Efficient Parallelization of Regular Expression Matching for Deep Inspection

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
Email: {fu-z13, zhi-liu12}@mails.tsinghua.edu.cn, junl@tsinghua.edu.cn

***Abstract*—Regular expression 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. Due to the increasing number of rules and the complex semantics of regular expressions, state-of-the-art regular expression matching techniques hardly meet the demands of network development. The rapid growth of parallel technology calls for an efficient parallel regular expression matching method. In this paper, we propose ParaRegex, a novel approach for fast parallel regular expression matching with high efficiency and low overhead. ParaRegex is a framework that implements data-parallel regular expression matching for finite automaton based methods. Experimental evaluation shows that ParaRegex produces a high-performance regular expression matching engine with low memory overhead and linear speed-up ratio, and obtains up to 6 times faster processing speed on a commodity multi-core workstation.**

I. Introduction

Deep Inspection, also known as complete packet inspection or payload scanning of network traffic, is now playing a vital role in network security. Given a set of predefined rules, the payloads of network packets are scanned to identify the potential security threats, including viruses, intrusions, spams, data leakage, and so forth.

In the early stages, exact strings are used to characterize the threat patterns in deep inspection systems. Knuth-Morris-Pratt (KMP) [1], Boyer-Moore (BM) [2], Aho-Corasick (AC) [3] and Wu-Manber (WM) [4] are classical algorithms which are designed to implement fast string matching. However, as the threat patterns are getting more and more complex, exact strings can hardly describe security threats.

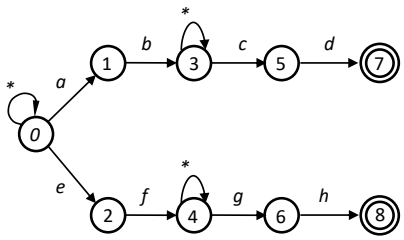
Due to the rich expressiveness and powerful flexibility, regular expressions (regexes) become more popular and have been widely used in today's deep inspection systems. For instance, the regular expression ".*seclog_[a-z]{5}\d{4}_\d{10}\x2Ekcb" matches any payload consisting of the string "seclog", the underscore, five any lowercase letters, four any numbers, the underscore, ten any numbers, the dot and the string "kcb", describing the occurrence of backdoor Backdoor.Win32.Qakbot.E [5].

To perform regular expression matching, regular expressions are compiled to Nondeterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA). Either NFA or DFA has its strengths and weaknesses. NFA has fewer states and transitions, and its space cost only linearly depends on the size of regular expression ruleset; thus it is space-efficient. However, NFA can hardly guarantee the performance in the worst case. In other words, attackers may maliciously generate specific network traffic which would significantly degrade the performance of deep inspection systems [6]. On the other side, DFA is always fast, and hence becomes the prior choice for practical time-sensitive applications. But the well-known state explosion problem makes DFA require excessive memory consumption in practice, especially when the ruleset is large.

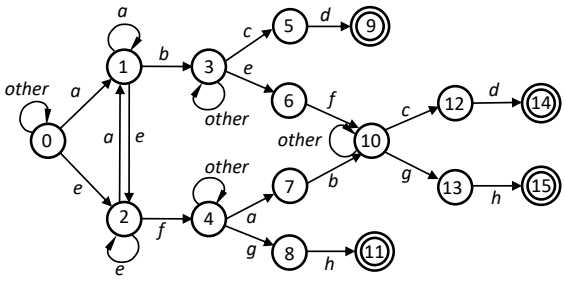
The booming development of network technology presents serious challenges for regular expression matching. First, the size of rulesets in practical use keeps increasing. Second, the semantics of regular expressions in the rulesets are getting more and more complex. As the network traffic bandwidth grows rapidly, state-of-the-art regular expression matching techniques hardly meet the demands and have become the bottleneck for high-performance content-aware network devices. Parallel computing becomes more and more popular and important with the growth of multi-core technology, which produces new ideas to solve the performance bottlenecks of regular expression matching by fast and efficient parallelization. However, today's parallel implementations of regular expression matching are either brute-force or with huge overhead on practical datasets.

This paper makes the following contributions. First, the *states aggregation* phenomenon in DFA states transitions is discovered, which brings hope for high-efficiency and low-overhead parallelization of regular expression matching. Second, we propose ParaRegex, a framework that implements data-parallel regular expression matching on existing Finite Automaton (FA). Third, two optimizations for ParaRegex, called *Smart Split* and *Quick Start*, are designed to further lower the overhead and accelerate the matching speed. Finally, experiments on real-world rulesets and network traffic are conducted, and the results show that ParaRegex can achieve up to 6 times faster processing speed compared to sequential implementations on a workstation with an Intel Core i7-4790 CPU.

The rest of the paper is organized as follows. Section II presents our motivations, and Section III describes the design details and two optimization methods of ParaRegex. Section IV evaluates the performance of ParaRegex and compares it to some state-of-the-art solutions. Section V states related work,



(1) NFA



(2) DFA

Figure 1. NFA and DFA of regular expression "ab.*cd" and "ef.*gh"

and Section VI concludes with a summary and future research directions.

II. Background and Motivation

A. Regular Expression Matching and Finite Automaton

Regular expression was first proposed in [7]. It consists of a sequence of ASCII characters and meta-characters. The meta-character, including quantification (such as " . " and " * "), position (such as " ^ " and " $ ") and character class, gives regular expression the power of representing a set of exact strings instead of a single exact string.

Nondeterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA) are two equivalent descriptions of regular expressions. Both of NFA and DFA are 5-tuples ($\{Q$, Σ, δ, q_0, $F\}$), where Q denotes a finite set of states, Σ denotes a finite set of input symbols, δ denotes a transition function, q_0 denotes the start state in Q, and F denotes a set of accepting states. The only difference between NFA and DFA is that in a DFA the transition function δ only takes one state and returns a single state for an input symbol, while in an NFA δ may return multiple states.

Figure 1 shows the NFA and DFA of the regular expression "ab.*cd" and "ef.*gh" (some transitions in DFA are omitted). As we can see from the figure, NFA has succinct data structures and hence small size. For N regular expressions with an average length of L, the space complexity of NFA is $O(N \times L)$. Nevertheless, the space-efficiency of NFA is at the cost of slow running speed, where a mass of state transitions need to be processed concurrently for each input character. For example, under the circumstance that state 3 is *active* and input character is c, both state 3 and state 5 will

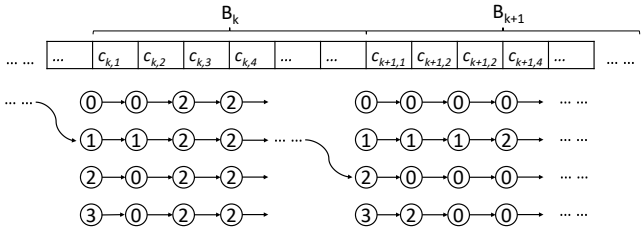


Figure 2. Example of enumeration methods of regular expression matching

be active. Therefore, two states need to be processed for next input character. In the worst case, for N regular expressions with an average length of L, the time complexity of NFA is $O(N \times L^2)$. Compared with NFA, DFA always requires only one state transition lookup per input character, so the time complexity of DFA is $O(1)$. However, DFA often suffers from the well-known state explosion problem, that is, the size of a DFA often grows exponentially along with the increase of the regular expression set size, and the space cost in worst case is $O(|\Sigma|^{N \times L})$ (where $|\Sigma|$ is the size of the alphabet) [8].

In spite of the poor space complexity of DFA, the relatively high matching efficiency and stable throughput that is independent of network traffic and rulesets enable DFA to provide wire-speed and deterministic processing rate, which could meet the requirements of real-time deep inspection. For this reason, our work is focused on the parallel implementation of regular expression matching using DFA for deep inspection applications.

B. Parallel Regular Expression Matching

An intuitive approach to parallelizing regular expression matching is to divide the input data into multiple blocks and match each data blocks separately. However, the strong data dependency of each block makes it hard to obtain the correct final matching result. Taking two divided data blocks as an example, the start state of the DFA for the second block is unknown until the first block finishes the matching process. If we want to match these two blocks in parallel and regard each block as independent input data, one situation could happen that one part of an attack pattern is in the first data block and the other part is placed into the second data block. As a result, this attack pattern would be missed, which is a critical and unaccepted error for the deep inspection systems.

There have been several proposals on the parallel matching of regular expressions. One way is to speculate or guess the start state of each data block. After the previous data block finishes matching process, the algorithm needs to check whether the predicted DFA start state of the following data block is identical to the correct final state of the previous one. It not, the following data block needs to be re-matched with the correct start state. Frequent re-matching of divided data block would degrade the performance, in which case the throughput cannot be guaranteed. Thus, speculation based methods are not suitable for regular expression matching in deep inspection systems.

Algorithm 1: Enumeration Methods

Input : DFA D = {Q, Σ, δ, q_0, F}, Input Data C = $c_{1,1}c_{1,2}...c_{1,m_1}c_{2,1}...c_{2,m_2}...c_{n,1}c_{n,m_n}$, number of threads n

Output: q_{final} that $q_0 \xrightarrow[C]{D} q_{final}$

```
// mapping and matching procedure
foreach i ∈ [1...n] parallel do
    S_i ← Q                    // sub-result of block i
    for j = 0 → m_i do
        foreach q ∈ S_i do
            S_i[q] ← δ(S_i[q], c_ij)
        end
    end
end
// reducing procedure
q' ← q_0
for i = 1 → n − 1 do
    q' ← S_i[q']
end
q_final ← S_n[q']
```

Another approach is the enumeration method, *i.e.*, enumerating all the transitions from every possible start state of each data block simultaneously [9], [10], [11]. Figure 2 and Algorithm 1 illustrate the idea of existing enumeration methods of parallel regular expression matching. Similar to the MapReduce model [12], the procedure can be divided into 3 steps: map, match and reduce. Starting from the set of all start states, each thread computes the sub-result based on the input of each data block independently, and then the sub-results of all threads can be reduced by joining them in sequential, to obtain the final matching result.

Obliviously, the huge overhead of this approach introduces significant computation load, making it not an efficient solution for practical use in deep inspection systems. Let D be the DFA, $|Q|$ be the number of states of the DFA, m be the size of the input data, and n be the number of threads. The time complexity of mapping and matching procedure of Algorithm 1 is $O(|Q|m/n)$, and the time complexity of reducing procedure is $O(|Q|n)$ when a sequential reduction is used [9]. This reveals that parallel implementation of regular expression matching based on enumeration fails to obtain higher matching performance than the non-parallel implementation when the DFA is large (*i.e.*, $m \gg n$).

