International Journal of Security and Its Applications
Vol. 10, No. 4 (2016), pp.267-278
http://dx.doi.org/10.14257/ijsia.2016.10.4.25

A Parallel Algorithm of Multiple String Matching Based on Set-Partition in Multi-core Architecture

Jiahui Liu[1, 2], Fangzhou Li[1] and Guanglu Sun[1]

[1]College of Computer Science and Technology, Harbin University of Science and Technology, China
[2]School of Computer Science and Technology, Harbin Institute of Technology, China
bsuljh@163.com

Abstract

With the coming of the big data era, the data processing in large scale comes out with a new challenge. However, string matching still plays an important role in the network security and information retrieval fields, because of the large size of pattern set with the overhead of memory and access memory time. Improving the string matching algorithm to adapt to the large scale tasks is desirable and meaningful. In this paper, we present and implement a parallel algorithm of multiple string matching based on multi-core platform. In addition, this work focuses on the partition of pattern set by using genetic algorithm through the internal relation of the patterns to reduce the memory overhead and execution performance. Compared with the classical ones, our experiments on both high and low hit-rate data demonstrate that the performance of algorithm enhances about on average by 20%-40% in general. Besides, the proposed algorithm reduces the memory cost on average by 4%-20%.

Keywords*: Parallel, multiple string matching, multi-core, genetic algorithm*

1. Introduction

The application of big data has already begun to gradually permeate every aspect of our work, business and lives. Its impact also spreads to the field of the network security, information retrieval and so on. However, big data processing and analysis capabilities are far less than the ideal level. The rapid growth of data becomes a challenge to large data processing technology. String matching is facing an urgent requirement of improvement, applied the core technology of Network Intrusion Detection System (NIDS), Anti-virus System and other applications. However, some of previous studies focus on a small size of pattern sets, the emerging applications require larger scale efficient algorithms to process pattern matching problems [1].

The performance of the classical multiple string matching algorithms is decided by the three main factors as follows: the number of the pattern, the minimal length of the pattern and the size of the alphabet [2]. Moreover, the distribution of alphabet in the text would also affect the performance [3]. With the rapid increasing of the size of pattern set in the application based on multiple pattern string matching, the space occupancy problem of the main matching algorithm is becoming more and more prominent. Consequently, large memory access overhead leads to a dramatic decreasing in the performance of algorithms. The problem is to optimize the memory overhead and memory access time in the condition of keeping the access time of state transition table with the time complexity $O(N)$ and ensuring the random access.

Our work presents a novel parallel algorithm for multiple string matching that spends more time in preprocessing phase to reduce the overhead in matching phase. The proposed parallel algorithm is based on the set partitioning. We use a DFA-based AC algorithm [4] as the exemplification to implement the parallel algorithm and extract the pattern set from open source project NIDS Snort [5] to experiment.

The paper is organized as follows: background and related work are presented in Section 2. In Section 3, the detailed implementation of the algorithm is introduced. Experimental methodology and results are given in Section 4. Finally, the conclusions of this work are presented in Section 5.

2. Related Works

2.1. Aho-Corasick Algorithm and Optimization

Aho-Corasick (AC) Algorithm is a classical algorithm of multiple string matching invented by Alfred V. Aho and Margaret J. Corasick. There are two specific implementations of AC Algorithm: Deterministic Finite Automaton (DFA) and Non-deterministic Finite Automaton (NFA). Because of the deterministic linear time complexity $O(N)$ and fixed size of state transitions table, string matching based on the DFA model is desirable [6]. Although AC Algorithm has a time complexity of $O(N)$, where N is the sum of the matching text and all of the pattern set, it requires a large memory to store the state transition table, which is a limitation to memory space. The state transition table will be generated in preprocessing phase. Our work will optimize the state transition table in the preprocessing phase, and we design a DFA-based parallel algorithm.

A deterministic finite automaton A can be described as 5-tuple. It consists of Q, Σ, δ, q_0, F [7]:

- The Q is a finite set, its every element is called a state.
- The Σ is an alphabet, its every element is named an input symbol.
- The δ is a transition function in the form of δ: $Q \times \Sigma \rightarrow Q$.
- The q_0 ($q_0 \in Q$), included in the Q set, is a non empty set of initial states.
- The F ($F \subseteq Q$), a set of accept states, is a non empty set of final states included in the Q.

In optimization works, some solutions have been proposed in literature [8–10] to accelerate the execution speed in AC algorithm. Considering the scalability of speed, the number of patterns and the pattern length, the work [8] presents a speed hardware-implementable pattern matching algorithm for content filtering applications. The designed algorithm is based on Bloom Filter, which is a memory efficient multi-hashing data structure.

