International Journal of Hybrid Information Technology
Vol.9, No.3 (2016), pp. 31-38
http://dx.doi.org/10.14257/ijhit.2016.9.3.04

A Parallel Algorithm of String Matching Based on Message Passing Interface for Multicore Processors

Jiaxing Qu[1], Guoyin Zhang[1], Zhou Fang[2] and Jiahui Liu[3]

[1]*Harbin Engineering University, Harbin 150001, China*
[2]*Heilongjiang Province Electronic Information Products Supervision Inspection Institute, Harbin 150090, China*
[3]*Harbin University of Science and Technology, College of Computer Science and Technology, Harbin 150080, China*
Corresponding Author:bsuljh@163.com

Abstract

Multicore has long been considered an attractive platform for string matching. However, some existing traditional algorithms of string matching do not adapt to multicore platform, which pose new challenges to parallelism designs. In this paper, we introduce a multicore architecture with message passing interface to address these challenges. We exploit the popular Aho-Corasick algorithm for the string matching engine. Data parallelism is utilized to design optimization technique of string matching. The experiments show that an implementation of the 8-core system achieves up to 10.5 Gbps throughput on the average.

Keywords*: Parallel computing, string matching, multiple pattern, multicore, message passing interface*

1. Introduction

String matching is the key operation in spam filters, network monitoring, virus scanners and intrusion detection systems (IDS). The open source Snort has a lot of content-based rules. The deep packet inspection requires string matching from entire network packets.

However, the problem is worsened by the fact that the size of patterns has increased greatly. In recent years, multicore architecture has been attractive for high performance implementations with parallel computational power. In order to support the heavy network traffic, parallel computing is suitable for preventing an intrusion detection system from becoming a network bottleneck. The processing data can be assigned to every core which executes one or more threads.

In this paper, we introduce a multicore architecture with Message Passing Interface (MPI) for large dictionary string matching. The Aho-Corasick algorithm is used for the string matching engine. Our major contributions are as follows.

(I) The proposed parallel algorithm demonstrates that multicore systems with message passing interface can get high performance. We believe that this technique can be generalized to other algorithms and scientific fields, to handle large data sets.

(II) We analyze the performance of the Aho-Corasick algorithm with the non-deterministic finite automaton (NFA) and the deterministic finite automaton (DFA) which is realized in parallel system. In general, the deterministic finite automaton possesses the steady and reliable performance.

ISSN: 1738-9968 IJHIT

(III) Multicore systems have the basic cores which can execute threads. We test string matching with the assigned data to threads by using MPI. A basic core is assigned from one to two threads. The slight threads can improve the parallel performance. To the best of our knowledge, the parallel Aho-Corasick algorithm of string matching based on shared memory by using MPI is reported for the first time.

The rest of the paper is organized as follows. In Section 2, we introduce related work, background on the Aho-Corasick algorithm, and the data parallelism. Section 3 presents our parallelization strategy. Section 4 gives the experimental results and the performance analysis. Concluding remarks are described in Section 5.

2. Related Work and Background

In this Section, we discuss the related work of string matching and the parallel computing. In addition, we give the definitions and functions in order to explain the algorithm of string matching.

2.1 Related Work

In this section the relationship between the string matching and multicore systems is introduced.

Traditional existing algorithms of string matching have Aho-Corasick algorithm [1], Knuth-Morris-Pratt algorithm [2], Boyer-Moore algorithm [3], Karp-Rabin algorithm [4] and the Bloom filter [5]. Typical and classical algorithms were designed in the single core systems.

In the multicore era, we must face to the problem how to take advantage of the multicore processors [6, 7]. Some of the parallel algorithms of string matching are introduced in different platforms such as Graphics Processing Unit (GPU) [8], Field-Programmable Gate Array (FPGA) [9], the platform-based System on Chip (SoC) [10] and so on.

Besides, over the past decade parallel systems have long been considered an attractive platform for string matching, signature searching in a networked collection of files [11], deep packet inspection [12] and the packed string matching problem [13]. Moreover, some optimal methods have been utilized in parallel string matching, for instance, the pattern group partitioning [14]. String matching is the key problems for speech recognition [15] and spam filtering [16].

As seen above, the typical and classical algorithms would be converted to the parallel algorithms in order to obtain the improved performance.

2.2 Aho-Corasick Algorithm

In this section we present the Aho-Corasick algorithm.