C. States Aggregation in DFA Transitions

However, the scenario will be quite different when taking the input data into consideration. The DFA 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 input character, which means that the number of concurrent *active* states is likely to decrease sharply under several arbitrary input characters. We define the reduction of the number of *active* states as

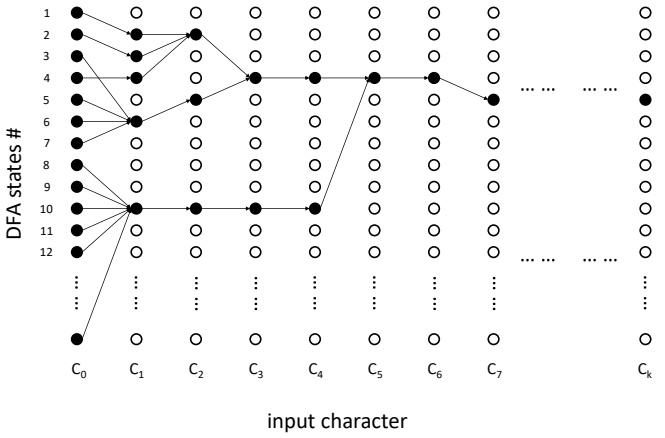


Figure 3. *aggregation* phenomenon of DFA states on real-world traffic

Table I
AVERAGE NUMBER OF *ACTIVE* STATES FOR FOUR DIFFERENT RULESETS

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

states aggregation. Figure 3 shows a schematic of the *states aggregation* phenomenon in the DFA constructed from regular expressions from Snort [5] ruleset along with the input traffic dumped from our campus network. In the beginning, all DFA states are set to *active* (denoted graphically by black circles in Fig. 3). After reading input character, the states move according to the transition function δ. As can been seen, state 3, state 5, state 6 and state 7 all move to state 6, and state 8 to the last state all move to state 10 upon reading the input character c_0. After 5 characters, only state 4 is *active*, and the number of *active* states remains one in the following matching procedure.

For snort24, a regular expression ruleset which contains 24 regular expressions, the number of all *active* states decreases from 8630 to 18 after reading just one input character from the dumped campus network traffic. To further support this notion, four different rulesets are tested with all possible inputs which consisted of any one or two input characters from the alphabet. To put it another way, we treat all 256 characters from ASCII Character Set [13] as one-character input, and 65536 various combinations of any two characters as two-character input. The average numbers of *active* states after all one-character and two-character input are counted in Table I. As shown in Table I, the average number of *active* states after one arbitrary input character is less than one percent of the number of total DFA states, and become even less after two arbitrary input characters.

The nature of DFA offers the theoretical basis for this observation. DFA only activates one state and requires exactly one state traversal per input character. As a result, the number of *active* states after one input character would be no more than

that before. Mathematically, we have the following Theorem 1.

Theorem 1. *Let Q_0 be the set of all states of DFA D, Q_m be the set of* active *DFA states after reading m input character when all DFA states are* active *as start states, then $\forall\ 0 \leq i < j \leq m$, we have $|Q_i| \geq |Q_j|$.*

Proof. Assume that $\exists\ 0 \leq i < j \leq m$, s.t. $|Q_i| < |Q_j|$. Then there must $\exists$ k, $i \leq k < k+1 \leq j$, s.t. $|Q_k| < |Q_{k+1}|$. This means, there must exist at least one DFA state, *e.g.* $\hat{q}$, s.t. $|\delta(\hat{q})| > 1$. This contradicts the fact that the transition function δ of DFA only takes and returns a single state and thus contradicts the assumption. □

Theorem 1 lays an essential theoretical foundation for the DFA *states aggregation*. Furthermore, the profiling of DFA transitions and input network traffic in real-world use also contributes to the *aggregation* of DFA states. For the DFA compiled from snort24, the average distinct transitions for each state is 14.18, and this number falls to only 3.29 for English letters, numbers and common symbols in the input traffic. And on average 90.8% of the states move to the same next state under the circumstance of evenly distributed input. On the other side, the character which leads to transitions to seldom-active states is extremely rare in real-world network traffic. This means that the set of *active* states would *aggregate* rapidly for real-world traffic.

Based on the analysis above, we propose a novel structure to implement fast parallel regular expression matching with low overhead in the next section. Two additional optimization techniques are suggested to further improve the matching efficiency.

III. DESIGN AND OPTIMIZATION

A. Design Details

1) MSU and ParaRegex: Depending on the *state aggregation* phenomenon in the DFA state traversals, we propose ParaRegex, a framework that implements low-overhead and high-efficiency parallel regular expression matching. Middle State Unit (MSU) is the fundamental structure of ParaRegex. MSU consists of two parts: a state id and a mapping vector. The state id of MSU in ParaRegex is identical with the original state id in a DFA, and the mapping vector is a bit vector that maintains mapping relationships between original start states and the state of this MSU. Similar to DFA, ParaRegex is defined formally as the following:

ParaRegex has 5-tuple $\{M, \Sigma, \delta, M_0, F_M\}$, consisting of

- M: a finite set of MSUs
- Σ : a finite set of input symbols called the alphabet
- $\delta : M \times \Sigma \rightarrow M$: a transition function
- $M_0 \subseteq M$: a set of initial or start MSUs
- $F_M \subseteq M$: a set of accepting MSUs

The design of MSU gives ParaRegex the power of efficient regular expression matching in parallel. ParaRegex performs the following steps as described in Algorithm 2. First, the input data is split into n data blocks, where n matches the number of threads that a platform supports. Second, all threads start with the MSU set M_0, and traverse the input character of each data block independently. When the task of each thread is completed, the number of MSUs will decrease to only one or other rather small number due to the *state aggregation*. Finally, all MSU sets of all threads are reduced to obtain the final result. Next part in this section will dig into the technical details of mapping, matching and reducing procedure of ParaRegex.

