Accelerating String Matching Algorithms on Multicore Processors

Cheng-Hung Lin

Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan

***Abstract*—** *String matching is the most computation-intensive process in many applications, such as network intrusion detection system, web searching and biological matching. The Aho-Corasick algorithm is the most popular string matching algorithm because of its ability to use one thread to match all patterns in parallel. In our previous work, we propose a string matching algorithm called Parallel Failureless Aho-Corasick algorithm to parallelize the traditional Aho-Corasick algorithm by adopting multiple threads on graphic processing units. Due to the advancing technology of multi-core processors, in this paper, we accelerate the Parallel Failureless Aho-Corasick algorithm on multi-core processors using multi-threaded implementation. Experimental results show that for processing large scale of inputs and patterns, the Parallel Failureless Aho-Corasick algorithm performing on multi-core processors delivers throughput up to 33 Gbps, 4 times faster than the traditional multi-threaded Aho-Corasick algorithm. Both the performance and scalability of the Parallel Failureless Aho-Corasick algorithm is improved on multi-core processors.*

***Keywords*— string matching, parallel processing, Aho-Corasick, multi-core.**

I. INTRODUCTION

String matching is the most computation-intensive process in many applications, such as network intrusion detection system, web searching and biological matching. For example, in signature-based network intrusion detection systems (NIDS), string matching is used to inspect packet payloads against thousands of attack patterns for finding malicious packets. To accommodate the ever-increasing attack patterns and satisfy the high-speed network communication, accelerating string matching has become a critical issue.

Due to the development of multi-core central processing units (CPUs) and graphic processing units (GPUs), many researches [2-9] are proposed to parallelize string matching algorithms on these multi-core machines. Among these string matching algorithms, the Aho-Corasick [1] algorithm is widely adopted because of its ability of searching multiple patterns simultaneously. In our previous works [2][3], we modify the AC algorithm and propose a parallel algorithm called Parallel Failureless Aho-Corasick (PFAC) algorithm to accelerate string matching on graphic processing units (GPUs). We have also released the open source library called PFAC [11] on Google Code. The PFAC algorithm performing on NVIDIA GPUs achieves remarkable performance improvement over the AC algorithm. However, because string matching is a memory-intensive application, the overhead of communication between CPU and GPU has significant impact on the performance of the PFAC algorithm performing on GPUs. In addition, the memory capacity of GPUs is another critical issue to process big data.

The advancing technology of transistor integration is producing increasing powerful multi-core processors. For example, the Intel® Xeon® E7-8870 [15] processor contains up to ten cores per processor which provides up to twenty threads per processor with Intel® hyper-threading technology. The Xeon E7 processors feature up to 30MB L3 cache which can be dynamically shared by all cores. The memory capacity supports up to 4,096 GB. In addition, Amazon Elastic Compute Cloud (Amazon EC2) [14] provides high memory cluster instance which features up to 244GB memory on Xeon processors. Without the overhead of PCIe communication, we would like to evaluate the performance of the PFAC algorithm performing on these multi-core processors.

In this paper, we accelerate the PFAC algorithm on multi-core processors using OpenMP [12] library and evaluate the performance and scalability of the PFAC algorithm on multi-core processors. The results are compared with a multi-threaded Aho-Corasick algorithm. For processing 16 GB inputs with 10K patterns, the PFAC algorithm achieves up to 33 Gbps throughput on the Intel® Xeon® processors, which delivers 4 times of performance improvement over the multi-threaded AC algorithm.

II. REVIEW OF PARALLEL FAILURELESS AHO-CORASICK ALGORITHM

Before we review the Parallel Failureless Aho-Corasick algorithm, we first introduce a data parallel approach adopted by many researches [4][5][6][7][8][9] to parallelize the Aho-Corasick algorithm, referred as the Data Parallel Aho-Corasick (DPAC) approach. The DPAC approach first uses the traditional AC algorithm to compile string patterns into a finite state machine of which the state transition table is called the AC state transition table. Then, the DPAC approach divides input string into multiple segments and assigns each segment an individual thread to perform string matching by traversing the AC state transition table. The DPAC has a well-known boundary detection problem that the pattern located across the boundary cannot be found. To resolve the boundary detection problem, each thread of DPAC must scan across the boundary for an additional length which is equal to the longest pattern length minus one. In other words, each thread of DPAC has constant duration time, O(s+m-1) where s is the segment size and m is the longest pattern length. Fig. 1 shows the AC state machine for matching the three patterns, "AABA", "ABA", and "BAB". In Fig. 1, valid lines represent valid transitions for specific input characters while the dotted lines represent failure transitions which are taken when no valid transition exists for an input character. Fig. 2 shows the DPAC approach where the input string is divided into multiple segments and each segment is assigned a thread to traverse the AC state machine. Because the length of the longest pattern, "AABA" is four, each thread has to scan across the boundary for three characters to resolve the boundary detection problem. In the example, the thread #3 will find the pattern "AABA" occurring in the boundary of segment #3 and #4. In Section IV, we will evaluate the relationship between the performance of DPAC, thread number and scheduling to realize how to achieve the maximum performance on multi-core processors.