Report [9] proposes a memory-efficient multiple-character-approaching architecture called TDP-DFA, consisting of multiple parallel DFAs. TDP-DFA reduces the complexity obviously through efficient representations of the transition rules in each DFA. Hua *et al*. [10] introduce a string matching algorithm that achieves high throughput while limiting both memory-usage and memory-bandwidth by moving away from a byte-oriented processing of patterns to a block-oriented strategy.

Moreover, there are also some approaches in literature [11–14] to optimize memory. By observing, the size of state transition table could be reduced by partitioning the pattern set into multiple subsets, and subsets implemented into multiple AC-based DFAs [11, 13]. Tuck *et al*. [12] propose to save the memory overhead of non-existing transitions by utilizing bitmap compression and path compression on AC automaton. Tan *et al*. [14] present an approach to partition the AC-DFA into several AC-based DFAs to reduce the memory overhead through bit-split. Besides the conventional architectures, in order to overcome the performance neck-bottle AC algorithm is implemented on specialized hardware like GPU [15, 16], field-programming gate arrays (FPGA) [17–19], combination with CPU and GPU parallel platform [20] and so on.

2.2. Parallel DFA on Multi-core Architecture

Multi-core CPU is a single chip integrated with multiple CPU cores. Each CPU core is a separate processor. Each CPU core can possess its own separate cache, the cache of the same multiple CPU cores could be shared. By cores communicating directly via a shared memory, multi-core computers make space for acceleration exists [21].

In modern multi-core hardware architecture, memory for multiple CPU cores is shared. CPU core are generally symmetrical, and therefore belong to a shared memory multi-core symmetric multi-processor (SMP). In the multi-core hardware architecture, if wanted to give full play to the performance of the hardware must be multithreaded execution, so that each CPU core has a thread executing at the same time.

Single-core and multi-threading on a different, more threads are executed in parallel on multi-core physically, is a real sense of parallel execution, and at the same time there are multiple threads in parallel. Single-core and multithreaded on a staggered multi-threaded execution, in fact, only one thread is executing at the same time. Through utilizing the parallelized multiple string matching algorithms, multi-core architecture will be able to exhibit a better performance.

AC Algorithm is one of the commonly used multiple string matching algorithm which is appropriate to be parallelized [22]. According to the description above, each core with user memory space can be a capable container for DFA. Each DFA on the core also can work independently through data parallelism. The Figure 1 is a parallel DFA design based on the multi-core architecture.

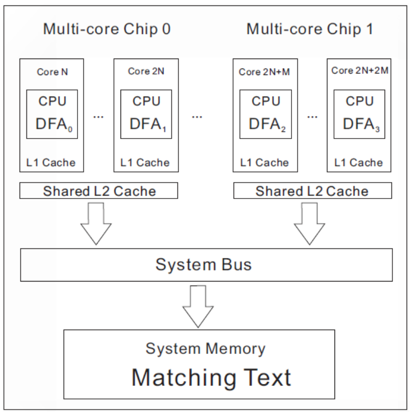

Figure 1. DFA on Multi-Core Architecture

2.3. Set Partitioning with Genetic Algorithm

Set Partitioning Problem (SPP) is a classical NP Hard Problem. The partition of the set is to divide the set into a number of subsets which cover all the elements of the set and are mutually exclusive. There are some solutions of heuristic algorithm to solve it, such as Genetic Algorithm (GA), Ant Colony Optimization (ACO) [23] and Particle Swarm Optimization (PSO) [24].

Genetic algorithm is a simulation of natural selection and biological evolution of evolutionary, which were originally developed by Holland (1975) [25]. Genetic algorithm is initialized by a population with possible solution, and a number of individual components consist of the population through gene encoding. Each individual is decided by a chromosome entity. Since the modelling work of the gene encoding is complicated, we tend to make it simple, such as binary code. The population of the generation after generation, according to the principle of survival of the fittest, has evolved more approximated solution.

In every generation, according to the individual problem domain fitness size to select individuals, and genetic operators from Nature Genetics help to crossover and mutate for next generation. It generates the new population represented the solution. This process will lead to that the previous generation is more propitious to the environment through the processing as nature evolution. The best individual after being decoded in the last generation of population can be used as a near-optimal solution of the problem.

Basic procedure of the genetic algorithm is described as following:

Step 1: Initialize the population and calculate the fitness of each individual in the population;
Step 2: Select the excepted individual as parents to operate for crossover operation from the population;
Step 3: Crossover between the parents selected in previous step, and mutate by probability;
Step 4: : Calculate the fitness of changed individual in the population;
Step 5: Update the population by replacing with the individual with better fitness and eliminating the worse ones, repeat Step 2-5 until the population reaches to the termination condition.