Algorithm 2: ParaRegex implementation

Input : ParaRegex P = $\{M, \Sigma, \delta, M_0, F_M\}$, input $C = c_{11}c_{12}...c_{1m_1}c_{21}...c_{2m_2}...c_{n1}c_{nm_n}$, number of threads n

Output: q_{final}

```
1  // mapping and matching procedure;
2  foreach i ∈ [1...n] parallel do
3      if i == 1 then
4          M_i ← q_0
5      else
6          M_i ← M_0
7      end
8      for j = 1 → m_i do
9          foreach msu ∈ M_i do
10             msu ← δ(msu, c_ij)
11         end
12         merge MSUs with the same msu.id
13     end
14 end
15 // reducing procedure;
16 msu_final ← M_1
17 for i = 2 → n do
18     foreach msu ∈ M_i do
19         if msu_final.id && msu.mapping ≠ 0 then
20             msu_final.id ← msu.id
21             break
22         end
23     end
24 end
25 q_final ← msu_final.id
```

2) Initialization: The operation of mapping DFA state to MSU only have to be executed when all MSUs are initialized. In a DFA with $|Q|$ states, state k ($0 \leq k < |Q|$) is attached to a $|Q|$-bit vector, in which the kth bit is set to 1 and others are 0. After initialization, ParaRegex has a set of $|Q|$ MSUs, instead of $|Q|$ DFA states. It must be noted that the first set only has one MSU because the start state of the first block is determined, which is definitely q_0. Line 3 to line 7 of Algorithm 2 reveals the initial mapping procedure, where M_0 denotes the set of MSUs corresponding to all DFA states, and M_i ($1 \leq i < n$) denotes the set of MSUs of data block i. Obviously, we have the following Theorem 2.

Theorem 2. *For a set of MSUs $M = \{msu_0, msu_1, ..., msu_i, ..., msu_{n-1}\}$, $0 \leq i < n$, we have $\bigcup_{i=0}^{n-1} msu_i.mapping = 1$*

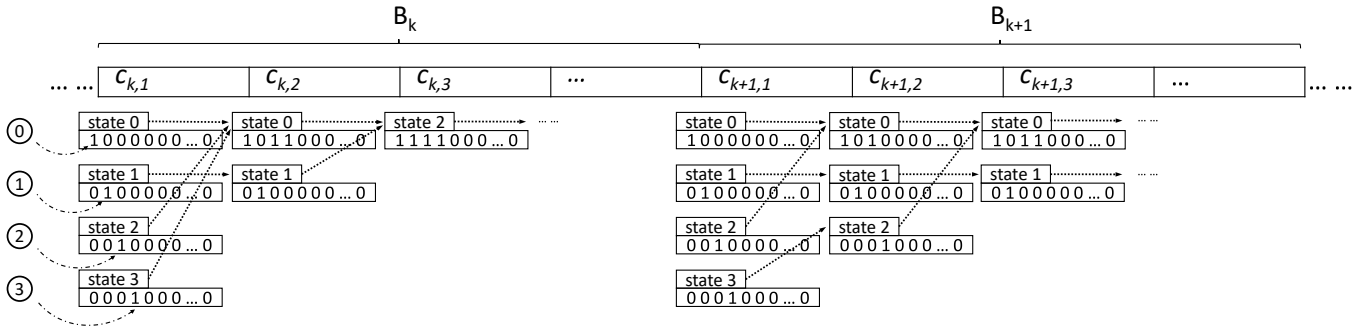


Figure 4. Mapping and matching procedure of ParaRegex using MSUs

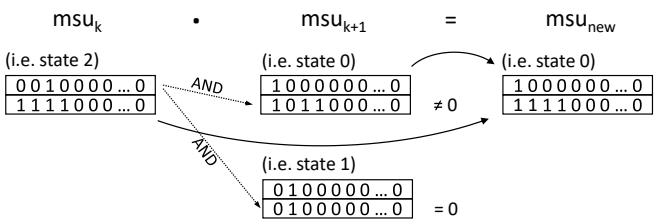


Figure 5. Reduction procedure of ParaRegex

and $\forall\ 0 \leq i < j < n, msu_i.mapping \cap msu_j.mapping = 0$.

Proof. According to the definition of MSU, every bit of value 1 in the mapping vector represents a unique DFA state, so the union of all mapping vector is a bit vector where all bits are 1. On the other hand, no DFA state can move to more than one state, thus the intersection of any two mapping vectors is always 0. □

3) Matching: Line 8 to line 13 of Algorithm 2 shows the matching procedure of each data block in ParaRegex. Each thread corresponds to a set of MSUs. In these sets, every MSU's state moves to the next according to the transition function. After all MSUs in one set complete state transitions, the MSUs with the same state id are merged. In other words, for MSU msu_i and msu_j, $0 \leq i, j < n$, if $msu_i.id == msu_j.id$, then the mapping vector of merged MSU is $msu_i.mapping$ && $msu_j.mapping$. The merging process can also be performed using a hash-based method, in order to reduce the merge complexity from $O(n^2)$ to $O(n)$. We have the following Theorem 3, which indicates the similar *aggregation* property of MSU.

Theorem 3. *Let M_0 be the set of MSUs, M_m be the set of MSUs after reading m input character, then $\forall\ 0 \leq i < j \leq m$, we have $|M_i| \geq |M_j|$.*

Proof. Since the state in MSU is identical to the original DFA state, this theorem can be derived directly Theorem 1. □

4) Reducing: The reducing operation is executed only after all threads complete the matching process. Line 16 to line 24 of Algorithm 2 explains a sequential reduction method. The $msu.state$ field can be transformed into another $|Q|$-bit vector, where the only $msu.state$th bit is set to 1 and others are 0. At first, the only MSU in M_1 is denoted as msu_{final}. From the second set, if the intersection between $msu_{final}.id$ and one $msu.mapping$ from the set of MSUs is not 0, then it means that starting from $msu_{final}.id$, the DFA state would finally move to $msu.id$ for this input data block. Therefore, the $msu_{final}.id$ becomes $msu.id$ as a combinational result. Theorem 4 below assures that there is one and only one msu from the set that meets the condition in line 19 of Algorithm 2. In the end, the final msu is obtained from which we can get the matching result.