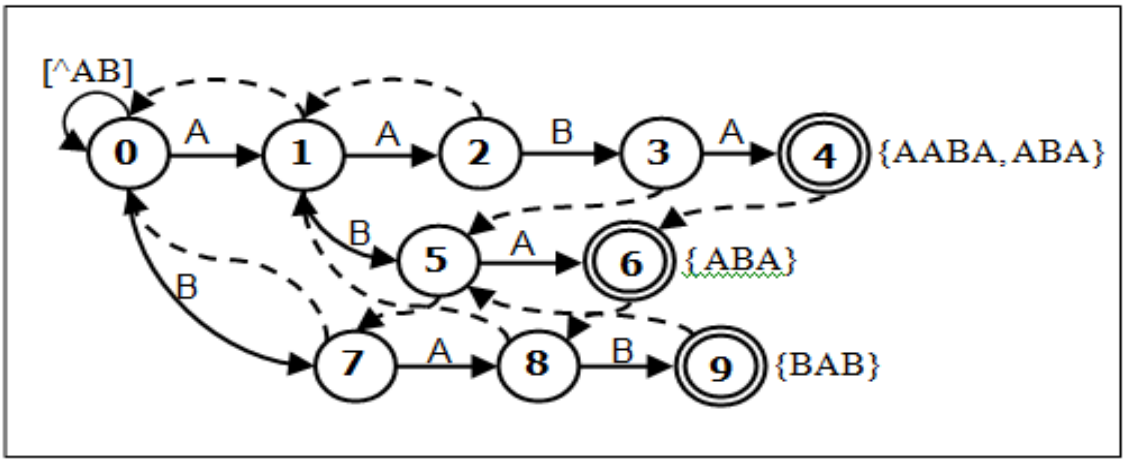


FIGURE 1. AC STATE MACHINE OF THE PATTERNS "AABA", "ABA", AND "BAB"

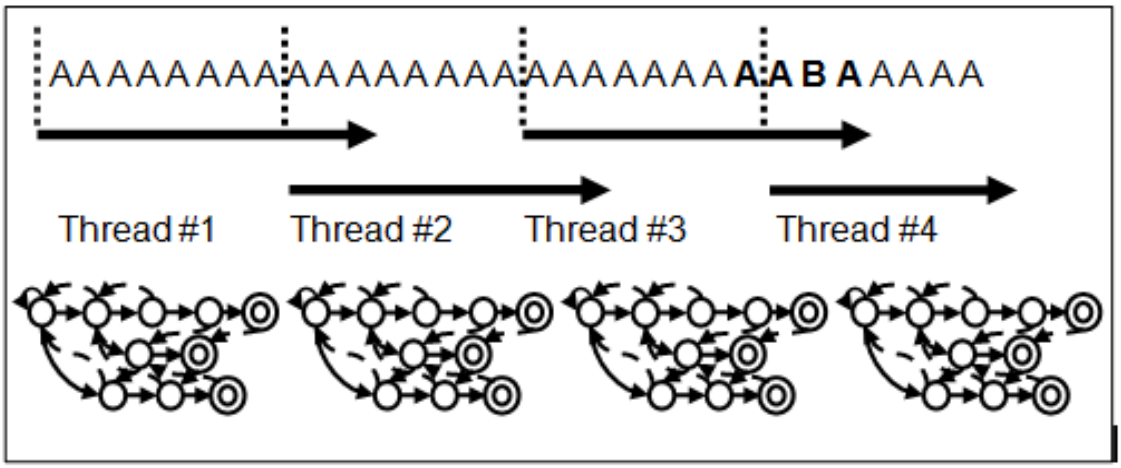


FIGURE 2. EACH THREAD HAS CONSTANT DURATION TIME EQUAL TO THE SEGMENT SIZE PLUS THE LONGEST PATTERN LENGTH MINUS ONE.

To improve the efficiency of DPAC, the PFAC [2][3] algorithm is proposed to accelerate string matching using multiple threads on GPUs. Different from the DPAC, the PFAC assigns each input byte an individual thread as shown in Fig. 3. Each thread in PFAC is only responsible of finding patterns from its starting position. Therefore, all failure transitions can be removed and only valid transitions exists as shown in Fig. 4. The compact state machine is referred as Failureless-AC state machine. Whenever the thread cannot find a valid transition for an input character, the thread terminates immediately without taking failure transitions. For example, Fig. 3 shows the status of each thread where the states activated are marked as red color. In Fig. 3, threads #5 and #6 reach the final states and find the pattern "AABA" and "ABA", respectively. Except for the threads #5 and #6, the other threads terminate in the early stages. Different from the DPAC algorithm, the duration time of threads is variant from O(1) to O(m), where m is the longest pattern length. Table 1 summarizes the time complexity of a thread in the AC, DPAC, and PFAC, where n, s, and m represent the input length, segment size and the pattern length, respectively. Compared to the AC and DPAC algorithms, the PFAC theoretically has the best time complexity.

