Two Way Approach For Matching Regular Expressions Using Multi-Threading And Reversing

Pankaj Panigrahi
Student,CET BBSR

Sibendu Dey
Student,CET BBSR

Anindita pani
Student,CET BBSR

Amitav Mohapatra
Lecturer,CET BBSR

Abstract

Regular expressions are a common choice for defining configurable rules for data parsing because of their expressiveness in detecting recurrent patterns and information. Regular expression matching is the first line of defense in performing online data filtering. Unfortunately, few solutions can keep up with the increasing data rates and the complexity posed by sets with hundreds of expressions. The two most popularly used methods for regular expression matching are pipelined char grid architecture and RE-NFA architecture. In this present scenario a method can be implemented to reduce the number of the regular expressions mismatches. Our model implements the concepts of RE-NFA, multi-threading, semaphores, reversing of regular expression, so as to reduce the time taken to find a regular expression mismatch.

1. Introduction

Regular expressions concisely describe a set of strings without explicitly listing the set content. Each expression consists of one or more strings connected with a set of operators such as alternate (|), which chooses among two strings; repetition (*), which repeats a string zero or more times; and optional (?). Given an input string, a matching operation determines if that string is a possible pattern instance. An example is ABC*D, which recognizes any string that starts with AB, continues with zero or more C, and finishes with D. Sample matching input strings might be ABD, ABCD, or ABCCCD.

Very fast regular expression matching is currently a hot topic in applied research, with more applications searching large pattern sets with increasingly faster data streams [3]. Deep packet inspection is one of the most demanding applications, and regular expressions are a common threat-detection mechanism in both commercial and open source NIDSs.

Regular ex engines typically rely on deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). DFAs, which require only one state traversal per character, yield higher parsing rates when they are not memory constrained. NFAs are more memory efficient; recent cache-friendly implementations of enhanced NFAs can provide gigabit-per-second processing rates with dictionaries that contain hundreds of thousands of keywords.

Our model suggests a method to reduce the time taken to find a regular expression mismatch. A Non-deterministic Finite Automata is used to match a string with a regular expression. We have devised a method in which the regular expression to be matched is reversed and the NFA is created accordingly. String is then parsed in reverse order. The model implements the regular RE-NFA parsing and the proposed model simultaneously using multi-threading. The two process running simultaneously would signal each other in case of any mismatch.

2. Existing Techniques

The two popular methods for matching regular expressions are pipelined char grid architecture and regular expression NFA. Both NFA and DFA are widely used for matching of regular expressions.

2.1 Pipelined char grid architecture

In this method, using techniques from graph theory, the patterns are partitioned n-ways such that the number of repeated characters within a partition is maximized, while the number of characters repeated between partitions is minimized, the system can be composed of n pipelines, each with a minimum of bit lines [1].

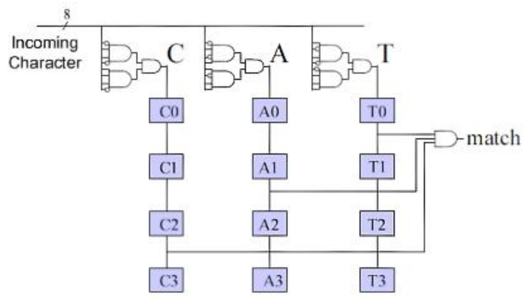

Figure 1 Pipelined Grid Architecture

The unary design utilizes simple pipeline architecture for placing the appropriate bit lines in time (Figure 1). Because of the small number of total bit lines required (generally around 30) and extensive pipelined fan out to the individual comparators, adding delay registers adds little area to the system design. The new design takes the the general brute force matching technique used by UCLA and Crete and moves the character decoding to the first stage in the pipeline, and reduces the overall size of the individual comparators by one-eighth [4]. Each pipeline contains only the characters required by the patterns for which the pipeline is responsible. The length of each pipeline is equal to the length of the longest pattern in the pipeline.

2.2 Regular expression NFA

There are two main approaches for turning regular expressions into equivalent NFA's. One is by Thompson, and the other one is by McNaughton and Yamada[2]. Thompson's construction is a simple, bottom-up method that processes the regular expression and constructs NFA as it is parsed. For regular expression R, the rules for constructing Thompson's NFA MR that accepts LR are as follows: There are exactly one initial and one final state in Thompson's NFA.

