ParaRegex: Towards Fast Regular Expression Matching in Parallel

Zhe Fu[*†], Zhi Liu[*†], and Jun Li[†‡]
[*]Department of Automation, Tsinghua University, China
[†]Research Institute of Information Technology, Tsinghua University, China
[‡]Tsinghua National Lab for Information Science and Technology, China
{fu-z13, zhi-liu12}@mails.tsinghua.edu.cn, junl@tsinghua.edu.cn

ABSTRACT

In this paper, we propose ParaRegex, a novel approach for fast parallel regular expression matching. ParaRegex is a framework that implements data-parallel regular expression matching for deterministic finite automaton based methods. Experimental evaluation shows that ParaRegex produces a fast matching engine with speeds of up to 6 times compared to sequential implementations on a commodity 8-thread workstation.

Keywords

Deep inspection, Regular expression matching, DFA, Parallelism

1. INTRODUCTION AND MOTIVATION

Regular expression (regex) matching has been widely used in today's network security systems, where the payloads of network packets are matched against a set of rules specified by regular expressions. To perform regex matching, regexes are constructed to Nondeterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA). DFA becomes the prior choice for practical time-sensitive applications because it requires only one state transition lookup per input character and is hence fast. However, due to the increasing number of rules and the complicated semantics of regular expressions, state-of-the-art regex matching techniques hardly meet the demands of network speed.

Parallelism is becoming more and more popular and important, which produces new ideas to solve the performance bottlenecks of regex matching. In order to parallelize the regex matching, the input data can be partitioned into several segments and assigned to each thread. For all but the first input segment, the DFA state at the beginning is uncertain. For example, the initial state of the DFA for the second input data segment is determined by the final state of the DFA for the first segment. The basic idea of existing parallel implementations is that starting from the set of all initial states, each thread computes the sub-result based on the input of each segment independently, and then the sub-results of all the threads can be reduced by joining them in sequential order [1, 2, 3].

Obviously, the huge overhead of this speculative implementation introduces significant computation load. Given a DFA with $|Q|$ states, input data with size m, and n processing threads, the time complexity of mapping and matching procedure is $O(|Q|m/n)$ and reducing procedure is $O(|Q|n)$. This reveals that these parallel implementations based on enumeration fail to obtain higher matching speed than the sequential implementation when the DFA is large.

However, the scenario will be quite different when considering with the input data. In this paper, the states that need to be traversed from are defined as *active* states. In a real-world situation, the simultaneously *active* states tend to move to very few states after reading any character, which means that the number of concurrent *active* states is likely to decrease sharply under several arbitrary input characters. As shown in Table 1, the average number of *active* states after one arbitrary input character is less than one percent of the total states number of the DFA. We define the reduction of the number of *active* states as states *convergence*, which brings hope for efficient parallelization of regex matching.

Table 1: Average number of *active* states of four different rulesets after any 1 and 2 input

Ruleset	bro50	bro217	snort24	snort34
Number of regexes	50	217	24	34
DFA states	667	8094	8630	10212
after any 1 input	4.60	37.63	62.42	78.70
after any 2 input	1.02	5.17	19.19	31.69

2. DESIGN AND IMPLEMENTATION

ParaRegex is proposed to implement fast parallel regex matching with low overhead, with MSU (Middle State Unit) as its key structure. MSU consists of two parts: a state and a mapping vector. The state denotes the ID of an *active* state, and the mapping vector is a bit vector that associates the original initial states to the state of this MSU.

Figure 1(a) and 1(b) explain how ParaRegex works in practice. As shown in Fig.1(a), there are two input segments S_k and S_{k+1}. Initially, each original state corresponds to an MSU, and the mapping vector of the MSU indicates which state has been traversed from. The first bit of the first MSU's mapping vector is set to 1 while others are set to 0,

ANCS '16 March 17-18, 2016, Santa Clara, CA, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4183-7/16/03.
DOI: http://dx.doi.org/10.1145/2881025.2889478

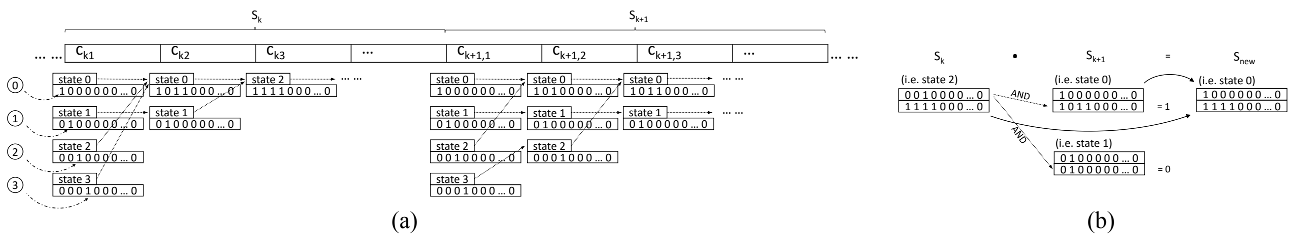

Figure 1: (a) Mapping, matching and (b) reduction procedure of ParaRegex using MSUs