Furthermore, each final state in the PFAC state machine only represents a unique pattern. For example in Fig. 4, the state 4 only represents the final state of pattern "AABA" while in Fig. 1, state 4 represents the final states of "AABA" and "ABA". Based on the property, we can use state encoding to represent final states. Then, the output table can be removed. Because the output table is eliminated, the performance of PFAC can be further improved.

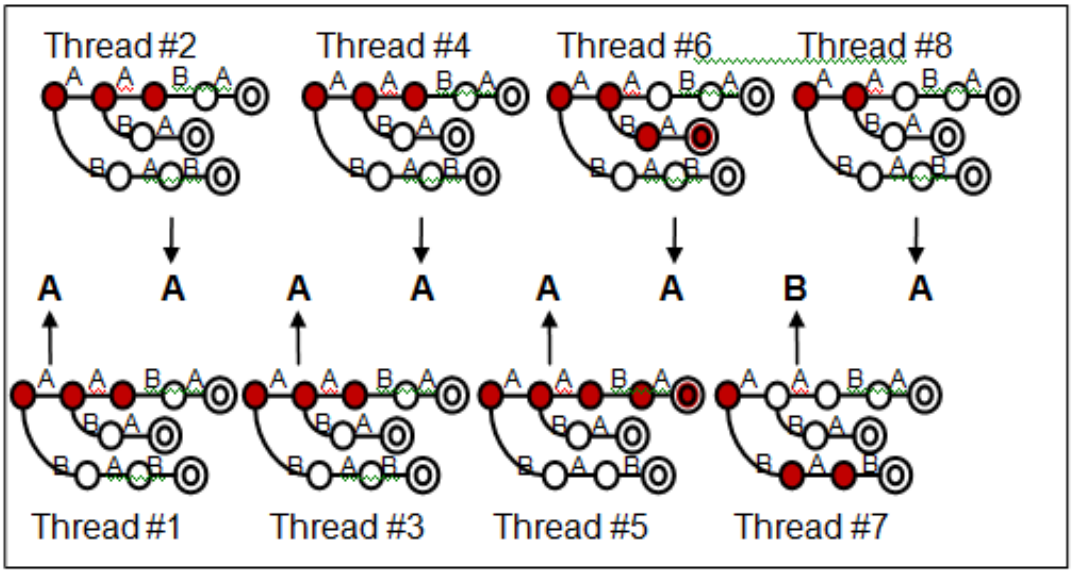


FIGURE 3. PARALLEL FAILURELESS-AC ALGORITHM

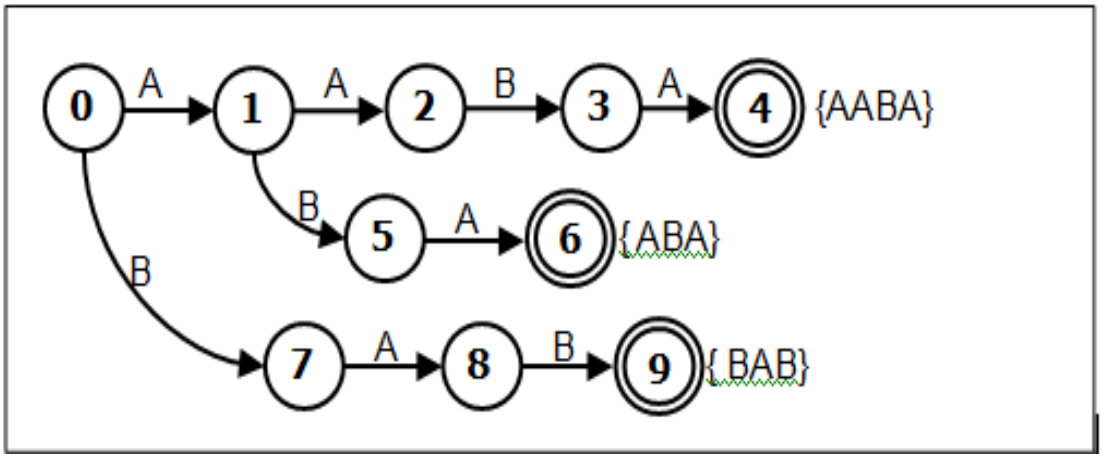


FIGURE 4. FAILURELESS-AC STATE MACHINE OF THE PATTERNS "AABA", "ABA", AND "BAB"

The PFAC is very adaptive to be implemented on GPUs because GPUs can issue huge amount of threads by their hundreds or thousands of physical cores simultaneously. However, the overhead of communication between CPU and GPU limits the total system throughput. In addition, memory capacity of GPUs is another critical issue for processing large scale of data. Because the advance of CPU technology, the latest CPUs, such as Intel® Xeon® E7-8870 processor features up to ten cores per processor which allow twenty threads per processor with Intel® hyper-threading technology. In addition, the new Xeon processors have large memory capacity including 30MB L3 cache and 4,096 GB main memory. With the increasing number of cores per processor and high memory capacity, performing PFAC on multi-core processors can deliver significant performance for processing large scale of data.