We improve the GA to adapt to the SPP in the parallel algorithm and take memory overhead expression as the fitness function. We also design the crossover and mutation operation to improve the performance, and propose the terminal function to get a feasible solution.

2.3.1. Initial Character: The work [26] classifies the pattern set by the initial character. Each category is used to make a DFA, it ignores the pattern set situation which initial characters of patterns is distributing sparsely.

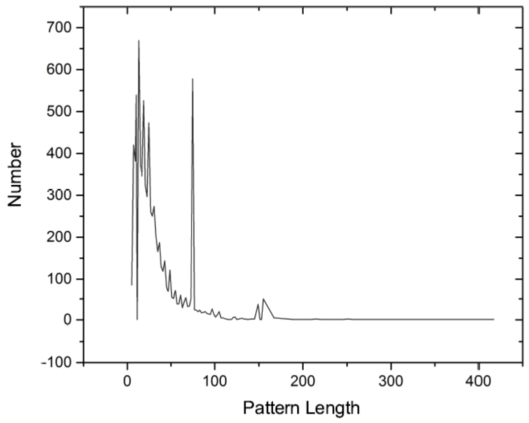

Figure 2. Distribution of length from Snort

2.3.2. Length and Length Pattern Grouping Metric: HyunJin Kim *et al.* [27] consider the pattern lengths, partitioning the target pattern set into two subsets respectively for short and long patterns. For the bit-split parallel pattern matching engine, it is required to optimize the memory overhead to suit the proper bit-split string matcher types for the two subsets.

The work presented in [28] focused on a method that partitioning the pattern set to makes each string matcher on each core has the average length which is close to the average length of total pattern by pattern grouping metric. It regards the average length of the total patterns length as the pattern grouping metric. The used states also average on each DFA. It maintains that the main influential factor of the throughput is the latency of state transitions. And the memory overhead is proportional to the number of states and the number of bits in each next state pointer. Therefore, it only takes consideration of the factor of length and number of states. Considering the situation with only few long patterns, we cannot allocate one separate core to process them.

2.3.3. Greed Algorithm and Dynamic Programming: The paper [29] maintains that the speed of string matching algorithm mainly depends on the number and minimal length of patterns. They proposed a heuristic algorithm using dynamic programming and the greedy algorithm techniques, to divide patterns set and choose an optimal string matching algorithm for them. It keeps balance of the running time on each core, to minimize total time.

Figure 2 plots the distribution of pattern set's length from Snort. It implies that its distribution is sparse. There is still possibility of improving performance. To solve the problems remaining above, we proposed a new heuristic based algorithm - Parallel algorithm of multiple string matching based on set-partition with GA. The internal relation of the patterns and the sparse distribution of the patterns' length are two more key factors we concerned.

3. Parallel Multiple String Matching Algorithm Based on Set-Partition

3.1. Preliminary Knowledge

A representation scheme should be considered first, the following two critical problems are closely related to the performance of the algorithm.

3.1.1. Optimal Patterns Set Decomposition Problem: The set partitioning problem in this work is a multi-objective optimization which also is the both key problem and NP-hard problem.

To describe the problem we use the symbol from the classical DFA definition as follows:

- P: state set, $|P|$ means the length of P, memory costs $|P|*log_2|p_i|$, stands for the longest length of p_i.
- Σ: character set, $|\Sigma|$ means the length of Σ.
- T_i: state transition table on $Core_i$, $|T_i|$ means the length of T_i.

State transition table gets a size equal to *maxiumstate***alphabetsize*, so $|T|=|P|*|\Sigma|*log_2|p_i|$ is the cost each time, when it accesses the memory of transition state table. Because of the complex relationship between P and Σ, $|\Sigma|$ and $|P|$ are hard to improve separately, but it can be cooperatively optimized to a suitable level in each core with a heuristic method.

We divide the set of pattern into subsets. Each subset is assigned to an independent core on the processors. The optimal pattern decomposition is a decisive factor to reduce the execution time of the parallel algorithm to the minimal possible degree.

