (2015). ISBN 978-1-4673-8675-3
16. Alzoabi, U.S., Alosaimi, N.M., Bedaiwi, A.S.: Parallelization of KMP string matching algorithm. In: World Congress on Computer and Information Technology (WCCIT). INSPEC Accession Number: 13826319 (2013)
17. Vance, A.: Hadoop, a free software program, finds uses beyond search, March 2009. http://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html
18. Vidanagama, D.: A comparative analysis of various string matching algorithms. In: International Research Conference, pp. 54–60 (2015)
19. Franek, F., Jennings, C.G., Smyth, W.F.: A simple fast hybrid pattern-matching algorithm. J. Discrete Algorithms **5**, 682–695 (2007)