IEICE Electronics Express, Vol.7, No.20, 1520–1526

Multithreaded pattern matching algorithm with data rearrangement

Doohwan Oh, Seunghun Kim, and Won W. Ro[a)]
School of Electrical and Electronic Engineering, Yonsei University,
134 Shinchon-Dong, Seodaemun-Gu, Seoul,120–749, Korea
a) *wro@yonsei.ac.kr*

Abstract: This letter proposes a multithreaded pattern matching algorithm which can efficiently distribute the patterns to be searched on multiple threads to achieve rapid pattern matching operation. The proposed idea is designed to fully exploit thread-level parallelism to enhance searching speed. By distributing a large number of patterns over multiple threads, pattern matching procedure experiences less cache misses and shows better performance. In addition, we propose to sort the target patterns according to the alphabetic order to achieve efficient data decomposition. From detailed experiments and performance analysis, our algorithm shows remarkable performance gain compared to the original Wu-Manber algorithm.

Keywords: multiple pattern matching, multithreading, data decomposition
Classification: Science and engineering for electronics

References

[1] S. Wu and U. Manber, "A Fast Algorithm for Multi-Pattern Searching," *Department of Computer Science, UA Tech. rep.*, TR94-17, 1994.
[2] C. Gibas and P. Jambeck, "*Developing Bioinformatics Computer Skills*".
[3] D. Gusfield, "*Algorithms on Strings, Trees and Sequences Computer Science and Computational Biology*".
[4] SNORT, [Online] http://www.snort.org/
[5] F. Sanchez, E. Salami, A. Ramirez, and M. Valero, "Parallel processing in biological sequence comparison using general purpose processors," *Proc. IEEE Int. Symp. Workload Characterization 2005.*
[6] AMD Code Analyst Tool, [Online] http://deveoper.amd.com/
[7] Advanced Micro Devices, Inc., "*Software Optimization Guide for AMD Family 10h Processors*," May 2009.

1 Introduction

Multiple pattern matching is an operation to search multiple strings in a target text (*database*) simultaneously; the Wu-Manber algorithm [1] is one of the famous multiple pattern matching algorithms and widely used in many

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

applications including genome search programs and anti-virus software [2, 3, 4]. Although the original algorithm has shown good performance and has been widely adopted in various applications, it has suffered from heavy computation overhead when the number of patterns to be searched becomes large [5].

In fact, the size of memory resource used for a large number of patterns causes excessive cache misses and severely degrades the overall performance. To address this problem, we have developed a multithreaded multiple pattern matching algorithm and have distributed the patterns over the multiple threads; each thread is required to find only the assigned patterns.

In addition, we have developed an efficient way to decompose the patterns over multiple threads; instead of randomly distributing the patterns, we first sort the patterns according to the alphabetic order. This results in a reduced number of data accesses during the prefix comparison process of the algorithm.

Consequently, the proposed idea with alphabetic data decomposition policy improves the overall performance by reducing the amount of workload on a single thread and results in less data accesses. In fact, it has maximum 5.7 times higher performance than the original Wu-Manber algorithm on 32-threads on a single-core.

2 Background Research: Wu-Manber Algorithm

The Wu-Manber algorithm exploits three kinds of tables for matching multiple patterns, which are *shift table*, *hash table*, and *prefix table*. The *bad character shift table* of Boyer-Moore has been adapted to the Wu-Manber algorithms as the form of *shift table*. The table informs the shift value for each matching block; a block is an n-character piece which is used for string comparison. The table is constructed at the pre-processing stage by analyzing the multiple patterns to be simultaneously searched. The shift table contains entries for all two-character blocks within the multiple patterns. This table basically informs the number of characters to be skipped at finding each block pattern. However, if the entry shows a zero value for a certain block pattern, it means that the current block matches to the suffix of some patterns. In this case, the next procedures would be initiated to compare the rest of characters in the patterns.

In order to compare the rest of characters in the patterns, the Wu-Manber algorithm also introduces two additional tables called *hash table* and *prefix table*. The hash table groups the patterns which share the same *suffix*. With the hash table, we now compare the prefixes under the current suffix. At the matching of the prefix, the procedure looks up content of the prefix, which points the location of each full pattern. Indeed, through these tables, the patterns with the same suffix and prefix can be filtered without comparing every character in the patterns.