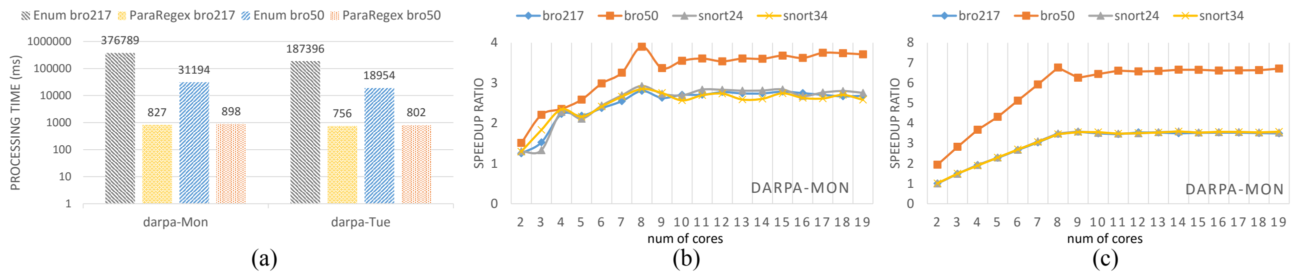

Figure 2: Comparison of ParaRegex and enumeration approach (a) overhead (b) vs. DFA (c) vs. D^{2}FA

denoting that this MSU derives from state 0. After reading the input character C_{k1} from the input segment S_k, state 0, state 2 and state 3 all jump to state 0, so the first, third and fourth MSUs are merged into one MSU whose state is state 0 and mapping vector is the union of MSU 0's, MSU 2's and MSU 3's mapping vectors.

Once all threads have completed their tasks, the set of MSUs corresponding to each segment would be reduced. The state in each MSU is encoded to a bit vector named state vector, and then the previous MSU's state vector performs an AND operation with the latter MSU's mapping vector. If the result of AND operation is 1, the two MSUs are supposed to be reduced into one which is composed of the previous MSU's mapping vector and the latter MSU's state.

Benefiting from the fast OR and AND operations of bit vectors, the processing of multiple MSUs can be very efficient. It must be noted that ParaRegex does not modify or create new DFAs, but just provides a general mechanism that is orthogonal to other work. In other words, state-of-the-art work on regex matching can be easily parallelized using ParaRegex by replacing original states with MSUs or just attaching a mapping vector to the original state.

3. EVALUATION

We carry out the preliminary evaluation with an Intel Core i7-4790 CPU (4 cores with 8 threads), and use the Regular Expression Processor [4] as the basic implementation of regex matching and pthread for the thread library. Four rulesets picked from Bro and Snort are tested (Table 1), while the Darpa network traffic is treated as the input data.

We compare ParaRegex to a general enumeration approach [1, 2, 3]. Figure 2(a) shows the matching time on different rulesets and traffic. By introducing the MSU structure, the processing speed of ParaRegex is at least one to two orders of magnitudes faster than that of enumeration approach. Figure 2(b) shows the speed improvement of ParaRegex on different rulesets, treating [4] as a baseline. As the number of threads in use increases, the matching speed of ParaRegex grows and maximum speed is obtained when 8 threads process simultaneously in parallel. We also apply ParaRegex to D^{2}FA [5] and gain up to 6 times speedup (Fig. 2(c)). More experiments on other rulesets and traffic, which are omitted from this paper, draw the similar conclusion.

4. CONCLUSIONS AND FUTURE WORK

This paper introduces ParaRegex, a framework orthogonal to state-of-art DFA-based regular expression matching methods. ParaRegex employs MSUs to implement low-overhead and high-efficiency parallel matching engine with nearly linear speed improvement. Our future work will focus on conducting experiments on other distributed processing platforms like Hadoop or Spark, and further parallelize multiple *active* MSUs by specific hardware. We hope all these platforms and algorithms can effectively work together to achieve high performance regular expression matching in parallel.

5. REFERENCES

[1] J. Holub and S. Štekr. On parallel implementations of deterministic finite automata. In *Implementation and Application of Automata*, pages 54–64, 2009.

[2] Y. Ko, M. Jung, Y.-S. Han, and B. Burgstaller. A speculative parallel dfa membership test for multicore, simd and cloud computing environments. *International Journal of Parallel Programming*, 42(3):456–489, 2014.

[3] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[4] Regular expression processor. http://regex.wustl.edu/.

[5] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 145–154, 2007.