Theorem 4. *Let msu_{final} be the combinational matching result of input data block $B_0, ..., B_k$, $0 \leq k < n$, M_{k+1} be the set of MSUs corresponding to input data block B_{k+1}, then* ∃! *msu* $\in M_{k+1}$ *s.t.* $msu_{final}.id$ && $msu.mapping \neq 0$.

Proof. Assume that $\nexists msu \in M_{k+1}$, s.t. $msu_{final}.id$ && $msu.mapping \neq 0$, *i.e.*, $\forall msu \in M_{k+1}$, $msu_{final}.id$ && $msu.mapping == 0$. Let $m = msu.state$, because only mth bit in $msu.state$ is 1 and others are 0, this means that $\forall msu \in M_{k+1}$, the mth bits are 0. As a result, $\bigcup_{msu \in M_{k+1}} \neq 1$, which contradicts Theorem 2. On the other side, assume that ∃ two or more msu $\in M_{k+1}$ s.t. $msu_{final}.id$ && $msu.mapping \neq 0$, *i.e.*, ∃ msu_i and msu_j, $0 \leq i, j < n$, the mth bits of $msu_i.mapping$ and $msu_j.mapping$ are 1, so $msu_i.mapping \cap msu_j.mapping \neq 0$, which contradicts Theorem 2. □

Figure 4 and Figure 5 explain how ParaRegex works in practice. As shown in Fig. 4, there are two input data blocks B_k and B_{k+1}. Initially, each original DFA 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, denoting that this MSU derives from DFA state 0. After reading the input character $c_{k,1}$ from the input data block B_k, state 0, state 2 and state 3 all move to state 0, so the first, third and fourth MSU are merged into one MSU whose state id is 0 and

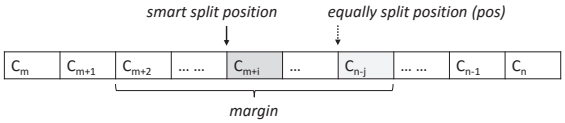


Figure 6. Example of the data partitioning optimization

mapping vector is the union of MSU 0's, MSU 2's and MSU 3's mapping vectors. The quick *OR* operations of bit vectors accelerate the merging of MSUs.

Once all threads have completed their tasks, the set of MSUs corresponding to each data block would be reduced. Figure 5 illustrates a fast reducing example. 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 the *AND* operation is 1, the two MSUs are to be reduced into one which is composed of the previous MSU's mapping vector and the following MSU's state vector. Benefiting from the fast *OR* and *AND* operations of bit vectors, the processing of multiple MSUs can be very efficient.

Accepted rules can be recorded at line 10 in Algorithm 2 to ensure that the same and complete information including the matched rules and positions would be obtained by ParaRegex as sequential implementations do. 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 such as $\mathrm{D}^2\mathrm{FA}$ [14]. In other words, state-of-the-art work on regular expression matching can be easily parallelized using ParaRegex by replacing original states with MSUs or just attaching a mapping vector to the original state. Section IV will present the experimental results of both DFA and $\mathrm{D}^2\mathrm{FA}$.

B. Optimizations

In this section, the basic ParaRegex method is optimized for more efficient implementation. One optimization is to split the input data more smartly for faster *aggregation* speed, and the other optimization is to reduce the memory consumption at the start stage of ParaRegex.

1) Smart Split: Splitting the input data equally seems to be the fairest way for balancing each thread's load. However, for a given DFA, the *state aggregation* situation varies with the input character. A smart split position would improve the *aggregation* speed and hence decrease the computation complexity.

To optimize the data partitioning for faster *aggregation*, we define the *Aggregation Factor (AF)* for each input character. Starting from all $|q|$ states of a DFA, the simultaneously *active* states number decrease to $|q'|$ after reading character c, so the *Aggregation Factor* of character c is defined as

$$AF(c) = 1 - |q'|/|q|$$

It is obvious to know that larger AF indicates faster *aggregation* ability. Since the alphabet and DFA are given before the matching procedure, the computation of *AFs* of all characters can be performed in a preprocessing stage. When splitting input data, let pos denotes the equally split position, $margin$ be an adjustable parameter, then the character with the biggest *AFs* in the range $[\ pos - margin,\ pos\ ]$ suggests a better split position.

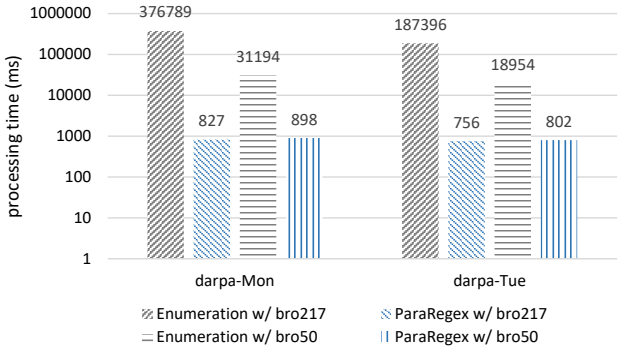


Figure 7. Overhead comparison of ParaRegex and enumeration approach

Figure 6 shows a simple example of the *Smart Split*. The AF of character c_{m+i} is the largest among the characters from c_{m+2} to c_{n-j}, so split the input data right before c_{m+i} is a smarter choice. Because the size of $margin$ is ignorable compared to the size of input data block, the load for each thread stays nearly the same while better *aggregation* abilities are gained.