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

3 Multithreaded implementation of the multiple pattern matching algorithms

To lessen the workload on the original algorithm, we have adopted the multithreading technique from the commercial Linux operating systems. Our idea intends to lessen the centralized workload by distributing the patterns over the multiple running threads. For efficient multithreaded implementation, the patterns are distributed on each thread which has a dedicated private memory address-space for the assigned patterns. On the other hand, we allocate the target text (which is a database to be searched for the pattern matching purpose) on the shared memory address-space which can be easily accessed by all running threads.

The contribution of the proposed idea is found in reducing the number of patterns dedicated on a single thread; by distributing the patterns with the proposed multithreaded algorithm, a large pattern-set is decomposed and distributed over multiple threads. In fact, when the number of patterns to be searched on a thread is large, it could introduce extremely long execution time due to a large number of cache misses. In fact, reduction of the number of patterns assigned on a thread can reduce the number of cache misses. In addition, our multithreading approach which is based on coarse-grained multithreading method can hide cache miss penalty by context switching on a core when the thread has a cache miss.

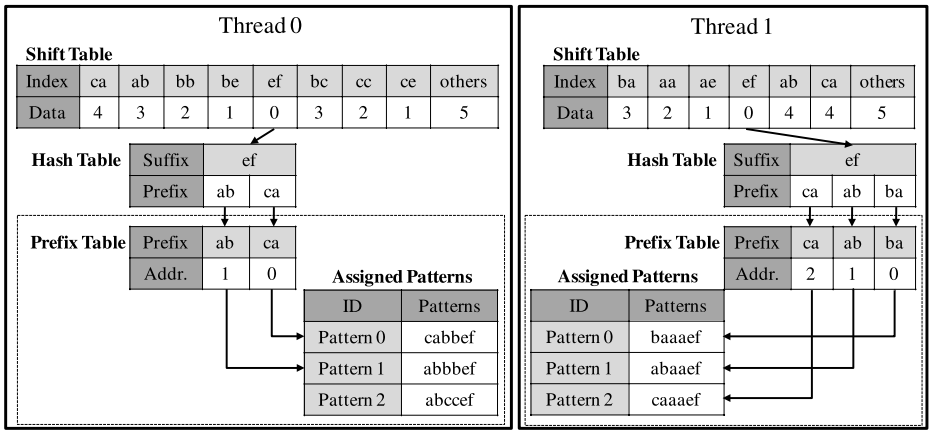
In addition, we propose a further enhancement on the pattern decomposition policy in order to efficiently distribute the patterns over multiple threads; we have made the rearrangement of patterns according to the alphabetic order before we distribute the patterns over multiple threads. With data sorting, the patterns with an identical or similar prefixes can be allocated on a thread. This, consequently, results in a less number of entries in the prefix table, less cache misses, and a less total number of instructions executed.

To show the effectiveness of pattern rearrangement, we show the difference with two different decomposition policies: random data decomposition and sorted data decomposition (Fig. 1). Each of the approaches has two threads; each thread has three assigned patterns which are evenly distributed from total six patterns to be searched simultaneously. Fig. 1-(a) shows the multithreaded Wu-Manber without the rearrangement of patterns (*random order*) and Fig. 1-(b) depicts the scenario with a sorted partitioning according to the alphabetic order. We name a random order partitioning as "*random decomposition*", and a sorted partitioning in alphabetic order as "*sorted decomposition*".