A set is divided into n disjoint subsets $\{P_1, P_2, \ldots, P_n\}$ which are requirements of the elements of these subsets $\{p_1, p_2, \ldots, p_n\}$ in the greatest possible small, the goal function of this task:

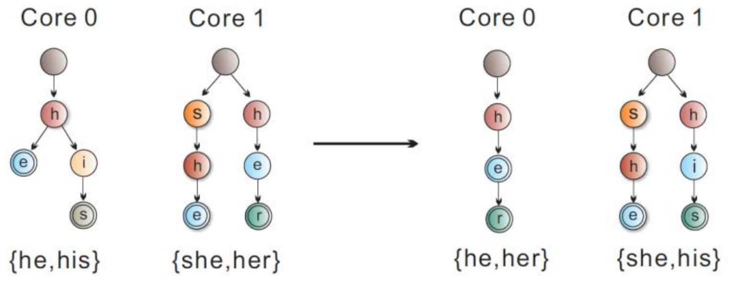

Figure 3. Example of a Pattern Set Aggregate

$$G(P)=Min(Max(|P_i|*|\Sigma|*log_2|p_i|)),\ p_i \in P \tag{1}$$

subject to :

$$\sum_{i=1}^{n} p_{ij} = 1 \tag{2}$$

where p_{ij} denotes *i*-th pattern assigned to subset P_j, that means each pattern only can be assigned to only one subset P_i.

$$P_i \cap P_j = \varnothing, \qquad i, j \in [1, N] \tag{3}$$

Any two pattern subset P_i and P_j have no common part.

3.1.2. Optimization Memory Cost with Recombination: The internal relation of the patterns on the partition-based DFA has more influence to the performance. For example, from Figure 3 we illustrate a pattern set :{ her, his, she, her}. Now we assign them to two cores, aiming to the comparison between the two allocating scheme. The front one costs 10 units size of memory, but the back one costs 9 units size of memory. Memory cost is also closely related to the time complexity according to the analysis of Section 3.1.1.

It is obvious that the combination between pattern set are the mainly factor to determine the performance of multiple string matching algorithm in data parallelism multi-core architecture. The distribution of the length of pattern set effects the performance of algorithm after the partitioning of pattern set, so we develop a heuristic algorithm to take advantage of considering the character we discussed above.

3.2. The Implementation of Parallel Algorithm

The algorithm includes three mainly phases as following:

- Pre-process phase: decomposing the pattern set into *n* subsets to prepare for the next phase;
- Parallel processing phase: building *n* DFAs on *n* cores, each core can play a role of multiple string matcher;
- Matching phase: parallel string matching on each DFA and reduce the result from all of the cores.

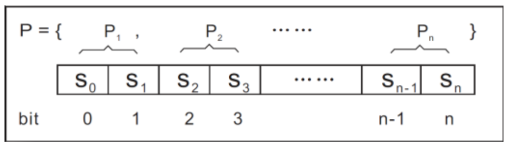

Figure 4. Representation of Solution

3.2.1. Decomposition Pattern Set with GA: We design a representation of the chromosome to generate solution which also subject to the limitation. N_{Core} and N_{Pat} stand for the number of cores and the number of patterns respectively. The length of solution *n* is computed from $log_2N_{Core}*N_{Pat}$. p_i is represented with log_2N_{Core} bits to locate which subset it will be put in.

For example, as Figure 4, it uses 2 bits per pattern because of the number of core is 4. p_2's expression of status s_2s_3 is binary $(10)_2$ which means p_2 will be put in the 2nd subset.

The binary code represents as $(s_0s_1…s_{n-1}s_n)_2$ using *l* to denote log_2N_{Core}, and the calculation expression is as follows:

$$p_i = \sum_{j=i\times l}^{(i+1)\times l} s_i \times 2^{(i+1)\times l - j} \tag{4}$$

Initial Population and Parent Selection

As Algorithm 1 describes, the initial population Pop_0 is built by a randomly approach. The L_{chrom} stands for length of chromosome and $|Pop|$ stands for the size of population. Set Pop_0 is initializing to be an empty set, for each individual in the Pop_0, it will randomly generate a chromosome sequence whose element has a limitation on the range of value from 1 to L_{chrom}. Each new created individual will join set Pop_0 until all of the individuals have been generated.

Algorithm 1 is called as the initializing original population P_0.

```
1: Set $Pop_0 = \varphi$
2: for each $i \in [1,|Pop|]$ do
3:     Set Individual$_i$ = new Individual
4:     for each $j \in [1, L_{chrom}]$ do
5:         randomly select a integer $r$, $r \in [0, N_{Core}-1]$
6:         Individual$_{i\,j}$ <= $r$
7:     end for
8: $Pop_0 <= Pop_0 \cup$ Individual$_j$;
9: end for
```

Crossover and Mutation Operation

Simply, we regard the top two individuals in the sorted population by fitness as the parents. In Figure 5, after reserving the part $A \cap B$, $A\text{-}A \cap B$ and $B\text{-}A \cap B$ have the same length, which will be divided into two parts (C and D) by the random position r. Set C from *start* to r in $A\text{-}A \cap B$ and set D from r to *end* in $B\text{-}A \cap B$.