In the open source Snort [17], the key string matching is the Aho-Corasick algorithm, which is a popular algorithm of string matching. The Aho-Corasick algorithm is typical string matching algorithm with multiple patterns, which scans an input text and discoveries occurrences of each of the patterns of a dictionary. The Aho-Corasick algorithm is based on the keyword tree of the given dictionary [18]. Figure 1 describes how to construct a keyword tree.

In multiple pattern string matching, patterns include a finite sequence of symbols from an alphabet [19]. The dictionary consists of a set of patterns. For example, a set of patterns includes the strings P = {the, theory, these, stream, string}. The patterns “the”,

"theory" and "these" have a common prefix. The keyword tree is extended with a failure function. Figure 2 shows how to build a non-deterministic finite automaton with failure function. The failure function is dependent on each state of non-deterministic finite automaton.

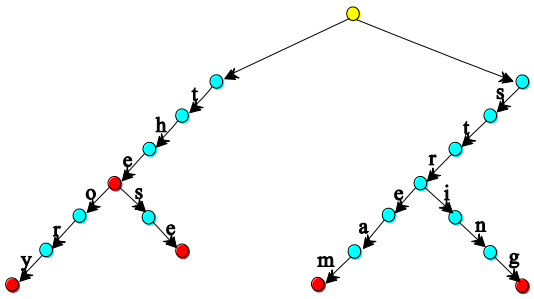


Figure 1. Constructing the Keyword Tree

In Figure 2, the yellow node is the initial state of non-deterministic finite automaton, which calls the root node. The red nodes represent a function which is the pattern from the root node to the current node. The failure function belongs to each state. When a current input character does not match any regular transition, the automaton executes the failure function.

In order to further explain the algorithm, assume that there are three functions as follows.

(I) Function *ft*(*s*1, *s*2) represents the state entered from the current state *s*1 by matching character *s*2, which is defined as the transition function.

(II) Function *ff*(*s*) stands for the state entered at a mismatch, which is named as the failure function.

(III) Function *fo*(root-*s*) gives the one of patterns entered at a match and outputs the string from the root node to the current node *s*, which is called as the output function.

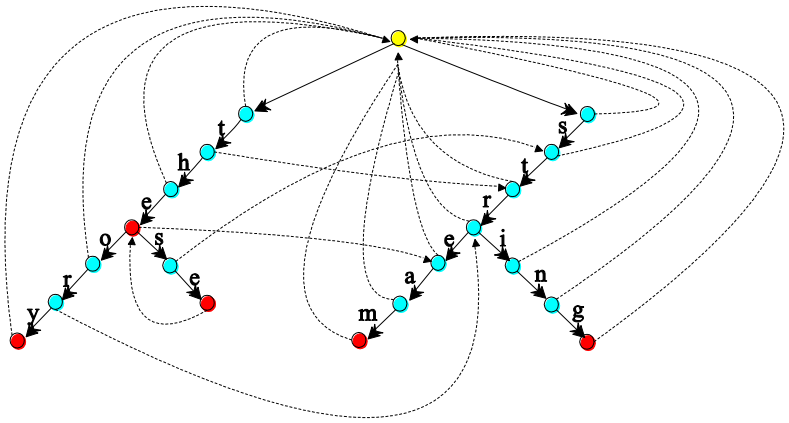


Figure 2. Building the NFA for an Example with the Failure Function

For example, the input text is "theater". The procedure of string matching is starting from the root node. Figure 3 shows the procedure of string matching for the text "theater". Step 1 executes function *ft*(root, t). Step 2 executes *ft*(t, h). Step 3 executes *ft*(h, e). Step 4 executes *ft*(h, e) and *fo*(root-e). The pattern "the" is found out. Step 5 takes the failure function *ff*(e). Step 6 takes *ft*(e, a). Step 7 takes the failure function *ff*(a) and returns to the root node. Step 8

executes *ft*(root, t) such as Step 1. Step 9 executes the failure function *ff*(t) and returns to the root node. Step 10 executes the failure function *ff*(root).

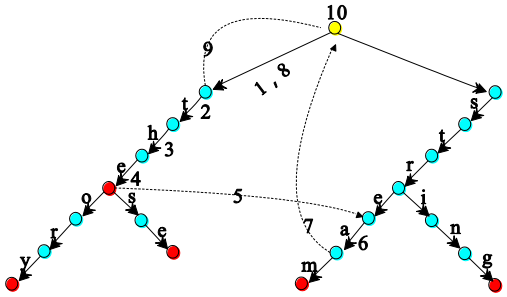


Figure 3. The Procedure of String Matching for the Text "Theater"

The non-deterministic finite automaton is very space efficient which requires only one extra transition per state. The deterministic finite automaton minimizes the number of logical steps, which require a fully populated array of transitions for each state [20]. The deterministic finite automaton needs much memory per state.