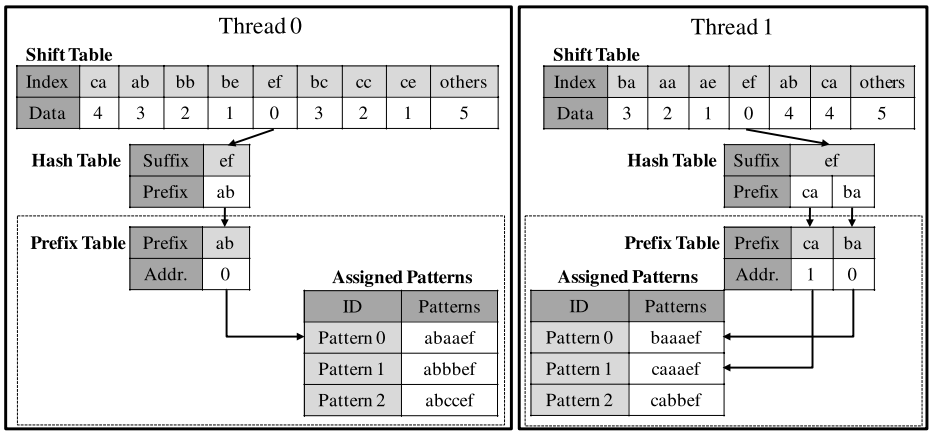
In Fig. 1-(a), the prefix table for *thread 0* has several indexes for "*ab*" and "*ca*". However, the prefix table for *thread 0* in Fig. 1-(b) has only one index for the "*ab*" prefix. Hence, data hit ratio (of course, assuming a lot more patterns to be searched exist) in the sorted decomposition (Fig. 1-(a)) would be improved compared to the random decomposition. Moreover, in Fig. 1-(b), the matching procedure for prefix "*ab*" happens only at *thread*

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

0 since all patterns have the prefix "*ab*" are found only on *thread 0* not on *thread 1*. However, in the case with the random decomposition in Fig. 1-(a), the prefix "*ab*" is found in both *thread 0* and *thread 1*. Hence, the matching procedure for prefix "*ab*" redundantly occurs on both threads. As a result, the decomposition in the alphabetic order not only improves the hit ratio at accessing the prefix table, but also reduces the number of redundant prefix matching procedures.



(a) Multithreaded WM for "*random decomposition*"



(b) Multithreaded WM for "*sorted decomposition*"

Fig. 1. Working examples of multithreaded Wu-Manber algorithms for the random and sorted decomposition policies

4 Experimental results and performance analysis

To demonstrate the advantages of the proposed algorithm, we have performed the detailed experiments with searching 10000, 30000, and 50000 multiple patterns. Fig. 2 shows the processing times of the multithreaded multiple pattern matching algorithms using different numbers of threads. The baseline performance is obtained by using the original Wu-Manber algorithm (WM) [1]. All the experiments are performed using only a single core of

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

the *AMD Phenom X4 Quad-Core processor* (2.2 GHz). In other words, the threads are running only on a single core in a time-shared manner. In this way, we can see the effect of multithreading with a fair condition, excluding the help of multiple physical cores.

In the diagram, the "*sorted decomposition*" means the data decomposition is made according to the alphabetic order and the "*random decomposition*" indicates the randomly distributed data decomposition. The best performance is achieved with running 32 threads at searching 50000 patterns. With the "*sorted decomposition*" policy, the processing time reaches up to 5.7 times faster compared to the original Wu-Manber algorithm which is 'WM' in Fig. 2. Considering that the same amount of hardware is used (*single core*), the processing time is remarkable and demonstrates that our proposed approach is superior to the original single threaded algorithm.

For each configuration, the peak performance is observed with a specific thread numbers. For example, in the case of searching 50000 patterns with the "*random decomposition*" policy, the peak performance is observed with 32 threads; moreover, the performance with 64 threads is severely degraded. We believe this is due to the contexts switching overhead caused by the large number of thread creations. The overhead becomes too large to take advantage of our approach.

In addition, we have observed that the performance improvement with the 10000 pattern matching is not remarkable. With 30000 and 50000 patterns, the processing times show notable improvement. Indeed, as the numbers of patterns increase, the results show better performance with multithreading. Therefore, this observation supports that our approach provides a good solution for the problem caused by heavy computation with the large number of patterns to be searched.

When we compare two different data decomposition policies, the "*sorted decomposition*" policy shows better results in most cases; it has achieved a maximum of 19% better performance compared to the "*random decomposition*" policy (which is achieved with 8-threads on 50000 pattern searching).