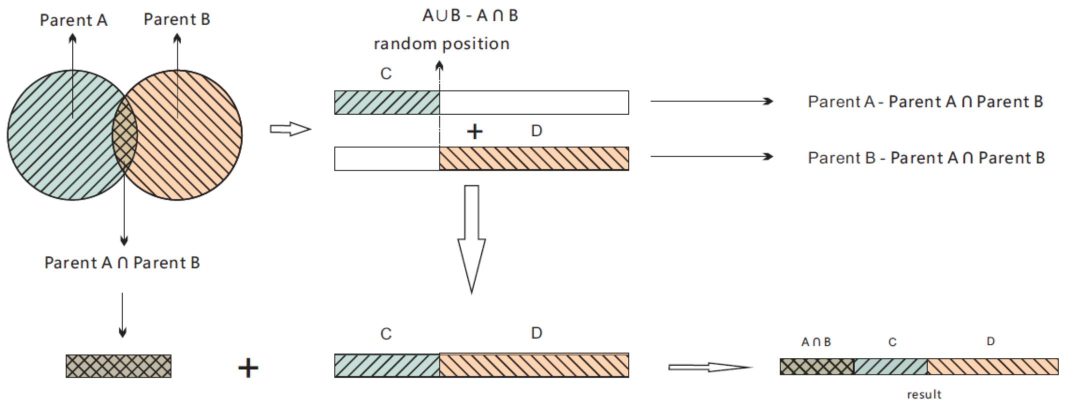

Figure 5. Crossover Procedure

Mutation operation is similar to the operation of initializing population, but it randomly change the chromosome according to the probability of the mutation as Algorithm 2 describes.

Termination Function

By utilizing the Pearson Product-moment Correlation Coefficient (PMCC) to estimate the two nearly fitness sequence, we can describe the degree of linear correlation between two variables. The Pearson correlation coefficient between the two variables is defined as the quotient of the covariance and standard deviation between the two variables. As the following equation:

$$\rho_{x,y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{5}$$

Algorithm 2 is called “Individual Mutation”.

```
1: for each $i \in [1, L_{chrom}]$ do
2:    generate a float number $p$, $p \in [0,1]$
3:    if $p$<mutation probability then
4:        randomly select a integer $r$, $r \in [0, N_{Core}-1]$
5:        Individual$_{ij}$ <= $r$
6:    end if
7: end for
```

As we observe, the distribution of ρ is subject to normal distribution, we use T test from statistics to judge the criterion of termination here. Once the T value is determined, we can learn whether the solutions of populations tend to be stable from result with the *p*-value. The T test equation is described as following:

$$t = \frac{\bar{\rho} - \mu_0}{S / \sqrt{N}} \tag{6}$$

where $\bar{\rho}$ is the sample mean from a sample { $\rho_{x_{n-N}, x_{n-N+1}}, \rho_{x_{n-N+1}, x_{n-N+2}}, \cdots, \rho_{x_{n-2}, x_{n-1}}$ } of size *N*. *S* is the ratio of sample standard deviation over population standard deviation. σ is the population standard deviation of the data, and μ is the population mean, denoting the newest solution ρ_{x_{n-1}, x_n}.

The null hypothesis H_0:μ=μ_0, the solutions of populations tends to be stable, and the alternative hypothesis, H_1:$\mu \neq \mu_0$. If the null hypothesis is accepted, the result comes out to be that the solutions of populations tends to be stable.

3.2.2. Construct DFA on Multi-core Architecture: DFAs, which are built with pattern subsets, will be assigned to each core to process in parallel. The pattern subsets after being partitioned are to be built the automation. We divide the pattern set according to the number of core, and each core is responsible for running a DFA and stores it in the RAM memory. As the Algorithm 3 describes, after allocating the DFA space according to the size of pattern set in memory block, we append patterns to the corresponding DFA. Matching up with existing states firstly, then we add new state for the rest of the pattern bytes. At last, we build DFA through generating state transition table.

Algorithm 3 is named as “DFAs on Multi-core”. It is described as following.

```
//step 1.
allocate DFA space in memory
//step 2.
for each core on each processor do
    if not match up with existing states than
        add new state for the rest of the pattern
    end if
    Build automation $A_i$ with $P_i$,
end for
```

