International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:10, 2007

A Practical Distributed String Matching Algorithm Architecture and Implementation

Bi Kun, Gu Nai-jie, Tu Kun, Liu Xiao-hu, and Liu Gang

***Abstract*—**Traditional parallel single string matching algorithms are always based on PRAM computation model. Those algorithms concentrate on the cost optimal design and the theoretical speed. Based on the distributed string matching algorithm proposed by CHEN, a practical distributed string matching algorithm architecture is proposed in this paper. And also an improved single string matching algorithm based on a variant Boyer-Moore algorithm is presented. We implement our algorithm on the above architecture and the experiments prove that it is really practical and efficient on distributed memory machine. Its computation complexity is *O(n/p + m)*, where *n* is the length of the text, and *m* is the length of the pattern, and *p* is the number of the processors.

***Keywords*—**Boyer-Moore algorithm, distributed algorithm, parallel string matching, string matching.

I. Introduction

STRING matching problem received much attention over the years due to its importance in various applications such as text processing, information retrieval, computational biology and intrusion detection [1], [2]. All those applications require highly efficient algorithm to find all the occurrences of a given pattern in the text.

According to whether the text or the pattern needs preprocessing, the string matching algorithms can be divided into two categories. If the pattern needs to be preprocessed, it is called online string matching algorithm; if constructing a data structure on the text is required, it is called index string matching algorithm. In this paper, we focus on online string matching algorithm.

The Knuth-Morris-Pratt (KMP) algorithm [3] and the Boyer-Moore (BM) algorithm [4] are both well-known single string matching algorithms. The KMP algorithm guarantees both independence from alphabet size and worst-case execution time linear in the pattern length, and the worst-case computation complexity is $O(n+m)$; on the other hand, the BM algorithm provides near optimal average case and best-case behavior, and it only needs $O(n/m)$ comparisons in the best case. Hundreds of algorithms based on the KMP or BM algorithm have been proposed, such as BMH algorithm [5], QS algorithm [6] and so on [12]. Based on the key observation on the characteristics of the "bad character" and "good suffix" in BM algorithm, Cantone and Faro [7] proposed a more efficient algorithm called Fast Search. Though it keeps the good characteristics of the BM algorithm, its worst-case computation complexity is still $O(n\times m)$.

Most proposed parallel string matching algorithms are usually based on PRAM (Parallel Random Access Machine) computation model [8]-[10], and this model is not practical in realistic distributed computation environment. CHEN [11] designed a parallel KMP string matching algorithm on distributed memory machine. The distributed memory model is practical in most parallel machines such as massive parallel processor machine and cluster machine. In PRAM model, all those operations are abstracted as they have the same cost, but in the realistic implementation, it is not the case. Memory access and communication between processors usually spend much more clock cycles than a single computation in processor. So, in the practical design of parallel algorithm, those impact factors should been concerned.

In this paper, based on the distributed algorithm published by CHEN [11], we propose a practical distributed string matching algorithm architecture. We observed that nearly all the string matching algorithms could be implemented in the framework, not only the KMP algorithm which had been implemented by CHEN.

This paper also presents an improved single string matching algorithm by making some modifications on the algorithm proposed by Cantone and Faro [7]. With those modifications, the good characteristics of the algorithm are still preserved, while the worst-case computation complexity has reduced from $O(n\times m)$ to $O(n+m)$.

We also implement our string matching algorithm on the architecture proposed above. The experimental results on our cluster prove that this architecture is suitable in distributed memory machine, and our string matching algorithm is really practical.

This paper makes the following research contributions:

Manuscript received November 14, 2005.

BI Kun is with the Department of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, P.R.China, 230027 (corresponding author to provide phone: 86-551-3601547; e-mail: bikun@mail.ustc.edu.cn).

GU Nai-jie is with the Department of Computer Science and Technology, University of Science and Technology of China (e-mail: gunj@ustc.edu.cn).

