Distributed Splay Suffix Arrays： A New Structure for Distributed String Search

Tu Kun, Gu Nai-jie, Bi Kun, Liu Gang and Dong Wan-li

Abstract— As a structure for processing string problem, suffix array is certainly widely-known and extensively-studied. But if the string access pattern follows the "90/10" rule, suffix array can not take advantage of the fact that we often find something that we have just found. Although the splay tree is an efficient data structure for small documents when the access pattern follows the "90/10" rule, it requires many structures and an excessive amount of pointer manipulations for efficiently processing and searching large documents. In this paper, we propose a new and conceptually powerful data structure, called splay suffix arrays (SSA), for string search. This data structure combines the features of splay tree and suffix arrays into a new approach which is suitable to implementation on both conventional and clustered computers.

Keywords— suffix arrays, splay tree, string search, distributed algorithm

I. Introduction

TODAY large databases become available, such as full text of newspapers of Web pages, and Genome sequences, therefore it is important to efficiently store them on memory for quick queries. In order to achieve this goal, some indexing data structures and searching tools have been introduced. The main approaches are: word indexes, character n-gram indexes, and suffix indexes.

Word indexes have the advantage of supporting very fast word queries, while they have difficulty with indexing unstructured texts- like DNA-sequences or some Asian language texts [1]. While character n-gram indexes enable us to index unstructured texts, the search for a lengthy query or regular expression is complicated and inefficient [2].Suffix indexes have been designed to overcome the above limitations by dealing with arbitrary texts, but this increases the cost due to the additional space occupied by the underlying indexing data structure [3] [4] [5]. Examples of such indexes are: suffix tree [6], suffix array [7]. Similar concepts were independently proposed in Oxford English Dictionary project [8] and in corpus-based natural language processing [9]. Suffix indexes can be used in many applications [10].

A suffix tree for a text of length n over an alphabet $\sum$ is of size $O(N)$ (where N is the text size) and can be built in $O(N\log|\sum|)$. Suffix tree enables us to find the longest substring of a text that matches the query string in $O(M)$ time, where M is the length of the pattern string. A problem of the suffix tree is its size and $\sum$ can be quite large for many applications. The suffix array is the most compact and simple among the suffix indexes mentioned above. The construction and searching time of the suffix array does not depend on the size of alphabet. The searching time is competitive with the suffix tree's in practice.

There are a number of different distributed multilevel data structures that have been investigated for multikey searching and sorting on both conventional and distributed computers [11] [12]. But they have shortcomings and drawbacks dealing with string problems. MSA (Multidimensional Suffix Arrays) [13] is a new data structure for string search which combines the features of suffix arrays and B-Trees, and is very amenable to implementation on both conventional and clustered computers. But MSA does not consider about the situation that access pattern follows the "90/10" rule. That means we should pay the same cost when we find something that has been just found.

This paper focuses on the performance of string search using suffix array when the access pattern follows the "90/10" rule. In this paper, we propose a new structure, the splay suffix arrays, for investigating and refining high-performance algorithms for searching strings on both conventional computers and clusters.

The paper is organized as follows. Section II introduces the related work of string search using suffix array. Section III gives the problem statement. Then we describe the splay suffix array (SSA) and distributed splay suffix array (DSSA) and their algorithms in section IV. Section V shows the experimental results, and section VI is the conclusion.

II. Related work

A suffix array is a linear structure composed of pointers to every suffix in the text (since the user normally bases his queries upon words and phrases, it is customary, in documents, to index only word beginnings). These index pointers are sorted according to a lexicographical ordering of their respective suffixes. To find patterns in the text, binary search is performed

Manuscript received November 14, 2005.

TU 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: tukun@ustc.edu).
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).
BI Kun is with the Department of Computer science and technology, University of Science and Technology of China, (e-mail: bikun@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).
DONG Wan-li is with the Department of Computer science and technology, University of Science and Technology of China, (e-mail: danli@mail.ustc.edu.cn)

on the array at $O(\log N)$ cost. Suffix array of string S, denoted SA, is an array of size n storing the sorted order of suffixes of text, i. e., SA[i] is j iff suff_j is the i^{th} lexicographically smallest suffix. A closely related concept is the longest common prefix (lcp) which is an array of some length often coupled with suffix arrays. One important information about lcp is the length of lcp of adjacent suffixes in the suffix arrays.