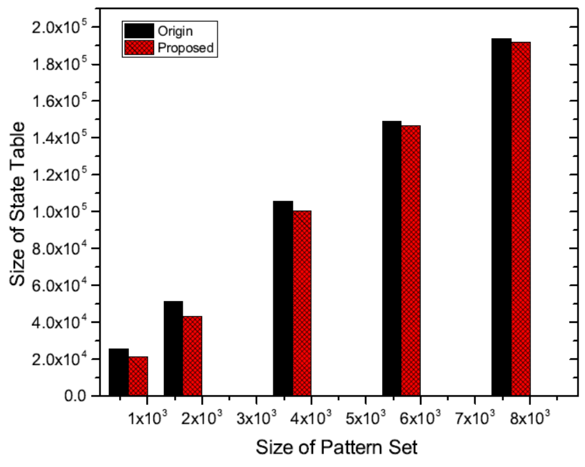

Figure 6. Comparison of the Size of State Table

3.2.3. Parallel Matching Algorithm: In matching phase, we try to achieve a goal that all of the computing resources are fully utilized. Before matching text, as the Algorithm 4 illustrates, we load the matching text into shared memory from the disk. Every DFA on core reads and processes the matching text from the shared memory in runtime. Once one of the DFA finishes processing, partial result of matching stores in each thread to record the count. When all the threads are finished, the master node core reduces all the result generated by DFAs from other cores, since the union of all is the complete set.

Algorithm 4 is called as "Text Matching Algorithm".

```
1: for each $DFA_i$ on each core do
2:     while $DFA_i$ not finish reading Text T from shared memory. do
3:         $r_{Pi} \leftarrow DFA_i$ search T, search Text T with $DFA_i$ built by $P_i$
4:     end while
5: end for
6: $r_P \leftarrow r_{P1} \cup r_{P2} \cup \ldots \cup r_{Pn}$, reduce all the result $r_{Pi}$ from DFAs
```

4. Experiments and Results

In this section, we make experiment to evaluate the execution time of matching and memory overhead of the proposed algorithm. The algorithm is implemented in C programming language and tested on a computer which is installed operating system with Linux kernel version 4.0 and equipped with an Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz, 3301Mhz with 4 cores and 8GB RAM memory. The pattern set is extracted from the Snort rule library (version 2.9).

4.1. Analysis of the Memory Overhead

Figure 6 illustrates that in different scale of pattern set, the size of state transition table optimized by proposed algorithm is smaller than origin one. With the increasing of the size of pattern set, the memory overhead, which is spent by state transition table, gradually gets a reduction from about 4%-20%.

The case used in this experiment applies a 4-core environment, we randomly select a pattern set from Snort with variable lengths as follows: {1000, 2000, 4000, 6000, 7985}. It reaches the highest value of memory optimization pattern set in size 2000 compared to origin situation where the memory reduced by about 20%, and in the case where the pattern set size reaches 7985 with memory optimization, it gets almost 4% memory reduction compared to the origin one.

It is easy to see the pattern set in all of the cases in length always has effect. It indicates that proposed algorithm is effective to memory usage optimization. Since the different distribution of features to pattern set, the effect of the pattern set optimization will receive the impact. When all patterns from the pattern set are the same, the optimization of the algorithm has no effect. However, because each feature of pattern in sets is unique, there is no two completely same pattern in the pattern set. As mentioned above, the proposed partition-based parallel algorithm has considerable effect in general.

4.2. Change of Variable Core Number

Figure 7 shows that in variable number of core, the execution time has different growth trend. It means that the proposed parallel algorithm could be more efficient as the number of core increases.