TABLE 1
COMPARISON OF MATCHING TIME COMPLEXITY

Algorithm	Time Complexity of a thread
Aho-Corasick (AC)	O(n)
Data Parallel Aho-Corasick (DPAC)	O($s + m$)
Parallel Failureless Aho-Corasick (PFAC)	O(1) to O(m)

III. PARALLELIZATION ON MULTI-CORE PROCESSORS

In this paper, we evaluate the performance of the PFAC algorithm and the DPAC algorithm on multi-core CPUs using multi-threaded implementations. Specifically, we put emphasis on the scalability of the PFAC algorithm to accommodate large scale of inputs and patterns. Finally, we evaluate the performance of two types of thread scheduling. OpenMP [12] library is adopted to parallelize the DPAC and PFAC algorithm on multi-core CPUs. OpenMP is a multithreading programming model which allows forking multiple threads to run concurrently in different processors. OpenMP supports multi-platform and operating systems, including Linux, Mac OS X, and Windows platforms, and provides a set of compiler directives, library routines, and environment variables that control run-time behavior.

OpenMP library has two major types of scheduling, static scheduling and dynamic scheduling. The static scheduling allocates all threads equal iterations before the threads are executed while the dynamic scheduling allocates small number of iterations to a smaller number of threads. When a thread finishes its allocated iterations, the thread returns to get new iterations. The parameter chunk defines the number of contiguous iterations that are allocated to a thread at a time.

As shown in Table 2, for processing an input of length n, the DPAC needs to fork n/s parallel threads where s is the segment size. Because the DPAC divides inputs into small number of segments equal to the number of virtual cores and each thread has constant duration time, static scheduling with chunk value of 1 will utilize all virtual cores (threads) to work simultaneously. On the other hand, because the PFAC needs a lot of threads and each thread has different duration time, dynamic scheduling with large chunk value would satisfy the behavior of the PFAC.

TABLE 2
COMPARISONS OF DPAC AND PFAC IN TERMS OF SCHEDULING

Algorithm	# of thread	Thread duration time	scheduling
DPAC	n/s	constant	static
PFAC	n	dynamic	dynamic

IV. EXPERIMENTAL RESULTS

To evaluate the performance and scalability of the DPAC and PFAC algorithm, the input benchmarks are generated from DEFCON [11] of which sizes are from 256MB to 32GB. The string patterns are extracted from the signature strings of Snort [13] and are grouped into three sets as shown in Table 3. We adopt Amazon EC2 as our experimental environment. Table 4 shows the EC2 instances we choose for evaluating the PFAC and DPAC on different number of cores. OpenMP [12] library is adopted to parallelize the DPAC and PFAC algorithm to achieve optimum performance on multi-core CPUs. All implementations are compiled using GCC 4.6.1 with the compiler flags "-O2". The throughput is defined as the input length

divided by the elapsed time of performing string matching. The preprocessing time for creating the state transition table and opening input files are not taken into account.

TABLE 3
THREE PATTERN SETS EXTRACTED FROM SNORT

Pattern	# of rules	# of characters	# of states
Set 1	1,998	41,997	27,754
Set 2	4,414	98,611	70,284
Set 3	10,075	187,329	126,776

TABLE 4
EXPERIMENTAL MACHINES

Instance type	CPU	Virtual cores	Mem	Memory bandwidth
M2.xlarge	Xeon® X5550	2	17.1GB	32GB/s
M3.xlarge	Xeon® E5-2670	4	15GB	51.2GB/s
M3.xlarge	Xeon® E5-2670	8	30GB	51.2GB/s
CC1.4xlarge	Xeon® X5570	16	23GB	32GB/s
CR1.8xlarge	Xeon® E5-2670	32	244GB	51.2GB/s

First, we evaluate the performance of the PFAC and DPAC on the Amazon computing units with different number of cores including 2, 4, 8, 16, and 32 cores. For processing the inputs of 2GB and the pattern set 3, Fig. 5 shows that both the PFAC and DPAC achieve performance improvement proportional to the number of cores. Furthermore, the PFAC delivers throughput up to 33Gbps on 32 cores, 4.6 times faster than the DPAC. The results show that the performance of PFAC scales up with the number of cores.