The simplest approach to build the suffix array sequentially is to perform a traditional sort of the pointers, such as mergesort of quicksort. However, there exist specialized algorithms for sequential construction of suffix arrays, such as the original one of Manber and Myers [7] and, more recently, those of Sadakane [14]. MSD radix sort [15] and Multikey Quicksort [16] are known as the fastest algorithms for sorting strings lexicographically. Many distributed algorithms for this problem are generalizations of general purpose sorting algorithms adapted to suffix arrays: mergesort(Msort [17])、quicksort (Qsort [18] and G-Qsort [19]) and of MMsort [20] have been used. All above algorithms paid main attention to the construction of suffix array. Different from them, MSA (Multidimensional Suffix Arrays) [13] proposed a new data structure for investigating and refining high-performance algorithms for searching strings on both conventional computers and clusters. The main drawback of MSA is that we should pay the same cost when we find something that has been just found, especially when the access pattern follows the "90/10" rule.

We combine the features of splay trees and suffix arrays into a new data structure. The purpose of this is to improve the performance of suffix arrays when the access pattern follows the"90/10" rule and reduce the amount of pointer manipulations for efficiently processing at the same time.

III. Problem Statement

Let S=a_0, a_1, …,a_{N-1} be a text of length N. Denote the substring a_i, a_{i+1}, …, a_j by SA[i,j]. Each a_i is a member of the finite alphabet ∑. Denote the suffix that starts at position i in the text S by S_i =SA[i,N]= a_i, a_{i+1}, …, a_{N-1} . The number i is called index of the suffix S_i .The suffix array SA built on S is an array of length N storing the sequence of indexes p_0,p_1,…,p_{N-1} such that S p_0< S p_1<…< S p_{N-1}, where "<" denotes the lexicographic order.

Now assume that we will perform a long series of $T > N$ operations on a suffix array. We concern about the performance of suffix array when the access pattern follows the "90/10" rule. That means we often find something that has been just found .

IV. Distributed Splay Suffix ARRAYS (DSSA)

A. Splay tree

A splay tree [21] is a variety of self-adjusting binary search tree. Static binary search trees either do not adjust in response to changing balance or access patterns, or if they do (as in AVL trees) , information is copiously maintained at each node to maintain balance. Sleator and Tarjan [21] devised splay trees to allow a binary tree to self-adjust in response to varying access patterns and yet remain approximately balanced without storing additional balance information. Then, instead of providing a firm $O(\log N)$ time guarantee for each operation, the amortized time is $O(\log N)$ while some individual operations may be more expensive. The fundamental heuristic used in splay trees to accomplish this task is called SPLAYING. SPLAYING moves the currently accessed node to the root of the tree through a series of rotations while keeping the tree roughly balanced during this move. Thus, SPLAYING can re-balance a tree as well as reduce the amortized cost of accessing nodes by keeping frequently accessed nodes near the root of the tree.

B. SSA and SSA algorithms

Different from standard splay trees, each node in splay suffix array contains a value to point out the start position of the correspond suffixes in suffix arrays. So we can apply Manber and Myer's string searching method [7] firstly to the suffixes which are often accessed. We show the splay suffix arrays of text "BANANA" in Fig1.

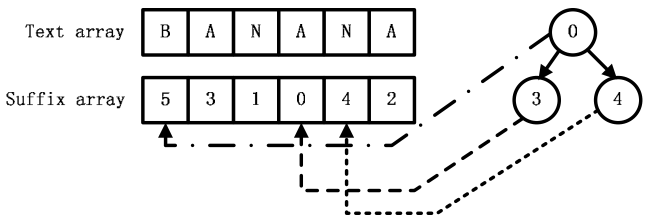

Fig 1. Splay Suffix Arrays (SSA)

1) The construction of SSA

Compare with suffix array, splay suffix array takes advantage of splay tree to improve the performance of string searching. The cost is the operation of SPLAYING and the space to store the splay tree. The following lemma captures the essence of how we construct the splay suffix array.

Lemma 1: Let P(x, M, N) be the performance of string searching using data structure x., then

$$P(SSA, M, N) \le P(suffixarray, M, N),$$

$$\text{if } \sum_{i=1}^{T} \log(N_i * R) \le \sum_{i=1}^{T} \log N_i$$

Where M is the size of pattern string, and N is the size of text, R is the number of nodes of splay suffix array, and N_i is the number of the suffixes that belong to the node R_i of splay suffix arrays.