TU Kun is with the Department of Computer Science and Technology, University of Science and Technology of China (e-mail: tukun@ustc.edu).

Liu Xiao-hu is with the Department of Computer Science and Technology, University of Science and Technology of China (e-mail: stain@mail.ustc.edu.cn).

LIU Gang is with the Department of Computer science and technology, University of Science and Technology of China (e-mail: liugang@mail.ustc.edu.cn).

1) A practical distributed string matching algorithm architecture that is suitable for paralleling nearly all the string matching algorithms in distributed memory machine.
2) An improved string matching algorithm that reduces the complexity of Fast-Search algorithm from $O(n \times m)$ to $O(n+m)$.
3) An implementation of the improved string matching algorithm on the distributed architecture and a show of its adaptability for paralleling string matching algorithms.

The rest of the paper is organized as follows. Section II reviews the related work. Section III describes our distributed architecture and presents the improved string matching algorithm and its implementation in the distributed architecture. Section IV analyzes the computation complexity of our algorithm. Section V evaluates the practical performance of the distributed string matching algorithm. Finally, section VI concludes the paper with a summary.

II. Related Work

A. Single String Matching Algorithms

With the publication of both the KMP and the BM string matching algorithms, lots of papers worked on improving the efficiency of the original algorithm. When the alphabet size is large enough, Horspool [5] suggested using only "bad character" rule shifts when a mismatch occurs. This algorithm is faster in practice when the alphabet size is not small because it does not need to make a comparison between the "bad character" shift and the "good suffix" shift which is used to shift the pattern when a mismatch occurs. Sunday [6] proposed to shift the pattern using "bad character" shift according to the next character of this match attempt in the text. Those algorithms all took the advantage of the "bad character" rule characteristic, which guaranteed the near optimal average case behavior. A list of other algorithm variants based on KMP or BM and a fairly complete bibliography are available on the web site [12]. Cantone and Faro [7] discovered that "the Horspool bad character rule leads to larger shift increments than the good suffix rule if and only if a mismatch occurs immediately, while comparing the pattern p with the window", so they suggested using the "bad character" rule when the mismatch occurred immediately in comparing the text, otherwise, using the "good suffix" rule.

If the pattern is periodical, the worst-case computation complexity of BM algorithm is $O(n \times m)$, so are the BMH algorithm, the QS algorithm and the Fast-Search algorithm. Galil [13] proposed an algorithm to improve the worst case running time of the BM algorithm, and he proved the improving BM algorithm is $O(n+m)$ in the worst case. Richard Cole [14] presented the tight bounds on the complexity of the BM algorithm.

B. Parallel String Matching Algorithms

The first optimal parallel string matching algorithm was proposed by Galil [8]. On SIMD-CRCW model, this algorithm required $n/\log n$ processors, and the time complexity is $O(\log n)$; on SIMD-CREW model, it required $n/\log^2 n$ processors and the time complexity is $O(\log^2 n)$. Vishkin [9] improved this algorithm to ensure it is still optimal when the alphabet size is not fixed. In [10], an algorithm used $O(n \times m)$ processors was presented, and the computation time is $O(\log \log n)$. A parallel KMP string matching algorithm on distributed memory machine was proposed by CHEN [11]. The algorithm is efficient and scalable in the distributed memory environment. Its computation complexity is $O(n/p+m)$, and p is the number of the processors.

III. Distributed Architecture and Algorithm

A. The Distributed String Matching Algorithm Architecture

Nowadays, the frequency of the processor is high. The speed gap between processors and memory access and communications between processors is large. Because the communication between different computing nodes requires the message to be sent by operating system and the network protocol is heavy for a single communication. It usually takes much more cycles to send data to other computing nodes than computing it in local processor.