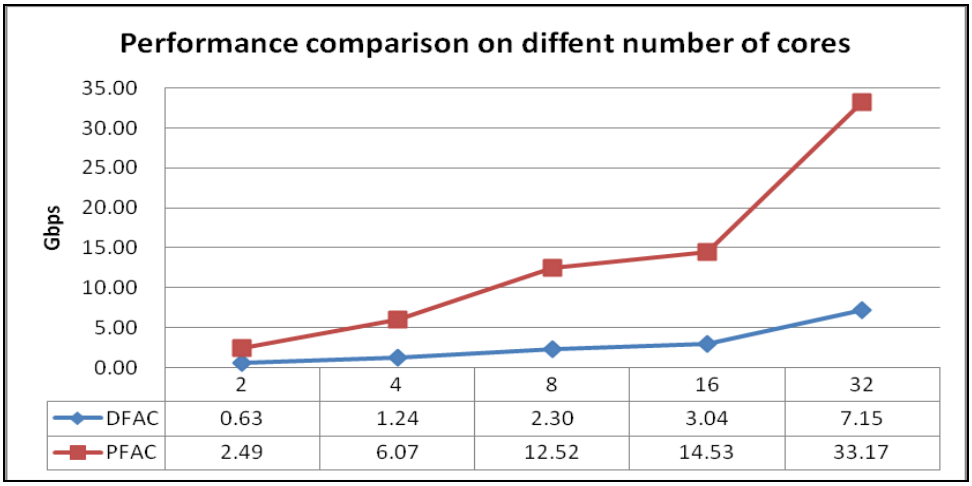
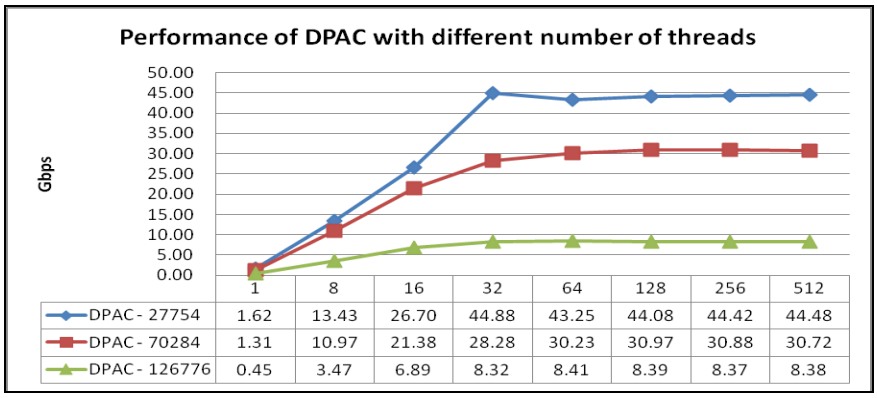


FIGURE 5. PERFORMANCE COMPARISON ON DIFFERENT NUMBER OF CORES

Second, we choose the high memory cluster instance, CR1.8xlarge to evaluate the relationship between performance, thread number, input size, and pattern size. The CR1.8xlarge instance is equipped with two Intel® Xeon® E5-2670 CPUs where each one has eight cores operating at 2.6 GHz. The main memory is 244 GB with the maximum bandwidth of 51.2GB/s. With hyper-threading technology, the 16 physical cores can issue 32 threads simultaneously. In other words, the number of virtual cores is 32.

To evaluate the performance of different number of threads, the number of threads varies from single thread to 512 threads. Fig. 6 shows for processing the 16GB inputs, the performance of DPAC increases with the increasing number of threads. For

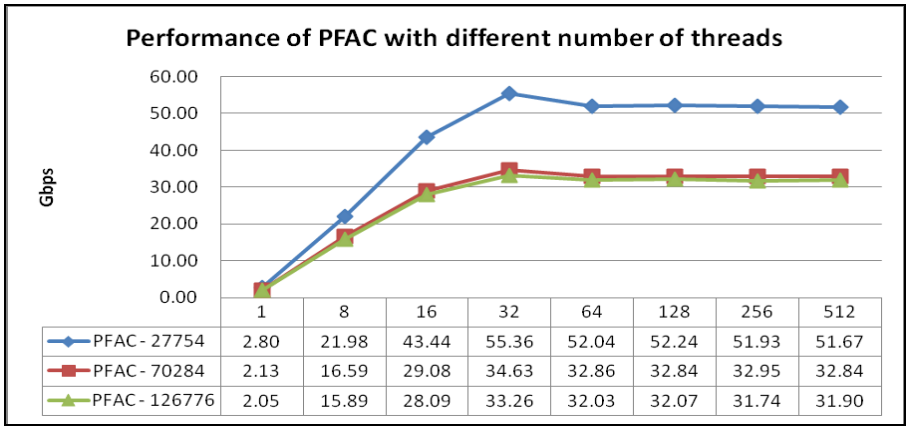
processing small pattern set (27,754 states), the DPAC achieves throughput up to 40 Gbps. But, the performance of DPAC decreases significantly, less than 10Gbps when processing the largest pattern set (126,776 states).