Another approach is based on Mc-Naughton and Yamada[2]. In addition to an initial state, McNaughton and Yamada's NFA has a distinct state for every alphabet symbol occurrence in the regular expression. All the edges in McNaughton and Yamada's machine are labelled by alphabet symbols; all the incoming edges of each state are labelled by the same symbol.

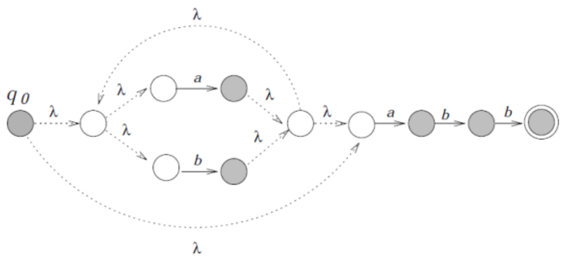

Figure 3 Thompson's NFA Equivalent to (a|b)*abb

Let q1 and q2 be states in a McNaughton and Yamada's NFA. There is a path from transition state q1

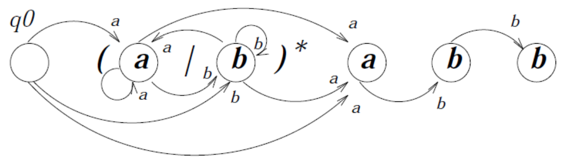

Figure 3 McNaughton and Yamada's NFA equivalent to (a|b)*abbb

to transition state q2 in a Thompson's Machine spelling a if and only if there is an edge labeled a from q1 to q2 in McNaughton and Yamada's corresponding machine. McNaughton and Yamada's corresponding NFA for (a|b)*abb is shown in Fig. 3

3. Our Contribution

our model, we first design the reverse of the regular expression to be matched. For example, if the regular expression (a+2+3*d) which is to be matched; the system generates the reverse of the regular expression which is (d3*2+a+). The actual string to be compared is not reversed; it is only compared in a descending order of index. Two processes start simultaneously.

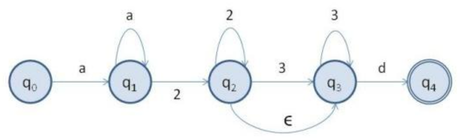

Figure 4 NFA for regular expression a+2+3*d

The first process starts comparing the actual string with the original regular expression. The second process starts comparing the string with the reversed regular expression.

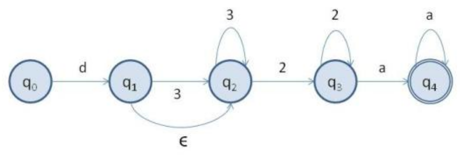

Figure 5 Reverse NFA for the regex

At around mid-way through the string, both of the process will stop matching as the whole string would have been matched till then. As two processes will be running simultaneously matching the provided string, the matching processes will be completed around half of the time required. Now the significant decision is regarding the selection of the midway point where the two processes will stop matching. Our model proposes one point for each process, so as to avoid a loss of regular expression match at the mid-point. Let us assume, the string to be matched is of length 100, the regular expression is of length 6.Thus, the midpoint for process 1 will be 53 and the midpoint for process 2 will be 47. This window of 6 bytes length will ensure that no regular expression is missed at the midpoint. If we take one midpoint for example 50, then it could be possible that a string of 6 bytes length residing at 48, 49, 50, 51, 52, 53 positions would be missed. The proposed model for midpoint is shown in figure 6.

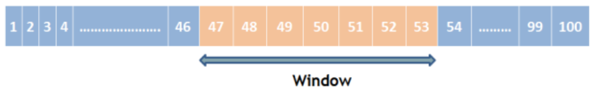

Figure 6 Window for midpoint determination

Our model would be using this technique to match a large string with multiple regular expressions by starting two processes simultaneously. In this type of scenario, the model would exhibit better results.