Our distributed string matching algorithm architecture is based on the following three assumptions:

1) The computing environment is distributed memory environment, such as cluster computing environment.
2) Communication between processors costs much more time than computing it in local processors.
3) The length of text is much longer than the length of the pattern string. The text is partitioned and then assigned to each processor before processing.

The proposed distributed architecture is shown in Fig 1.

```
(1) PE0 broadcast pattern string pat [1, m]
(2) for i=0 to p-1 par-do
        Call procedure BUILD
    end for
(3) for i=0 to p-1 par-do
        StringMatchingAlgorithm (Ti, pat)
    end for
(4) Deal with the boundary condition
```

Fig. 1 The distributed string matching algorithm architecture

In step 1, the processor with number zero broadcasts the pattern string to other processors. The length of the pattern is ***m***, and stores in an array named pat. This step can be implemented by the binomial tree-based communication strategy or the Fibonacci communication strategy [15].

In step 2, all the processors call the procedure "BUILD" to

preprocess the pattern string in parallel. For example, if you want to parallel BM algorithm, you need to compute the "bad character" shift and "good suffix" shift in this step.

In step 3, all the processors call the string matching algorithm to search the pattern string locally in the text T_i. T_i denotes the text assigned to processor whose number is ***i***. The string matching algorithm could be all the classical string matching algorithms, such as the KMP algorithm and the BM algorithm.

In step 4, the boundary condition should be considered, because the pattern matching may occur just across the text T_i and T_{i+1}. The boundary pattern matching condition can be solved by sending the last ***m-1*** characters' information from processor PE_i to the next processor PE_{i+1}.

B. The Improved Single String Matching Algorithm

Our improved single string matching algorithm is based on the algorithm Fast-Search published by Cantone and Faro [7]. The aim of our algorithm is to improve the worst case complexity of the Fast-Search algorithm while keeping its practical efficiency.

The key idea of our algorithm is that when a pattern matching occurs, the position in the text is recorded, and in the next matching attempt, a pattern matching will be reported when the character comparison comes to that position instead of the end of this attempt. This method was first discovered by Galil [13].

Fig. 2 shows our improved single string matching algorithm.

```
j = 0;
pos = 0;
end = m − bmGs[0];
while (j <= n − m ) {
  if(pat[m-1] != T[j+m-1] ) {
    shift = bmBc[T[j+m-1]];
    pos = 0;
  }
  else {
    for ( i=m-2; i>= pos && pat[i] == T[i+j];i-- );
    if ( i < pos ) {
      OUTPUT( match at position j+1 );
      shift = bmGs[0];
      pos = end;
    }
    else {
      shift = bmGs[i];
      pos = 0;
    }
  }
  j = j + shift;
}
```

Fig. 2 The improved single string matching algorithm

In this algorithm, we have not presented the definition and the construction algorithm for arrays named "bmBc" and "bmGs", because the definition and computation algorithm can be found in the web site [12] through the hyperlink titled "Boyer-Moore algorithm".

We use the variable "end" to record the position, and the variable "pos" indicates where the current matching attempt should stop and a matching position will be reported at that time. In section IV, we will prove that the worst-case time complexity of this improved algorithm is $O(n+m)$.

C. Implement the Improved Algorithm in the Distributed String Matching Algorithm Architecture

In this part, using the proposed distributed string matching algorithm architecture, we parallel the improved single string matching algorithm.

Step 1 is the same as mentioned in the distributed algorithm architecture. The procedure "BUILD" in step 2 is to call the construction algorithm [12] to compute the "bad character" shift and "good suffix" shift which are stored in array "bmBc" and "bmGs" separately. In step 3, the common procedure "StringMatchingAlgorithm" is implemented by our improved single string matching algorithm. In step 4, the boundary condition should be considered. The last ***m-1*** characters are sent from processor PE_i to the next processor PE_{i+1}, and PE_{i+1} receives the ***m-1*** characters from processor PE_i. After this phase, each processor checks whether a match occurs just on the boundary or not.