2.3 Data Parallelism

Data parallelism is a form of parallel computing which can run at multiple processors in parallel environments. Data parallelism is running counter to task parallelism as another form of parallel computing. In comparison with task parallelism, data parallelism has been used widely. It focuses on distributing the data across computing nodes in multicore platforms.

Data parallelism is realized in the case of each core of multicore systems performs the same work on different parts of distributed data. In other words, each core executes the same instructions, but pieces of data are controlled by different threads. Figure 4 shows data parallelism based on shared memory.

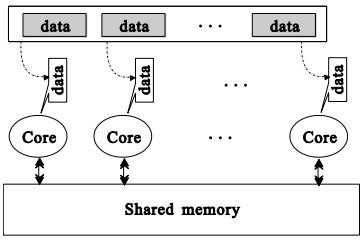


Figure 4. Data Parallelism based on Shared Memory

Most multicore architectures utilize shared memory, where multiple processes have simultaneous access to a buffer of system memory. In other words, the address space of memory is unified [21].

Figure 5 shows data parallelism based on network. Each processor has itself memory. Data communication among processors utilizes the message passing.

Message Passing Interface is a standardized message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. Several well-tested and efficient implementations of MPI have been widely used, which include free or the public domain. Besides, many programming languages such as Fortran, C and C++ support MPI.

Version 1.0 of MPI was released in June 1994. MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented, which can build upon the collection of standard low-level routines to create higher-level routines for the distributed-memory communication environment supplied.

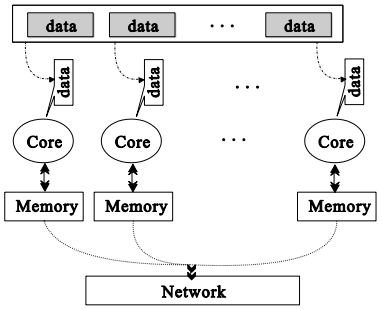


Figure 5. Data Parallelism based on Network

Moreover, MPI provides the interface to allow programmers to use the high-performance message passing operations available on multicore platforms [22].

3. Parallel Algorithm

In this section, the proposed parallel algorithm is presented.

An input text is split in blocks. The blocks have to partially overlap to allow pattern-matching across a boundary, which is equal to the length of the longest pattern in the dictionary minus 1 character.

Threads are executed in the basic core of multicore systems. Each thread processes data blocks with the automaton. Figure 6 shows the parallel algorithm with the data blocks. The parallel algorithm consists of a master thread and slave threads. The master thread assigns data blocks to slave threads. The slave thread executes the searching procedure.

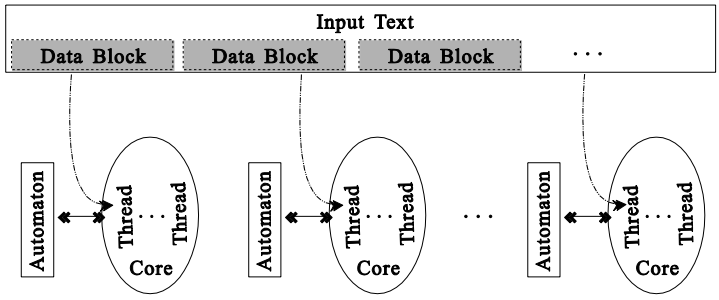


Figure 6. Parallel Algorithm with the Data Blocks

The message communication between the master thread and the slave thread utilizes the functions “MPI_Send” and “MPI_Receive”. Figure 7 shows the message passing between the master thread and the slave thread.

The master thread sends the data block to the slave thread by using the function MPI_Send. The slave thread sends a result to the master thread while the master thread receives the result by using function MPI_Receive. Besides, the master thread builds the automaton in shared memory of the multicore system.