	1	8	16	32	64	128	256	512
DPAC - 27754	1.62	13.43	26.70	44.88	43.25	44.08	44.42	44.48
DPAC - 70284	1.31	10.97	21.38	28.28	30.23	30.97	30.88	30.72
DPAC - 126776	0.45	3.47	6.89	8.32	8.41	8.39	8.37	8.38

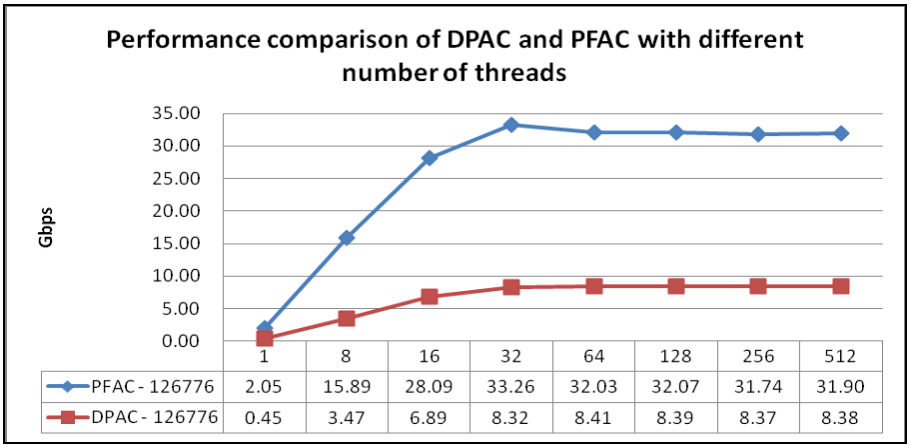
FIGURE 6. PERFORMANCE OF DPAC WITH DIFFERENT NUMBER OF THREADS

On the other hand, the PFAC outperforms the DPAC on processing the largest pattern set. Fig. 7 shows that PFAC still achieves throughput up to 30Gbps on the largest pattern set. Fig. 8 compares the performance of DPAC and PFAC for processing 16GB inputs and the largest pattern set (126,776 states). The PFAC delivers up to 4 times performance improvement over the DPAC with different number of threads. In addition, we can find that both the DPAC and PFAC algorithms are saturated in performance when the number of threads exceeds 32. This result shows that the maximum performance is dominated by the number of virtual cores.



	1	8	16	32	64	128	256	512
PFAC - 27754	2.80	21.98	43.44	55.36	52.04	52.24	51.93	51.67
PFAC - 70284	2.13	16.59	29.08	34.63	32.86	32.84	32.95	32.84
PFAC - 126776	2.05	15.89	28.09	33.26	32.03	32.07	31.74	31.90

FIGURE 7. PERFORMANCE OF PFAC WITH DIFFERENT NUMBER OF THREADS

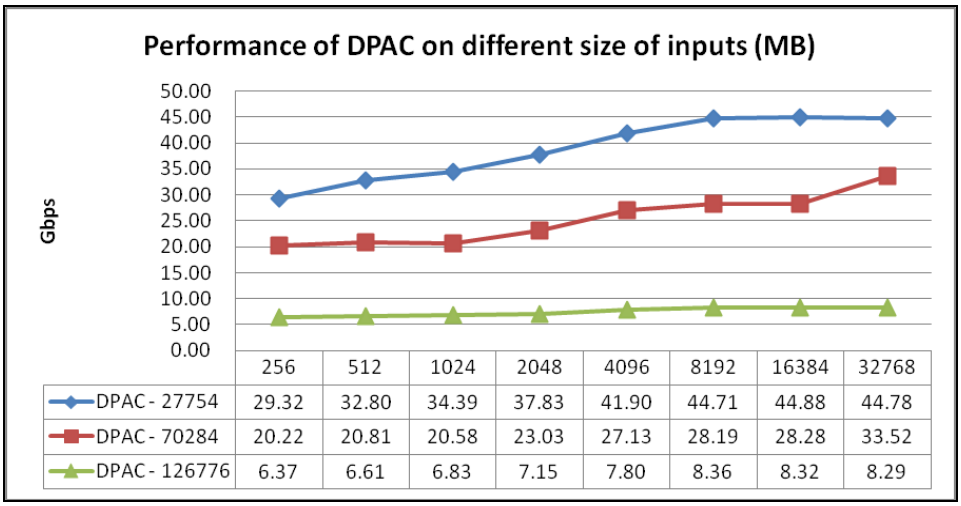


	1	8	16	32	64	128	256	512
PFAC - 126776	2.05	15.89	28.09	33.26	32.03	32.07	31.74	31.90
DPAC - 126776	0.45	3.47	6.89	8.32	8.41	8.39	8.37	8.38