IV. Complexity Analysis

A. The Time Complexity of the Improved Single String Matching Algorithm

Definition. A string u is periodical if it can be written in the form wv^k, where w is a proper suffix of v and $k \geq 2$. Equivalently, u is said to be period of v.

Theorem 4.1. If the pattern is non-period, the improved single string matching algorithm performs at most *4n* comparisons when matching a pattern of length *m* against a text of *n*.

Proof. Richard Cole [14] proved that "If the Boyer-Moore string matching algorithm determines its shifts using only the good suffix shift rule, then it performs at most 4n comparisons when matching a non-period pattern of length *m* against a text of length *n*." In his proof, he used the amortized analysis method, and defined the potential function was "3·#positions not yet shifted over + #unread text characters." So we can extend his proof to the case of using the "bad character" shift rule when a mismatch occurs in the first comparison of this attempt, because in this case, the pattern at least needs to shift one position, so the potential is reduced by at least 3. So, Cole's proof is still hold in this case. □

Lemma 4.1. If the pattern is period and the pattern never occurs in the text, the algorithm performs at most *4n* comparisons.

Proof. We use the same potential function as mentioned above.

Case 1: The mismatch occurs in the first $|v|$ comparisons in one attempt. In this case, the pattern will shift forward at least $|u|-|w|$, so the potentials reduced by at least $3\cdot(|u|-|w|)-|v|$.

Case 2: The mismatch occurs after at least $|v|+1$ times of

comparisons. In this case, the next attempt at most compares $|u|$-($|v|$-1) characters, and all those characters are yet not covered. The next shift is at least $|u|$-$|w|$, so the amortized cost is at most zero.

Because the initial potential is $4n$, so the lemma holds. □

Theorem 4.2. If the pattern is period, the improved single string matching algorithm performs at most $4n$ comparisons when matching a pattern of length m against a text of n.

Proof. We use the same potential function as mentioned above.

Case 1: There is no occurrence of the pattern in the text. From lemma 4.1, it holds.

Case 2: A match occurs when all the comparisons are in the characters which are not compared before. In the case the "#unread text characters" is equal to the comparison cost, so the amortized cost is below zero.

Case 3: A match occurs just after a match in the previous attempt. In the case, because the position is recorded when a match occurs, and in the next match, it only needs to check those characters unread before. So the "#unread text characters" is equal to the comparison cost in this match. The amortized cost is below zero.

Case 4: A match occurs just after a mismatch. In this case, from lemma 4.1, the number of previous comparisons is no more than $4 \cdot k$, where k denotes the number of characters ever read in the text. When a match occurs, in the next attempt, there are only two cases. The first case is another match occurs, the other is a mismatch occurs. The first case returns to the case when a match occurs. If the match continues occurring till the end of text, the amortized cost is below zero. So, only the latter case is concerned. If a mismatch occurs, the number of comparisons in this attempt is less than $|v|$, and the shift is no less than $|u|$-$|w|$. So the amortized cost is no more than zero in totals.

Because the initial potential is $4n$, so this theorem holds. □

Theorem 4.3. The worst case time complexity of the improved single string matching algorithm is $O(n+m)$.

Proof. Combined the theorem 4.1 and theorem 4.2, it is clear that the improved algorithm performs at most $4n$ comparisons no matter that the pattern is periodical or not. Considered the length of the pattern is m, and the computation complexity of the value of array "bmBc" and "bmGs" is $O(a)$ and $O(m)$ separately, where a denotes the alphabet size. So the worst case time complexity of the improved single string matching algorithm is $O(n+m)$. □

B. The Complexity of the Parallel Improved Single String Matching Algorithm

The computation complexity of the procedure "BUILD" is $O(m)$, because in this phase, the values of array "bmBc" and "bmGs" are computed, m is the length of the pattern. In step 3, the time complexity of our improved single string matching algorithm is $O(n/p+m)$, because the text of length n is partitioned and then assigned to each processor before processing by our assumptions in section III. So in each processor, the length of the assigned text is n/p, and p is the number of the processors. In step 4, at most $2m-2$ characters need to be checked further, so the complexity is $O(m)$ in this step. So the total time complexity of the algorithm is $O(n/p+m)$.