The main function is as follows.
MPI_Init(); // MPI Initialize.
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);// numprocs is the number of cores.
MPI_Comm_rank(MPI_COMM_WORLD, &myid); // myid is the identification of threads

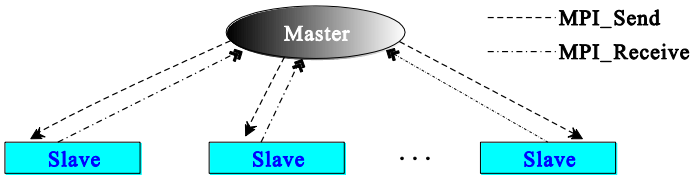


Figure 7. Message Passing between the Master Thread and the Slave Thread

```
if (myid = = 0) // I am the master thread.
{
   Building automaton;
   for (i =1; i < numprocs; i++)
   {
      // To send automaton and data blocks.
      MPI_Send(Buffer, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
   }
   for (i =1; i < numprocs; i++)
   {
      // To receive result from slave threads.
      MPI_Recv(Buffer, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
   }
}
else  // I am the slave thread.
{   // To receive blocks from the master thread.
   MPI_Recv(Buffer, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
   AC(DataBlock, automaton, Result); // To execute searching procedure.
   // To send the results to the master thread.
   MPI_Send(Buffer, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}
MPI_Finalize(); // End MPI.
```

The main function includes the master thread and slave threads, which identifies by using the variable of "myid". For example, if myid is equal to 0 then the thread is the master thread; otherwise, the thread is the slave thread.

4. Performance Analysis

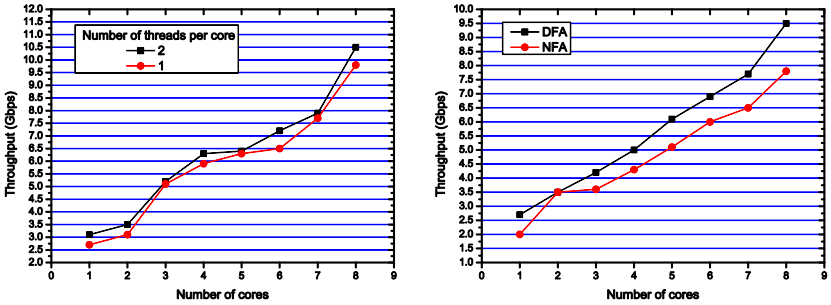
This Section gives the performance analysis of the proposed algorithm.

The multicore system has cores which can execute one or more threads. We test the parallel algorithm on the 8-core multicore system. The one of cores executes the master thread. Others execute slave threads. Figure 8(a) shows the searching procedure for the number of threads per core. The master thread has only one in the multicore systems.

From Figure 8(a), we can see that the performance will enhance when the number of threads per core is increasing. In addition, when the number of cores is growing, the

throughput becomes larger. The maximum value achieves up to 10.5 Gbps throughput on the average. The results are tested by using average values for ten times.

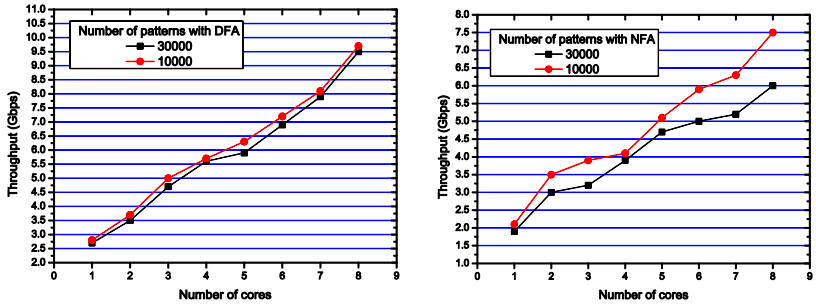
Figure 8(b) shows the performance of the parallel algorithm with the DFA and the NFA. We can see that in comparison with the NFA, the DFA becomes more efficient. The DFA utilizes greater memory than the NFA. We utilize large patterns from Snort to test the parallel algorithm.



(a) (b)

Figure 8. (a) Searching Procedure for the Number of Threads Per Core (b) Searching Procedure with the DFA and the NFA

Figure 9 shows the performance of the parallel algorithm with the number of patterns. It is obvious that the performance becomes efficient when the number of patterns is decreasing. The situation is similar to the NFA and the DFA. We can see that the performance is enhancing when the number of cores is increasing not only for the NFA, but also for the DFA.



(a) (b)

Figure 9. (a) The Number of Patterns with the DFA (b) The Number of Patterns with the NFA

As seen above, the performance of the parallel algorithm with the DFA is steady and reliable.

5. Conclusions

In this paper, we have shown that the multicore system can be successfully utilized to perform high-performance string matching. While implementing the parallel Aho-Corasick algorithm with data parallelism, the throughput achieves up to 10.5 Gbps. We believe that the techniques described in this paper can be successfully applied to other data-intensive applications.