2) Quick Start: As mentioned previously, the huge overhead caused by parallelization is only at the start stage of ParaRegex. After 2 or 3 characters, the amount of MSUs would decrease to a very small number which brings negligible extra memory consumption. To avoid the expensive overhead, an index table of the first few start states which consists of all possible combinations of any k characters could be built in advance. Then each thread first reads k input character from the data block and find the corresponding result of *active* MSUs directly.

As with *Smart Split*, the *Quick Start* optimization can be also accomplished in a preprocessing stage. For a DFA with $|Q|$ states, let k be the number of pre-computed characters, so the extra space for the index table is $|Q| \times \Sigma^k$, which could save $k \times |Q|^2$ memory consumption. It's always a trade-off how to choose the right k: larger k results in quicker start for ParaRegex, but the extra memory consumption would be even more than the basic ParaRegex implementation without *Quick Start* optimization. Empirically, k of 1 or 2 will be an appropriate choice to relieve the overhead at the start stage. Section IV-B shows the experimental result of this optimization on memory usage reduction when $k = 1$.

IV. Evaluation

In this section, we conduct a series of experiments to evaluate the performance of ParaRegex and the optimizations. Experiments are performed on a workstation with Intel Core i7-4790 CPU (4 cores with 8 threads). Regular Expression

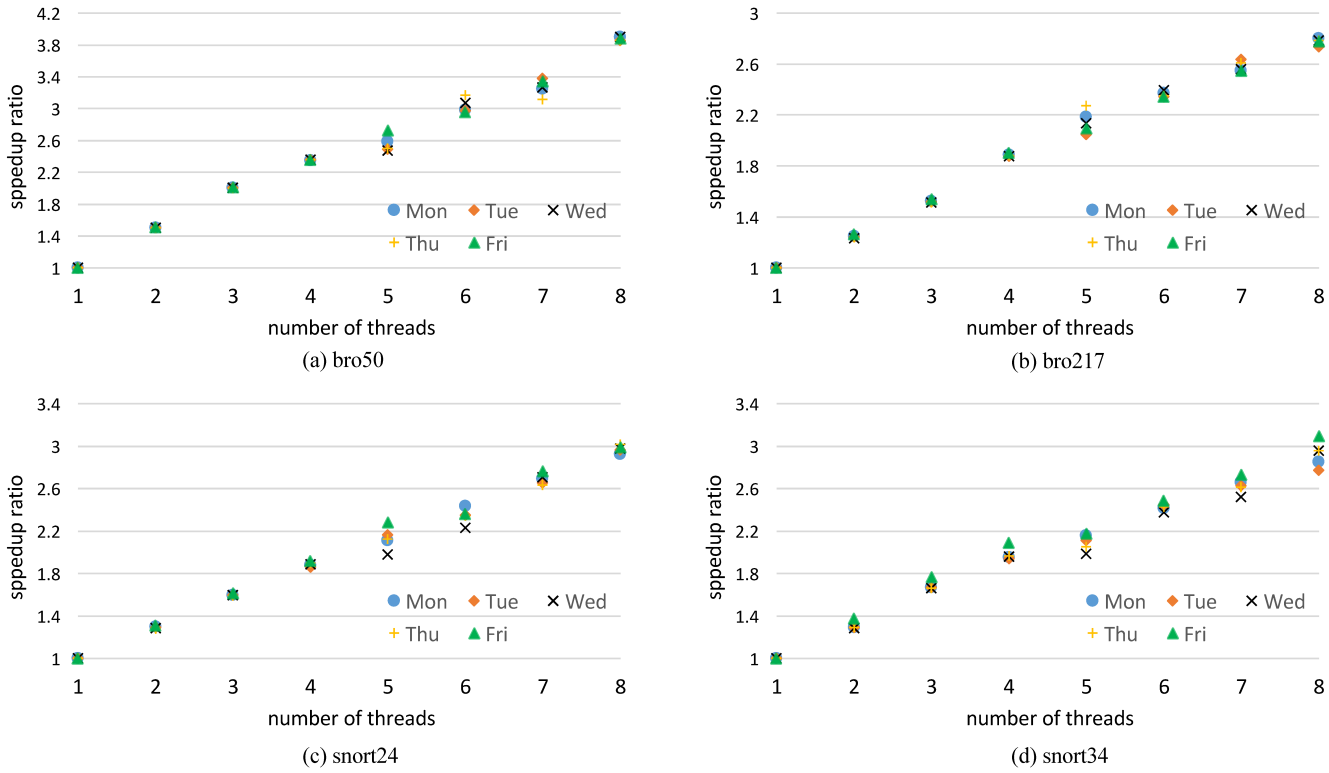


Figure 8. Speedup ratios of ParaRegex on different network traffic and rulesets

Table II
DARPA DATASET USED IN THE EVALUATION

Traffic	Mon	Tue	Wed	Thu	Fri
Size (MB)	140	125	168	146	135

Table III
MEMORY CONSUMPTION OF DFA, PARAREGEX WITHOUT AND WITH QUICK START (QS) OPTIMIZATION

Number of input char	1	2	3	4	5
DFA (MB)	10.18	10.18	10.18	10.18	10.18
ParaRegex w/o QS (MB)	19.49	10.20	10.19	10.19	10.19
ParaRegex w/ QS (MB)	10.46	10.20	10.19	10.19	10.19

Processor [15] is used as the basic implementation of regular expression matching. Four rulesets picked from open source software Bro [16] and Snort [5] are tested (shown in Table I), while the network traffic from Darpa [17] is treated as the input data (presented in Table II).