In step1, the pattern of length m needs to be broadcasted to all of the processors, if the binomial tree-based communication strategy is used, the communication complexity in this step is $O(m\log p)$. In step 4, the last $m-1$ characters in each processor except the last one have to be sent to the next processor, so the communication complexity in this step is $O(m)$. So the total communication complexity of the algorithm is $O(m\log p+m)$.

V. Experimental Results

A. Experimental results of the Improved Single String Matching Algorithm

We present the experimental results in Fig.3 which allow comparing the running time of the following three single string matching algorithms: Fast-Search, BMH and our improved string matching algorithm. All those three algorithms have been implemented in the **C** programming language. The computer is with AMD Athlon processor of 1.67GHz and 256M memory. The length of the text is fixed to 10,000,000 Bytes, and the length of the pattern is fixed to 50 Bytes. The size of the alphabet is 4, 8, 12, 16, and 26. The text and pattern are randomly generated from the alphabet. The results show that when the size of the alphabet is small, the performance of the Fast-Search algorithm or our improved Fast-Search algorithm is better than the BMH algorithm and our improved Fast-Search algorithm is a little better than the Fast-Search algorithm; when the size of the alphabet has increased to 26, the performance of the Fast-Search algorithm and our improved Fast-Search is similar and a little better than BMH algorithm. Our improved Fast-Search algorithm performs fewer comparisons than the other two algorithms in the worst case, because it guarantees that the worst-case time complexity is $O(n+m)$, while the other two algorithms are both $O(n\times m)$ in the worst case.

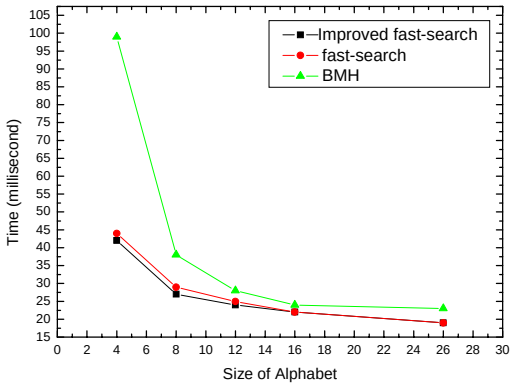

Fig. 3 The running times of single string matching algorithms

B. Experimental Results of the Paralleled Single String Matching Algorithm

We evaluate the performance of our practical distributed string matching algorithm architecture by paralleling our improved Fast-Search algorithm in the proposed architecture. The experimental results are shown in Table 1. The algorithm is implemented in the **C** programming language with MPI. We have run it in our own cluster. The computers with AMD Athlon processor of 1.67GHz and 256M memory are connected by a switch of 100Mbit/s. The operating system is "Microsoft Windows 2000 professional". The MPICH version is 1.2.5. The length of the total text is 80 megabyte, and the length of the pattern is 100 Bytes, and the size of the alphabet is fixed to 16. In Table1, the first column is the number of the processors. The second column is the computing time in milliseconds which is the sum of the preprocessing time and the time running in searching the pattern in the text. The third column is the communication time in milliseconds between processors. The total time in milliseconds in the fourth column is the sum value of computing time and communication time. In the last column, the ratio is defined as $((T_{Comp}+T_{Comm})-(T_{Serail}/p))/(T_{Serail}/p)$, T_{Comp} denotes the computing time, T_{Comm} denotes the communication time, p denotes the number of processors, and T_{Serial} denotes the computing time when $p=1$. The ratio reflects the gap between the realistic performance and the ideal performance. The last column shows that our distributed string matching algorithm architecture is really practical when paralleling the string matching algorithm. The ratio is also a measurement of the performance influenced by the communication time between processors.