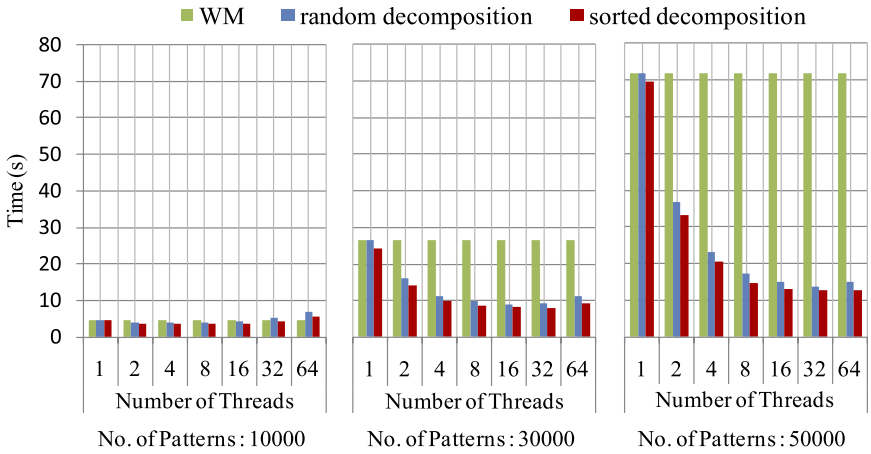


Fig. 2. The processing times with various numbers of threads on a single-core

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

The reason is that the hit rate at accessing prefix table is improved and overall computation load is reduced in the sorted data decomposition policy.

For further analysis, we obtain the data for the processing behavior using "*AMD CodeAnalyst Performance Analyzer*" [6]. Table I shows the hit rate and the number of accesses on L1 data cache and L2 unified cache. The data is obtained with searching 50000 patterns on a single-core. In addition, there are statistics for the number of instructions retired during the overall execution. As the number of threads increases, the hit rate generally shows better results; the hit rates on L1 and L2 cache show remarkable improvement compared to the traditional sequential algorithm (WM). The reason why more L2 accesses happen than the L2 cache misses is due to the automatic stride based prefetching from the L2 cache to the L1 cache [7].

Regardless of data decomposition policy, our approach has almost 60% higher hit rate on the L1 data cache with 64-thread compared to the single-thread execution. Indeed, the "*random decomposition*" shows a little better rate as comparison with "*sorted decomposition*". The case of L2 cache also has shown similar results. This is due to the fact that the sorting operation is added and causes more cache misses in the "*sorted decomposition*" policy. In the case of number of retired instructions, the "*sorted decomposition*" shows a smaller dynamic instruction counts than the "*random decomposition*"; the matching procedures which redundantly occur on multiple threads cause more dynamic instructions to be executed with the random decomposition policy.

Table I. The analyzed data for 50000 patterns with different number of threads in different pattern arrangement on single-core processor

No. of threads	*Random decomposition*					*Sorted decomposition*				
	Instructions retired	L1 data cache		L2 cache		Instructions retired	L1 data cache		L2 cache	
		Access	Hit rate	Access	Hit rate		Access	Hit rate	Access	Hit rate
1	87K	31K	27.8%	103K	29.3%	87K	30K	27.7%	95K	31.6%
2	88K	31K	50.4%	81K	59.0%	88K	30K	50.3%	68K	60.3%
4	81K	28K	61.2%	62K	82.4%	80K	27K	62.2%	50K	83.0%
8	78K	28K	68.9%	50K	97.0%	76K	26K	68.8%	39K	97.5%
16	81K	29K	75.5%	37K	99.5%	76K	26K	74.8%	31K	99.6%
32	89K	31K	82.6%	22K	99.7%	80K	27K	80.7%	23K	99.7%
64	106K	38K	91.8%	10K	99.7%	89K	30K	87.4%	15K	99.6%

5 Conclusion

The proposed multithreaded multiple pattern matching algorithm provides remarkable performance gain with proper data decomposition; the performance results show better execution time with higher cache hit rate. Our method provides a high-performance multithreaded approach which exploits both instruction-level and data-level parallelisms.

Especially, the data decomposition with alphabetic sorting has reduced

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010

the number of instructions executed as well as the number of data accesses. Overall, the proposed idea provides better results when the number of patterns to be searched increases. On conclusion, our approach achieves performance enhancement through not only multithreading but also pattern decomposition policy. As a future work, we will extend the algorithm further targeting today's multi-core processors.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2009-0077326).

DOI: 10.1587/elex.7.1520
Received August 04, 2010
Accepted September 15, 2010
Published October 25, 2010