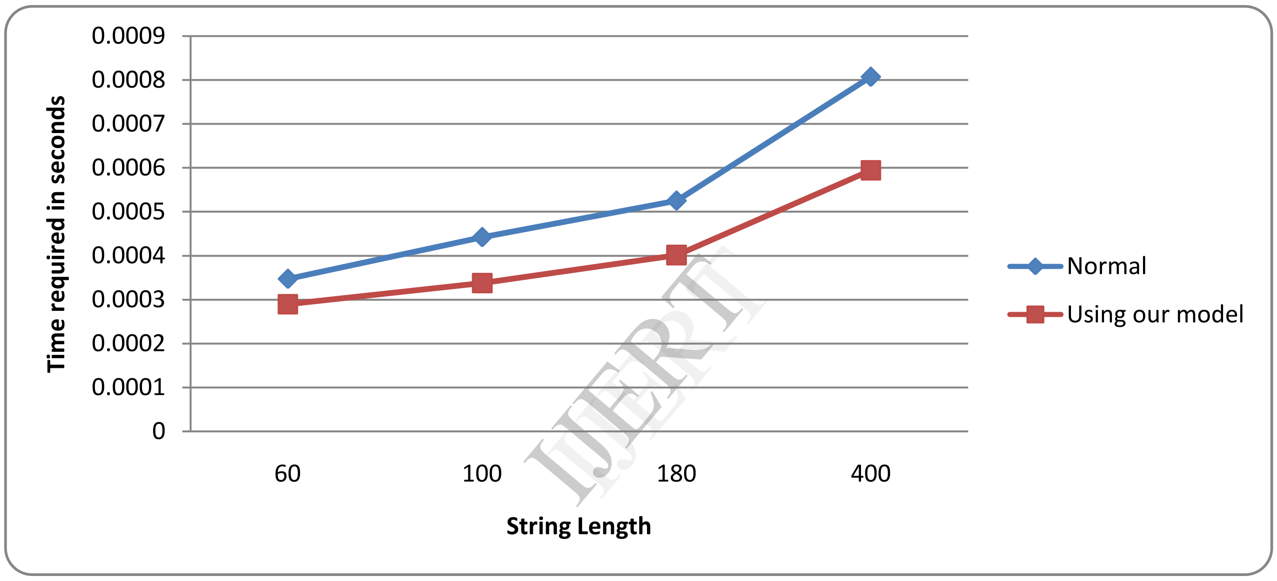

Figure 7 Chart Showing the Experimental Results

4. Experimental Evaluation

We evaluated our model with set of data strings and regular expressions for matching. In our simulation, we have taken few assumptions such as we already provided the second thread process with a reverse string with half number of characters. We performed our simulation using python language with the help of python libraries such as “threading” and “re” for multi-threading and regular expressions respectively. We incorporated a laptop with the following specifications

- Intel core 2 duo 2.6 GHz
- 4 GB RAM
- 32-bit Windows 7 OS
- Python 2.7.3
- Net Beans IDE

We tested the above with four set of strings of different lengths and respective regular expressions for matching. For avoiding complexity the length of the regular expression was fixed at six. For the calculation of time required we used the time module and its clock function. The results we got has been displayed in the below figure-7. The graph clearly shows the difference between the time required by our model is less than the time required by the program normally. We believe that the performance of the model can be enhanced by using a computer with a better configuration. A computer with more number of cores and more main memory would surely give more performance, especially in the case of multi-threading. We also believe our model can be enhanced by using the concept of multi-processing instead of multi-threading.

5. Conclusions and future work

In this paper, we studied the existing techniques for regular expression matching and proposed a model for a faster approach. Our model uses the concept of multi-threading to fasten the process of string matching. The regular expression to be matched is reversed. One process matches the string with the given regular expression while the other process matches .The main distinguishing factor of our model is the degree to which the RE matchers are enforced to prune the search time for large input data set. In real time applications such as deep packet inspection, we need to search large pattern sets. In such cases, the proposed model would exhibit better results.

We believe that optimization in searching of regular expression is an important and fertile area of research. Numerous fields of application would be benefited from the research. Although much research has not been completed on the above said multi-threading matching technique, there is scope for further research.

6 Acknowledgements

This work is supported by college of engineering and technology. The authors would like to thank their family and friends for their support. The authors would also like thank all the faculties of Computer science department of CET Bhubaneswar for their help.

7 References

[1] A Scalable Hybrid Regular Expression Pattern Matcher - James Moscola, Young H. Cho, John W. Lockwood Department of Computer Science and Engineering Washington University
[2] From Regular Expression to DFA's Using Compressed NFA's by Chia-Hsiang Chang.
[3] Huei Lee, Hardware Architecture for high performance Regular expression matching IEEE Transactions on Computers, VOL 58, JULY 2009
[4] A Compact Architecture for High-Throughput regular Expression Matching on FPGA by Yi-HUA E. Yang, Weirong. Jiang and Viktor K. Prasanna
[5] A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs by Zachary K. Baker and Viktor K. Prasanna.