A. Computational Overhead

We compare ParaRegex to general enumeration approaches [9], [10], [11] that enumerate all the states of the DFA during the matching process. Figure 7 shows the matching time of ParaRegex and the enumeration approach on different rulesets and traffic. By introducing the MSU structure, ParaRegex takes full advantage of the *states aggregation* property along with efficient bitwise operation. As a result, the processing speed is at least one to two orders of magnitudes faster than that of enumeration approach. Besides, the processing speed of enumeration approach falls sharply as the DFA states grow (say the DFA of bro50 has 667 states and the DFA of bro217 has 8094 states). In contrast, ParaRegex shows excellent scalability since the *active* states will *aggregate* to a small amount in most cases, no matter how large the DFA is.

B. Memory Consumption

Table III demonstrates the evaluation result of memory consumption of DFA, ParaRegex, and ParaRegex with the *Quick Start* optimization. Since the memory usage by ParaRegex is positively related to the number of *active* MSUs, the total consumption would decrease dynamically with the number of input characters. As shown in Table III, for the ruleset snort24, the memory usage is constantly 10.18 MB for traditional DFA method. For the basic ParaRegex implementation, the initial memory consumption is 19.49 MB, more than 90% larger than the DFA. After reading 2 characters, the usage of memory reduces to 10.20 MB, nearly the same as the size of DFA. Optimized by *Quick Start* ($k = 1$), the memory consumption of ParaRegex at the start stage is only 10.46 MB (index table included), only 2.7% larger than the DFA. Considering with the size the input traffic, the memory usage overhead of ParaRegex is negligible.

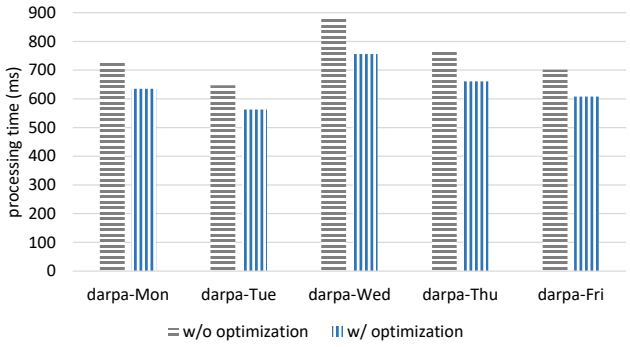


Figure 9. Evaluation on the *Smart Split* optimizations of ParaRegex

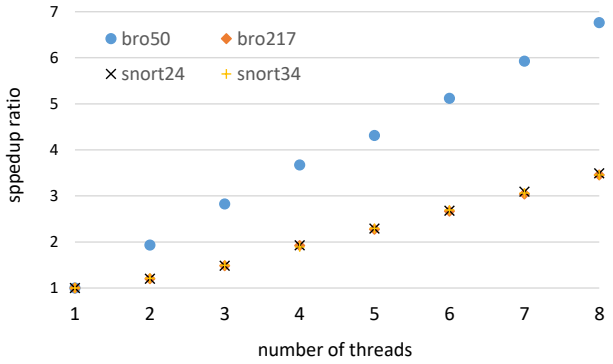


Figure 10. Evaluation of ParaRegex on D^2FA

C. Speed-up Ratio

To evaluate the performance of ParaRegex, we conduct a series of experiments using different numbers of threads, and treat the DFA matching method in Regular Expression Processor [15] as a baseline. Figure 8 shows the speedup ratios of ParaRegex on different network traffic and rulesets. As the number of working threads increases, the matching speed of ParaRegex grows and maximum speed is obtained when 8 threads process simultaneously in parallel.

In Fig. 8, the trends of speedup ratios stay nearly the same on different input traffic, which indicates the performance of ParaRegex to be robust on various input data. One noticeable difference in Fig. 8 is that the speedup ratio on bro50 is higher than other rulesets. We look deep into the *active* states that need to be traversed in each thread and find that the *active* states *aggregate* to only one rapidly when the ruleset is bro50, and in the cases of other rulesets the *active* states *aggregate* to two, which slows down the processing speed. It may be ascribed to the fact that the number of DFA states of bro50 is smaller than the others, so the *active* states number is more likely to *aggregate* to one for bro50. Another difference is that the speed-up ratio drops slightly when 5 or more threads are used. This may be due to the limitation of hyper-threading technology [18], where the sharing of cache and CPU resources influences the performance.

More experiments on other rulesets and traffic draw the similar conclusion that ParaRegex produces a fast matching engine with speeds of up to 4 times with 8 parallel threads compared to the original sequential implementation.

D. Data Partitioning Optimization

The *Smart Split* optimization method is proposed in Section III-B, aiming at gaining faster *aggregate* speed and decrease the computation complexity. Figure 9 shows the average processing time of ParaRegex using different numbers of threads with and without the data partitioning optimizations. Two groups of experiments are conducted on five Darpa traffic, and the $margin$ parameter is set to 20 in the evaluations. The experimental results suggest that about 15 percent of the processing time can be saved by introducing the *Smart Split* optimization.

E. Experiments on D^2FA

As mentioned before, ParaRegex is a framework that could efficiently parallelize existing approaches on regular expression matching. In this paper, we parallelize an improved D^2FA method [19] and Fig. 10 shows the power of ParaRegex. When 8 threads are used, the processing speed has increased by nearly 3 times on the rulesets of bro217, snort24 and snort34, and up to nearly 6.6 times on the ruleset of bro50. Since the improved D^2FA method accentuates the locality behavior of the DFA traversal operation, higher speedup ratio is obtained on ruleset bro50. However, the improvement of cache utilization fails to significantly affect the throughput of ParaRegex when the rulesets are large.