Proof: From [21] and [7], we know that the max complexity of T operations of splay tree which has R nodes is $O(T \log R)$,and the complexity of string searching of suffix array is $O(M + \log N)$. Then it is easy to see:

$$P(SSA, M, N) = T \log R + T * M + \sum_{i=1}^{T} \log N_i$$

$$= T * M + \sum_{i=1}^{T} \log(N_i * R)$$

$$P(suffixarray, M, N) = T * M + T \log N$$

$$= T * M + \sum_{i=1}^{T} \log N$$

$\because \sum_{i=1}^{T} \log(N_i * R) \leq \sum_{i=1}^{T} \log N$

$\therefore P(SSA, M, N) \leq P(suffixarray, M, N)$ ■

Now we are ready to describe the algorithm of the construction of SSA. The key point here is $\sum_{i=1}^{T} \log(N_i * R) \leq \sum_{i=1}^{T} \log N_i$. That means to keep $N_i \leqslant$ [N/R] when the correspond node in SSA is deep, and let the nodes that have more than [N/R] suffixes to be near the root.

So when we construct the SSA, we perform the following steps:

a) Divide all suffixes into different buckets according to their first three characters. We first propose that strings be processed on a character by character for the first three characters. The reason for this splitting after the first three characters is that many common words in the English dictionary contain the same first three characters [13]. For example, "then", "theory", "theatre". So we can divide the text into different buckets, suffixes in the same bucket have same first three characters. We can put one or more buckets into a node. On the other hand, if there are too many suffixes in one bucket, we can divide it into several nodes. Then we can apply Manber and Myer's string sorting method [7] to each node to finish the construction of splay suffix array. The complexity will be $O(N)$.
b) Let every bucket be a node of SSA.The complexity will be $O(R)$.
c) Use Manber and Myer's string sorting method [7] to finish the construction of suffix array in each node. The complexity will be $O(\sum_{i=1}^{T} N_i)$.

2) String searching

Let $f(s)$ denote the first three characters of string s, $g(R_i)$ denote the first suffix of the suffixes that belong to the node R_i of SSA, where R_0 denotes the root node. Let m denote the pattern string. Then we can perform the following steps:

a) If $f(m) = f(g(R_0))$,then use Manber and Myer's string searching method, the complexity will be $O(\log N_0 + M)$;else do b).
b) If $f(m) < f(g(R_i))$,then compare f(m) and $f(g(R_i.lchild))$;else compare f(m) and $f(g(R_i.rchild))$.
c) Do this recursively till we find the node R_i that $f(m) = f(g(R_i))$ and then do d), or if there is no such node then we can say that no match position. The complexity will be $O(\log R)$.
d) Use Manber and Myer's string searching method, the complexity will be $O(\log N_i + M)$. Adjust the structure; the complexity will be $O(\log R)$

C. DSSA and DSSA algorithms

Our distributed model is that of a cluster. Assume that we have a number p of computers, we call each one computing node. A problem in the implementation of DSSA is how to reduce the amount of pointer manipulations and communication complexity for efficiently processing. In order to resolve this we change the structure of SSA a little.

If there are too many nodes in SSA, then the cost will be high when there is only one computing node to maintain all of SSA nodes. So we choose some of computing nodes to maintain the nodes of SSA. We just keep 3~5 SSA nodes in one computing node, and then the SPLAYING can be operated efficiently in it. We call these computing nodes "SSA-keeping " nodes. Each leaf node in "SSA-keeping" node keeps a value to point another "SSA-keeping" node. The relationships between these "SSA-keeping" nodes are same as the relationships between the nodes of splay tree. We choose one computing node, called Home node, to maintain the splay tree that represents the relationship between the "SSA-keeping" nodes. We show the DSSA of a cluster which has 10 nodes in Fig .2a and Fig .2b.

