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}\x2Eekb” 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,
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 \( \{ Q, \Sigma, \delta, q_0, F \} \), where \( Q \) denotes a finite set of states, \( \Sigma \) denotes a finite set of input symbols, \( \delta \) 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 \( \delta \) only takes one state and returns a single state for an input symbol, while in an NFA \( \delta \) 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).

![Figure 1](image1.png)

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

Figure 2. Example of enumeration methods of regular expression matching

![Figure 2](image2.png)

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.
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.

Obviously, 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 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 \( \delta \). 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

---

### Algorithm 1: Enumeration Methods

**Input**: DFA \( D = (Q, \Sigma, \delta, q_0, F) \), Input Data \( C = c_1c_2...c_m, c_2...c_{m+1}, ..., c_{n-1}c_n, n \), number of threads \( n \)  
**Output**: \( q_{final} \) that \( q_0 \xrightarrow{C} q_{final} \)

```plaintext
1 // mapping and matching procedure
2 foreach \( i \in [1..n] \) parallel do
3     \( S_i \leftarrow Q \) // sub-result of block \( i \)
4     for \( j = 0 \rightarrow m_i \) do
5         foreach \( q \in S_i \) do
6             \( S_i[q] \leftarrow \delta(S_i[q], c_j) \)
7         end
8     end
9 end
10 // reducing procedure
11 \( q' \leftarrow q_0 \)
12 for \( i = 1 \rightarrow n - 1 \) do
13     \( q' \leftarrow S_i[q'] \)
14 end
15 \( q_{final} \leftarrow S_n[q'] \)
```

---

![Figure 3. aggregation phenomenon of DFA states on real-world traffic](image-url)

**Table I**

<table>
<thead>
<tr>
<th>Ruleset</th>
<th>bro50</th>
<th>bro217</th>
<th>snort24</th>
<th>snort34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of regexes</td>
<td>50</td>
<td>217</td>
<td>24</td>
<td>34</td>
</tr>
<tr>
<td>DFA states #</td>
<td>667</td>
<td>8094</td>
<td>8630</td>
<td>10212</td>
</tr>
<tr>
<td>after any 1 input</td>
<td>4.00</td>
<td>37.63</td>
<td>62.42</td>
<td>78.70</td>
</tr>
<tr>
<td>after any 2 input</td>
<td>1.02</td>
<td>5.17</td>
<td>19.19</td>
<td>31.69</td>
</tr>
</tbody>
</table>

---

Table I: AVERAGE NUMBER OF ACTIVE STATES FOR FOUR DIFFERENT RULESETS
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. $q$, s.t. $|\delta(q)| > 1$. This contradicts the fact that the transition function $\delta$ 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
- $\Sigma$: 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 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_1c_2...c_{1m_1}c_{2m_2}...c_{nm_n}, number of threads n
Output: q_{final}

// mapping and matching procedure;
for i \in [1...n] parallel do
    if i == 1 then
        M_i \leftarrow q_0
    else
        M_i \leftarrow M_0
    end
    for j = 1 \rightarrow m_i do
        foreach msu \in M_i do
            msu \leftarrow \delta(msu, c_{ij})
        end
        merge MSUs with the same msu.id
    end
end
// reducing procedure;
msu_{final} \leftarrow M_1
for i = 2 \rightarrow n do
    foreach msu \in M_i do
        if msu_{final}.id \&\& msu.mapping \neq 0 then
            msu_{final}.id \leftarrow msu.id
            break
        end
    end
end
q_{final} \leftarrow 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 $k$th 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 $\bigcup_{i=0}^{n-1} msu_i.mapping = 1$
state 0, thus the intersection of any two mapping vectors is 0. On the other hand, no DFA state can move to more than one state, the union of all mapping vectors is a bit vector where all bits are 1 in the mapping vector represents a unique DFA state, so the theorem can be derived directly Theorem 1.

Proof. Assume that \( \exists msu_i \in M_{k+1}, s.t. msu_{\text{final.id}} \land msu_i.\text{mapping} \neq 0, i.e., \forall msu_i \in M_{k+1}, msu_{\text{final.id}} \land msu_i.\text{mapping} = 0. \)

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 \( msu.\text{state} \) field can be transformed into another \(|Q|\)-bit vector, where the only \( msu.\text{state} \) bit is set to 1 and others are 0. At first, the only MSU in \( M_1 \) is denoted as \( msu_{\text{final}} \). From the second set, if the intersection between \( msu_{\text{final.id}} \) and one \( msu.\text{mapping} \) from the set of MSUs is not 0, then it means that starting from \( msu_{\text{final.id}} \), the DFA state would finally move to \( msu.\text{id} \) for this input data block. Therefore, the \( msu_{\text{final.id}} \) becomes \( msu.\text{id} \) 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 \( msu_{\text{final}} \) 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 \( \exists! msu \in M_{k+1} s.t. msu_{\text{final.id}} \land msu.\text{mapping} \neq 0. \)

Proof. Assume that \( \exists msu \in M_{k+1}, s.t. msu_{\text{final.id}} \land msu.\text{mapping} \neq 0, i.e., \forall msu \in M_{k+1}, msu_{\text{final.id}} \land msu.\text{mapping} = 0. \) Let \( m = msu.\text{state} \), because only \( m \)th bit in \( msu.\text{state} \) is 1 and others are 0, this means that \( \forall msu \in M_{k+1}, \) the \( m \)th bits are 0. As a result, \( \bigcup_{msu \in M_{k+1}} \neq 1, \) which contradicts Theorem 2. On the other side, assume that \( \exists \) two or more \( msu \in M_{k+1} s.t. msu_{\text{final.id}} \land msu.\text{mapping} \neq 0, i.e., \exists msu_i \text{ and msu}_j, 0 \leq i, j < n, \) the \( m \)th bits of \( msu_i.\text{mapping} \) and \( msu_j.\text{mapping} \) are 1, so \( msu_i.\text{mapping} \land msu_j.\text{mapping} \neq 0, \) 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...
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. Benefitting 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 D²FA [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 D²FA.

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 \(\text{pos}\) denotes the equally split position, \(\text{margin}\) be an adjustable parameter, then the character with the biggest AFs in the range \([\text{pos} - \text{margin}, \text{pos}]\) suggests a better split position.

Figure 6 shows a simple example of the Smart Split. The AF of character \(c_{m+1}\) is the largest among the characters from \(c_{m+2}\) to \(c_{n-j}\), so split the input data right before \(c_{m+1}\) 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
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 without and with Quick Start (QS) 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 8. Speedup ratios of ParaRegex on different network traffic and rulesets](image-url)
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 D²FA 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