FIGURE 8. PERFORMANCE COMPARISON OF DPAC AND PFAC

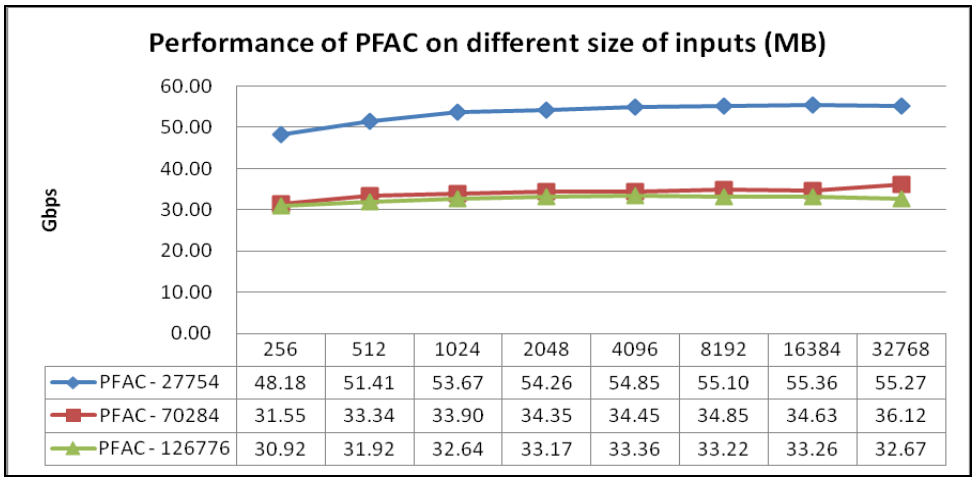
Thirdly, we evaluate the scalability of DPAC and PFAC in terms of the scale of inputs and patterns. The size of inputs varies from 256MB to 32GB. We allocate 32 threads to achieve the maximum performance. Fig. 9 shows the DPAC has considerable performance on processing the inputs larger than 8GB. Fig. 10 shows the PFAC also has good performance on

processing the large-scale inputs. Fig. 11 shows PFAC delivers 4 times performance improvement over DPAC on large scale of inputs.



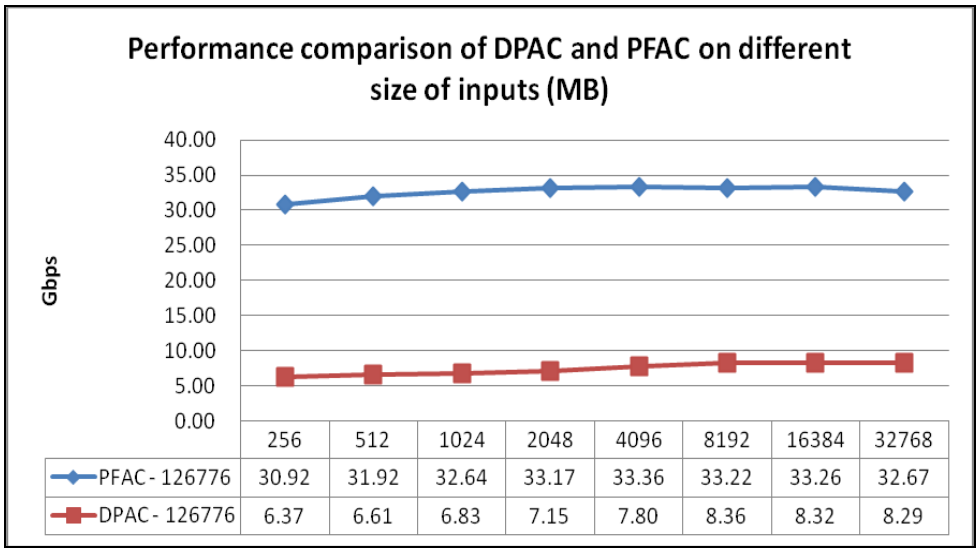
	256	512	1024	2048	4096	8192	16384	32768
DPAC - 27754	29.32	32.80	34.39	37.83	41.90	44.71	44.88	44.78
DPAC - 70284	20.22	20.81	20.58	23.03	27.13	28.19	28.28	33.52
DPAC - 126776	6.37	6.61	6.83	7.15	7.80	8.36	8.32	8.29

FIGURE 9. PERFORMANCE OF DPAC ON DIFFERENT SIZE OF INPUTS



	256	512	1024	2048	4096	8192	16384	32768
PFAC - 27754	48.18	51.41	53.67	54.26	54.85	55.10	55.36	55.27
PFAC - 70284	31.55	33.34	33.90	34.35	34.45	34.85	34.63	36.12
PFAC - 126776	30.92	31.92	32.64	33.17	33.36	33.22	33.26	32.67