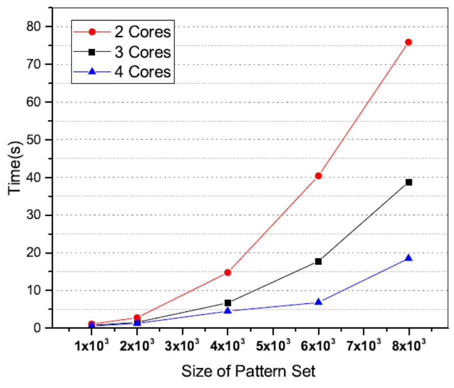

Figure 7. Comparison of Execution Time between Different Numbers of Core

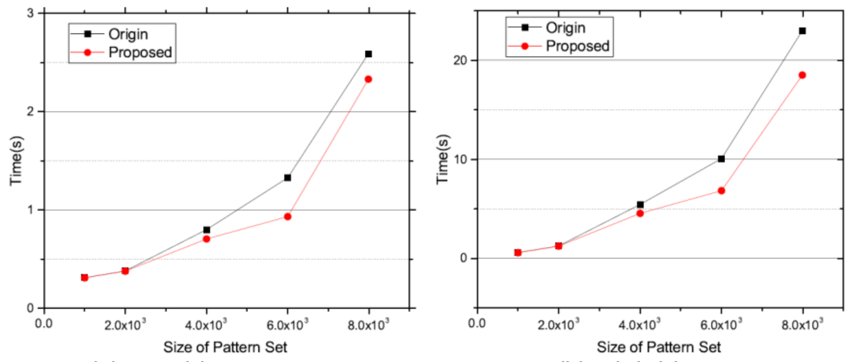

Figure 8. Analysis of Execution Time with Hit Rate

In this case we randomly generated an around to 90% hit rate matching text mixed from pattern set and the Bible text in a 100 MB size. From Figure 7, we can see that in the same pattern set, the larger the size of pattern set is, the more obvious the difference of execution time between different numbers of cores will get. For example, when the size of pattern set is set to 4000, in range of core number from 2 to 4, the ratio of escape time is 5:3:2. However, when the size is set to 7985, the ratio is 6:3:1, we can clearly see the larger pattern set size is, optimization results are in better execution time.

The larger size pattern set has, available space of combinations for allocating gets greater, so does optimized space. When meeting the same number of core, with the pattern set size growth, the execution time of matching also follows the growth. However, it may be drawn from Figure 7, the greater the number of cores is, the more steady growth trend will be, on the contrary, it has the trend of increasingly steep.

Due to the increasing number of cores, subsets for partitioning getting more as well as the space for combination getting larger, resulting in that the optimization of effect is better. With the size of pattern set enlarging, the 2-core case increases apparently, while the growth of other two cases is slightly. Compared with 2 cores, the proportion 4 cores to 2 cores of matching time can be up to six times. We can also learn that with the scale of pattern set increasing, the speed-up rise rapidly.

4.3. Analysis with High and Low Hit Rate

Figure 8(a) and 8(b) demonstrates the execution time between origin parallel AC algorithm and the proposed algorithm with the low hit rate and the low hit rate matching text respectively. We randomly generated matching text by the hit rate 10% and 90% from pattern set and the Bible text, the size of which is 100 MB, and using the same set of pattern.

The experimental results show that the range of the execution time in 10% hit rate is 0.3 second to 2.7 seconds, and in 90% gets 0.3 second to 26 seconds. The hit rate has a significant impact on matching speed of the DFA. We can also see that the proposed algorithm has more or less effect in optimization of execution time. The proportion of execution time optimization with high hit rate is also very similar with low hit rate.

With the increment of pattern set size, the optimization effect of execution time steadily has no proportional increment, but the reduce of execution time reaches a peak point when the size increase to 6×10^3. The reason is that in the case of a low hit rate, the result led by the partitioning optimization scheme in 6×10^3 size and the characteristics of matching text.

In Figure 8(a), with the size of pattern set ascends, the execution time of the proposed algorithm is reduced on average by approximate 1.0%-42.4%, and the reduce of execution time reaches a peak point when the size increase to 6×10^3.

5. Conclusions

In this paper we propose a parallel string matching algorithm, mainly by decomposing the pattern set with GA on multi-core architecture to reduce the memory cost, and improve the performance. The processing of the irregular distribution of the characteristic such as average length and initial character of the pattern set makes the algorithm more adaptive.

The proposed algorithm is tested in environment with the variable number of core and different sizes of pattern set. It gives an example to develop an efficient parallel algorithm through a heuristic set partitioning method. As a result, the enhanced performance of the proposed algorithm can be useful for the IDS[30-31], Anti-virus System， and so on.

Acknowledgements

This work is supported by Natural Science Foundation of Heilongjiang Province of China under Grant No. F201304.

References

[1] O. Villa, D. P. Scarpazza, and F. Petrini, “Accelerating real-time string searching with multicore processors,” Computer, vol. 41, no. 4, pp. 42–50, 2008.
[2] G. Navarro and M. Raffinot, Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences. Cambridge University Press, 2002.
[3] P. Liu, Y. Liu, and J. Tan, “A partition-based efficient algorithm for large scale multiple-strings matching,” Springer Berlin Heidelberg, vol. 3772, pp. 399–404, 2005.
[4] 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.
[5] “Snort, network intrusion detection system. available:, http://www.snort.org.”