V. Related Work

Several techniques for parallel computation of regular expression matching have been proposed. Enumeration methods [9], [10], [11] enumerate all the states of the DFA and then associate each part's results. However, the overhead of this kind of methods is too high, making these methods hardly applicable for practical use. SFA (Simultaneous Finite Automaton) [20] extends an automaton so that it involves the simulation of transitions. SFA has a good property of parallelism, however, the constructing time of complex rules for SFA is unbearable. For the rule "([0-4]{500}[5-9]{500})*", it will take more than 1000 times longer to construct an SFA than DFA. Therefore, it is also unpractical for real-world use. PaREM [21] optimizes the possible initial states by excluding all the states that have no outgoing or incoming transitions for specified characters, but it suits for simple regular expressions and will become less efficient for large-scale rulesets.

Regular expression grouping methods [22], [23] divide the given regular expression into several groups to construct multiple automata, with the purpose to deflate the space consumption of DFA. Then each automaton is processed either in sequential or in parallel. Unlike these grouping based methods, ParaRegex splits the input data into a certain number of data blocks and traverses each block concurrently.

Hardware-based techniques [24], [25], [26] which use FPGA (Field-Programmable Gate Array) or TCAM (Ternary

Content Addressable Memory) have advantages in the parallel implementation using pipelines. However, the small size of on-chip memories limits the practical deployment of large-scale rulesets. Worse more, the high cost and big power consumption make FPGA and TCAM devices expensive for regular expression matching. With full utilization of the parallelism on a general-purpose platform, ParaRegex is more flexible and cost-efficient compared to these hardware-based methods.

VI. Conclusions and future work

In this paper, we present ParaRegex, a novel approach to achieve efficient parallelization of regular expression matching for Deep Inspection. Based on the *aggregation* phenomenon of DFA States in the matching process, ParaRegex uses MSUs to implement low-overhead and high-efficiency parallel matching engine. Moreover, two optimizations named *Smart Split* and *Quick Start* are proposed to further accelerate the matching speed and relieve the overhead. The experimental results on practical rulesets from Bro [16] and Snort [5] and real-world network traffic from Darpa [17] draw the conclusion that ParaRegex nearly obtains linear speedup ratios and up to 4 times speedup with 8 parallel threads compared to the original sequential implementation. In comparison with the enumeration approach, the processing speed of ParaRegex is at least one to two orders of magnitudes faster. We also apply ParaRegex to $\mathrm{D}^2\mathrm{FA}$ and gain more than 6 times speedup.

Our future work will focus on the following respects. First, the procedure of mapping, matching and reducing of ParaRegex suits distributed computing platforms like Hadoop [27] or Spark [28] naturally, so experiments on large-scale regular expression matching on these platforms are expected. Second, multiple *active* MSUs that would slow down the processing efficiency of ParaRegex can be parallelized by specific hardware. Third, the bit vector used in MSU could be compressed to further reduce the memory overhead. We hope all these platforms and algorithms can effectively work together to achieve efficient and high-performance regular expression matching in parallel for Deep Inspection.

References

[1] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[3] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[4] S. Wu, U. Manber *et al.*, "A fast algorithm for multi-pattern searching," 1994.

[5] "Snort." [Online]. Available: https://www.snort.org/

[6] R. Cox, "Regular expression matching can be simple and fast," Feb. 2007. [Online]. Available: https://swtch.com/ rsc/regexp/regexp1.html

[7] S. C. Kleene, "Representation of events in nerve nets and finite automata," DTIC Document, Tech. Rep., 1951.

[8] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, 2006, pp. 93–102.

[9] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," in *Implementation and Application of Automata*. Springer, 2009, pp. 54–64.

[10] 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*, vol. 42, no. 3, pp. 456–489, 2014.

[11] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] C. E. Mackenzie, *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[15] "Regular expression processor." [Online]. Available: http://regex.wustl.edu/

[16] "Bro." [Online]. Available: https://www.bro.org/

[17] "Darpa dataset." [Online]. Available: http://www.ll.mit.edu/ideval/data/1999data.html/

[18] "Intel hyper-threading technology." [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html/

[19] 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)*. ACM, 2007, pp. 145–154.

[20] R. Sinya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*. IEEE, 2013, pp. 220–229.

[21] S. Memeti and S. Pllana, "Parem: A novel approach for parallel regular expression matching," in *Proceedings of the 17th International Conference on Computational Science and Engineering (CSE)*. IEEE, 2014, pp. 690–697.

[22] Z. Fu, K. Wang, L. Cai, and J. Li, "Intelligent grouping algorithms for regular expressions in deep inspection," in *Proceedings of the 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2014, pp. 1–8.

[23] J. Rohrer, K. Atasu, J. van Lunteren, and C. Hagleitner, "Memory-efficient distribution of regular expressions for fast deep packet inspection," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES)*. ACM, 2009, pp. 147–154.

[24] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang, and V. Prasanna, "Feacan: Front-end acceleration for content-aware network processing," in *Proceedings of the IEEE INFOCOM*. IEEE, 2011, pp. 2114–2122.

[25] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2001, pp. 227–238.

[26] Y. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on fpga," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, 2008, pp. 30–39.

[27] "Apache hadoop." [Online]. Available: https://hadoop.apache.org/

[28] "Apache spark." [Online]. Available: http://spark.apache.org/