TABLE I
RUNNING TIMES IN DISTRIBUED MEMORY MACHINE

Time / Proc	Computing time	Communication time	Total time	Ratio
1	192.15	0	192.15	0.000
2	94.76	3.63	98.39	0.024
3	64.09	6.66	70.75	0.105
4	46.64	6.21	52.85	0.100

VI. Conclusion

In this paper, we presented a practical distributed string matching algorithm architecture which is suitable and efficient in distributed memory computing environment. We also presented an improved single string matching algorithm based on the Fast-Search algorithm proposed by Cantone and Faro. And then, we paralleled the improved algorithm in our distributed architecture. This distributed architecture is also suitable for paralleling the multipattern string matching algorithms and approximate string matching algorithms.

References

[1] Zheng Liu, Xin Chen, James Borneman and Tao Jiang, "A fast algorithm for approximate string matching on gene sequences," in *Symposium. 16th Annu. Combinatorial Pattern Matching, LNCS, Springer-Verlag*, vol. 3537, pp. 79-90, June 2005.
[2] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, vol. 4, pp. 2628-2639, March 2004.
[3] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, pp. 323-350, 1977.
[4] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762-772, 1977.
[5] R. N. Horspool, "Practical fast searching in strings," *Software - Practice and Experience*, vol. 10, pp. 501-506, 1980.
[6] D. M. Sunday, "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, pp. 132-142 ,1990.
[7] D. Cantone and S. Faro, "Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm," in *Proc. Second International Workshop on Experimental and Efficient Algorithms, LNCS, Springer-Verlag*, Vol. 2647, pp. 47–58, May 2003.
[8] Z. Galil, "Optimal parallel algorithms for string matching," in *Proc. 16th Annu. ACM symposium on Theory of computing*, pp. 240-248, 1984.
[9] U. Vishkin, "Optimal parallel matching in strings," *Information and control*, vol. 67, pp. 91-113, 1985.
[10] Y. Takefuji, T. Tanaka, and K. C. Lee, "A parallel string search algorithm", *IEEE Trans. Systems, Man and Cybernetics*, vol. 22, pp. 332-336, March-April 1992.
[11] CHEN Guo-liang, LIN-Jie, and GU Nai-jie, "Design and analysis of string matching algorithm on distributed memory machine," *Journal of Software*, vol. 11, pp. 771-778, 2000.
[12] C. Charras and T. Lecroq, "Exact string matching algorithms," Laboratoire d'Informatique de Rouen Université de Rouen. Available: http://www-igm.univ-mlv.fr/~lecroq/string/
[13] Z. Galil, "On improving the worst case running time of the Boyer-Moore string matching algorithm," *Communications of the ACM*, vol. 22, pp. 505-508, 1979.
[14] R. Cole. "Tight bounds on the complexity of the Boyer-Moore string matching algorithm," *SIAM Journal on Computing*, vol. 23, pp. 1075–1091, 1994.
[15] GU Nai-jie, LI Wei and LIU Jing, "Fibonacci series-based multicast algorithm," *Chinese Journal of Computers*, vol. 25, pp. 365-372, 2002.

BI Kun was born in 1981. He is a Ph.D. student in the Department of Computer Science and Technology, USTC. His research interests include parallel and distributed computing.
GU Nai-jie was born in 1961. He is a Professor and Doctoral Advisor in the Department of Computer Science and Technology, USTC. His research interests include parallel computing architecture, interprocessor communication, and high-performance computing
TU Kun was born in 1980. He is a Ph.D. student in the Department of Computer Science and Technology, USTC. His research interests include parallel and distributed computing.
LIU Xiao-hu was born in 1979. He is a Ph.D. student in the Department of Computer Science and Technology, USTC. His research interests include secure multicast and distributed computing.