[6] P. C. Lin, Y. D. Lin, Y. C. Lai, and T. H. Lee, “Using string matching for deep packet inspection,” Computer, vol. 41, no. 4, pp. 23–28, 2008.
[7] Y. E. Yang, V. K. Prasanna, and C. Jiang, “Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance,” in Parallel and Distributed Processing Symposium, International, 2010, pp. 1–11.
[8] S. Dharmapurikar and J. W. Lockwood, “Fast and scalable pattern matching for network intrusion detection systems,” IEEE Journal on Selected Areas in Communications, vol. 24, no. 10, pp. 1781 – 1792, 2006.
[9] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, “A memory-efficient parallel string matching architecture for high-speed intrusion detection,” Selected Areas in Communications IEEE Journal on, vol. 24, no. 10, pp. 1793–1804, 2006.
[10] N. Hua, H. Song, and T. V. Lakshman, “Variable-stride multi-pattern matching for scalable deep packet inspection,” in INFOCOM 2009, IEEE, 2009, pp. 415 – 423.
[11] J. V. Lunteren, “High-performance pattern-matching for intrusion detection.” Proceedings - IEEE INFOCOM, pp. 1 – 13, 2006.
[12] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” in INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, 2004, pp. 333–340.
[13] T. Song, W. Zhang, D. Wang, and Y. Xue, “A memory efficient multiple pattern matching architecture for network security,” in INFOCOM 2008. The 27th Conference on Computer Communications. IEEE, 2008, pp. 166–170.
[14] L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” in Computer Architecture, 2005. ISCA ’05. Proceedings. 32nd International Symposium on, 2005, pp. 112–122.
[15] G. Vasiliadis, S. Antonatos, M. Polychronakis, and M. S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors,” Lecture Notes in Computer Science, vol. 5230, pp. 116–134, 2008.
[16] D. Man, K. Nakano, and Y. Ito, “The approximate string matching on the hierarchical memory machine, with performance evaluation,” 7th International Symposium on Embedded Multicore Socs, (2013).
[17] Y. E. Yang and V. K. Prasanna, “Memory-efficient pipelined architecture for large-scale string matching,” in Field Programmable Custom Computing Machines, 2009. FCCM ’09. 17th IEEE Symposium on, 2009, pp. 104 – 111.
[18] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using fpgas,” in Field-Programmable Custom Computing Machines, 2001. FCCM,039,01. The 9th Annual IEEE Symposium on, 2001, pp. 227–238.
[19] H. J. Jung, Z. K. Baker, and V. K. Prasanna, “Performance of fpga implementation of bit-split architecture for intrusion detection systems,” in Parallel and Distributed Processing Symposium, International, 2006, p.177.
[20] C. S. Kouzinopoulos, J.-A. M. Assael, T. K. Pyrgiotis, and K. G. Margaritis, “A hybrid parallel implementation of the ahocorasick and wumanber algorithms using nvidia cuda and mpi evaluated on a biological sequence database,” International Journal on Artificial Intelligence Tools, vol. 24, no. 01, p. 1540001, 2015.
[21] M. Herlihy and N. Shavit, “The art of multiprocessor programming,” p. 22 23, 2008.
[22] S. Arudchutha, T. Nishanthy, and R. G. Ragel, “String matching with multicore cpus: Performing better with the aho-corasick algorithm,” in Industrial and Information Systems (ICIIS), 2013 8th IEEE International Conference on, 2013, pp. 231–236.
[23] A. Colorni, “Dorigo m, maniezzo v. distributed optimization by ant colonies,” Proceedings of First European Conference on Artificial Life, pp. 134 – 142, 1991.
[24] I. C. Trelea, “The particle swarm optimization algorithm: convergence analysis and parameter selection,” Information Processing Letters, vol. 85, no. 6, pp. 317–325, 2003.
[25] P. C. Chu and J. E. Beasley, “Constraint handling in genetic algorithms: The set partitioning problem,” Journal of Heuristics, vol. 4, no. 4, pp. 323–357, 1998.
[26] Y. Fan, H. Zhang, J. Liu, and D. Xu, “An efficient parallel string matching algorithm based on dfa,” Communications in Computer & Information Science, pp. 349 – 356, 2013.
[27] H. Kim, H. Hong, D. Baek, J. Ahn, and S. Kang, “A memory-efficient heterogeneous parallel pattern matching scheme in deep packet inspection,” vol. 7, no. 18, pp. 377–382, 2010.
[28] H. J. Kim and S. Kang, “A pattern group partitioning for parallel string matching using a pattern grouping metric,” Communications Letters IEEE, vol. 14, no. 9, pp. 878 – 880, 2010.
[29] G.-M. Tan, P. Liu, D. Bu, and Y. Liu, “Revisiting multiple pattern matching algorithms for multi-core architecture,” Journal of Computer Science and Technology, vol. 26, no. 5, pp. 866–874, 2011.
[30] Li G U, Fei L I, Qiao P L. Research and Implementation of Network Intrusion Prevention System under the Linux Platform. Journal of Harbin University of Science & Technology, vol. 14, no.2, pp. 8-12, 2009.
[31] Xie Y N, Liu S H. Research and Realization of Intrusion Detection System’s Rule Base Based on CVE Characters. Journal of Harbin University of Science & Technology, vol. 10, no.2, pp. 23-25, 2005.