Acknowledgement

This work was supported by the Natural Science Foundation of Heilongjiang Province, China (Grant No. F201304).

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search", Commun., ACM, vol. 18, no. 6, **(1975)**, pp. 333-340.
[2] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast pattern matching in strings", SIAM Journal on Computing, vol. 6, no. 2, **(1977)**, pp. 323-350.
[3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm", Communications of the ACM, vol. 20, no. 10, **(1977)**, pp. 62-72.
[4] R. M. Karp and M. O. Rabin, "Efficient randomized pattern matching algorithms", IBM Journal of Research and Development, vol. 31, no. 2, **(1987)**, pp. 249-260.
[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", Commun. ACM, vol. 13, no. 7, **(1970)**, pp. 422-426.
[6] S. Muthukrishnan, "Simple optimal parallel multiple pattern matching", Journal of Algorithms, vol. 34, no. 1, **(2000)** January 1-13.
[7] Y. Wang and X. Hu, "AOBA: Recognizing object behavior in pervasive urban management", IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 11, **(2014)** November, pp. 2625-2638.
[8] D. Xu, H. Zhang and Y. Fan, "The GPU-based high-performance pattern-matching algorithm for intrusion detection", Journal of Computational Information Systems, vol. 9, **(2013)** May, pp. 3791-3800.
[9] H. J. Kim, "A failureless pipelined Aho-Corasick algorithm for FPGA-based parallel string matching engine", Lecture Notes in Electrical Engineering, vol. 339, **(2015)**, pp. 157-164.
[10] Y.-D. Lin, K.-K. Tseng, T.-H. Lee, Y.-N. Lin, C.-C. Hung and Y.-C. Lai, "A platform-based SoC design and implementation of scalable automaton matching for deep packet inspection", Journal of Systems Architecture, vol. 53, no. 12, **(2007)** December, pp. 937-950.
[11] Z. Ying and G. R. Thomas, "Signature searching in a networked collection of files", IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 5, **(2014)** May, pp. 1339-1348.
[12] H. Kim, H. Hong, D. Baek and S. Kang, "A pattern partitioning algorithm for memory-efficient parallel string matching in deep packet inspection", IEICE Transactions on Communications, vol. E93-B, no. 6, **(2010)** June, pp. 1612-1614.
[13] B.-K. Oren, B. Philip, B. Dany, G. Leszek, G. Roberto and W. Oren, "Towards optimal packed string matching", Theoretical Computer Science, vol. 525, **(2014)**, pp. 111-129.
[14] H. Kim and S. Kang, "A pattern group partitioning for parallel string matching using a pattern grouping metric", IEEE Communications Letters, vol. 14, no. 9, **(2010)** September, pp. 878-880.
[15] Y.-J. He, M.-G. Fu and G.-L. Sun, "An overview of speech feature enhancement method", Journal of Harbin University of Science and Technology, vol. 19, no. 2, **(2014)**, pp. 19-25.
[16] H. Yang, G.-L. Sun and Y.-J. He, "Spam filtering with naive bayes", Journal of Harbin University of Science and Technology, vol. 19, no. 1, **(2014)**, pp. 49-53.
[17] "Snort. The Open Source Network Intrusion Prevention and Detection System", http://www.snort.org.
[18] C.-C. Chen and S.-D. Wang, "An efficient multi-character transition string-matching engine based on the aho-corasick algorithm", Transactions on Architecture and Code Optimization, vol. 10, Issue 4, Article No. 25, **(2013)** December.
[19] C.-C. Chen and S.-D. Wang, "A hybrid multiple-character transition finite-automaton for string matching engine", Microprocessors and Microsystems, vol. 39, no. 2, **(2015)** March, pp. 122-134.
[20] Y. Fan, H. Zhang, J. Liu and D. Xu, "An efficient parallel string matching algorithm based on DFA", Proceedings of International Conference on Trustworthy Computing and Services, **(2013)** May 28-June 2, Beijing, China.
[21] J. Liu, H. Zhang, D. Song, G. Sun, W. Bi and M. K. Buza, "A parallel encryption algorithm of the logistic map for multicore with OpenMP", Proceedings of 8th International Forum on Strategic Technology, **(2013)** June 28-July 1, Ulaanbaatar, Mongolia.
[22] J. Liu, D. Song and Y. Xu, "A parallel encryption algorithm for dual-core processor based on chaotic map", Proceedings of Fourth International Conference on Machine Vision: Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies, **(2011)** December 9-10; Singapore, Singapore.