FIGURE 10. PERFORMANCE OF PFAC ON DIFFERENT SIZE OF INPUTS



	256	512	1024	2048	4096	8192	16384	32768
PFAC - 126776	30.92	31.92	32.64	33.17	33.36	33.22	33.26	32.67
DPAC - 126776	6.37	6.61	6.83	7.15	7.80	8.36	8.32	8.29

FIGURE 11. PERFORMANCE COMPARISON OF DPAC AND PFAC ON DIFFERENT SIZE OF INPUTS

Finally, we evaluate the impact of thread scheduling on the performance of DPAC and PFAC. Fig. 12 shows that the DPAC achieves better performance using static scheduling with setting chunk as 1. Fig. 13 shows the PFAC achieves better performance using dynamic scheduling with large chunk value.

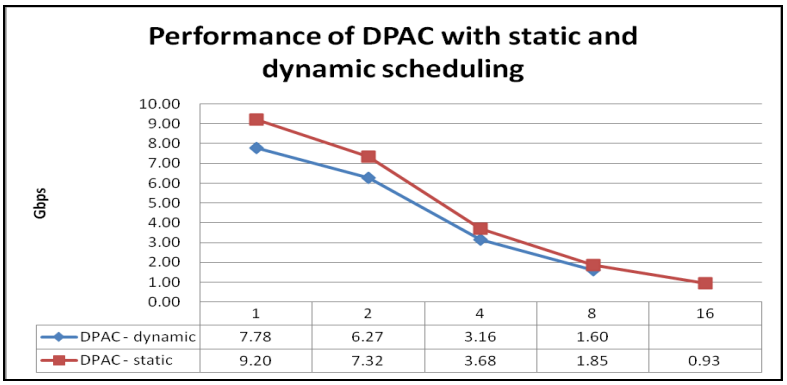


FIGURE 12. PERFORMANCE OF DPAC WITH STATIC AND DYNAMIC SCHEDULING

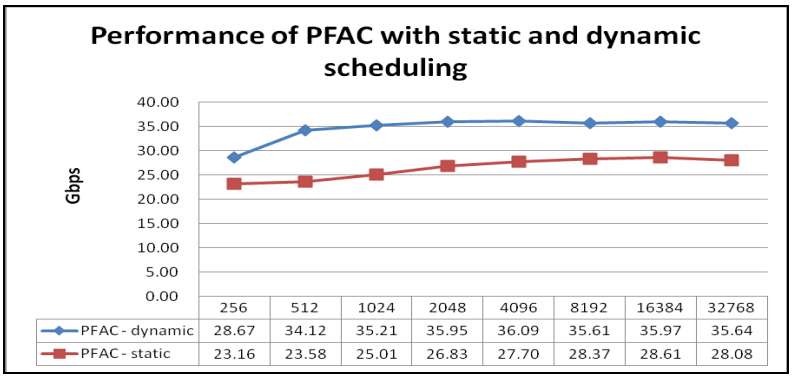


FIGURE 13. PERFORMANCE OF PFAC WITH STATIC AND DYNAMIC SCHEDULING

V. CONCLUSION

In this paper, we have evaluated the performance and scalability of the DPAC and PFAC algorithm on multi-core processor to process large-scale inputs and patterns. We also evaluate the impact of thread scheduling on the performance of DPAC and PFAC. Experimental results show that the PFAC algorithm performing on multi-core processor achieves significant improvement in performance over the DPAC algorithm.

REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*, 18(6):333–340, 1975.

[2] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu, "Accelerating String Matching Using Multi-threaded Algorithm on GPU," in *Proc. IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM)*, 2010.

[3] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, Shih-Chieh Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," *IEEE Transactions on Computers*, 24 Oct. 2012. IEEE computer Society Digital Library. IEEE Computer Society, <http://doi.ieeecomputersociety.org/10.1109/TC.2012.254>

[4] A. Tumeo, S. Secchi, and O. Villa, "Experiences with string matching on the fermi architecture," *Proc. the 24th international conference on Architecture of computing systems*, Como, Italy, 2011.

[5] J. Yu and J. Li, "A Parallel NIDS Pattern Matching Engine and Its Implementation on Network Processor," *Proc. the 2005 International Conference on Security and Management (SAM)*, 2005.

[6] C. V. Kopek, E. W. Fulp, and P. S. Wheeler, "Distributed Data Parallel Techniques for Content-matching Intrusion Detection Systems," *Proc. IEEE Military Communications Conference (MILCOM)*, pp.1-7, 2007.

[7] G. Vasiliadis and S. Ioannidis, "GrAVity: A Massively Parallel Antivirus Engine," *Proc. 13th international conference on Recent advances in intrusion detection (RAID'10)*, 2010.