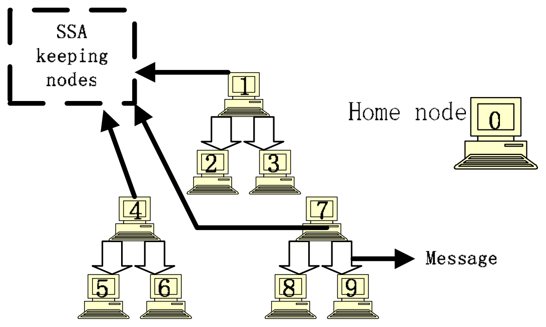

Fig .2(a) The real structure of the cluster

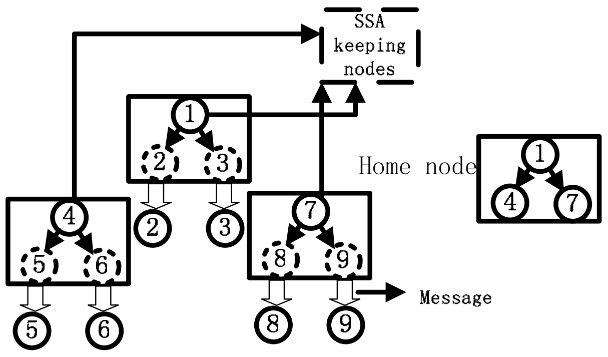

Fig. 2 (b) The correspond logical structure of DSSA of the cluster in Fig. 2 (a)

We keep the structure of SSA in the SSA keeping nodes (node 1, 4, 7) instead of keeping the structure in all of the computing nodes. And we maintain the splay tree that represents the relationship of SSA keeping nodes in the Home node (node 0). Now only Home node and "SSA-keeping" nodes need to communicate when the pattern string is not found, and each node needs to communicate no more than two "SSA-keeping" nodes.

1) The construction of DSSA

Let R be the number of SSA keeping nodes and P denote the number of computing nodes. It's clear that $R \le P$. Then we can perform the following steps:

a) Divide all suffixes into different buckets according to their first three characters. The complexity will be $O(N)$. Send each bucket to the correspond computing node. The communication complexity will be $O(P)$.

b) Let every bucket be a node of SSA and choose some of the computing nodes to be "SSA keeping node". Construct the splay tree that represents the relationship between the "SSA-keeping" nodes in Home node. The complexity will be $O(R)$.

c) Use Manber and Myer's string sorting method [7] to finish the construction of suffix array in each node. Because every node can do this parallel, the complexity will be O(logN$_{max}$), where N$_{max}$=max (N$_i$, 1≤i≤P).

2) String searching

a) Find the "SSA keeping" node Ri in Home node R0 that $f(m) \le f(g(R_i))$, and adjust the structure of splay tree that R0 keeps. The complexity will be $O(T \log R)$, if we do this operation T times.

b) Let $R_i.lSSA$ denote the left SSA keeping node of Ri and $R_i.rSSA$ denote the right SSA keeping node of Ri. If $f(m) < f(g(R_i))$,then send message to $R_i.lSSA$, and compare $f(m)$ and $f(g(R_i.lSSA))$;else send message to $R_i.rSSA$,compare $f(m)$ and $f(g(R_i.rSSA))$. Do this recursively till we find the node Ri that $f(m) = f(g(R_i))$ and then do **c)**, or if there is no such node then we can say that no match position. The complexity will be $O(\log R)$. And the communication complexity will be $O(\log R)$.

c) Use Manber and Myer's string searching method, the complexity will be $O(\log N_i + M)$. Adjust the structure; the complexity will be $O(\log R)$.

We show the process of string searching in Fig 3. Assume that the string we find is in node 5. Firstly, we find the right SSA keeping node in Home node. In this case, the word "right" means that we can find the clue of node 5 in it. And then adjust the structure of splay tree in Home node. Secondly, we search the SSA keeping node continuously till we find the node 5. Finally, use Manber and Myer's string searching method to find the string. And then we adjust the structure.

V. Experimental Results

In this section, we show the performance of DSSA. We implemented DSSA using MPI. The programs were executed on HP RX2600 cluster. Fig 5 shows the relationship between the SSA and the access patterns which are shown in Fig 4. The size of text string is 16kB, and the pattern string's is 8kB. From Fig.4 we know that when the access pattern follows the "90/10" rule, we access the strings that begin with "the" more often than we access the strings that begin with "acc".

It's clear in Fig.5 that the DSSA is inefficient when we find something randomly. The reason is that we should change the structure of DSSA when we access it. But it is amenable to improvement when the access pattern follows the "90/10" rule. Because it takes advantage of the fact that we often find something that we have just found.

We compare the string searching time between the DSSA and one of the distributed versions of suffix array [20]. The results presented in Fig 6 show the fact that DSSA is very suitable for the situation that we often find something that we have just found.

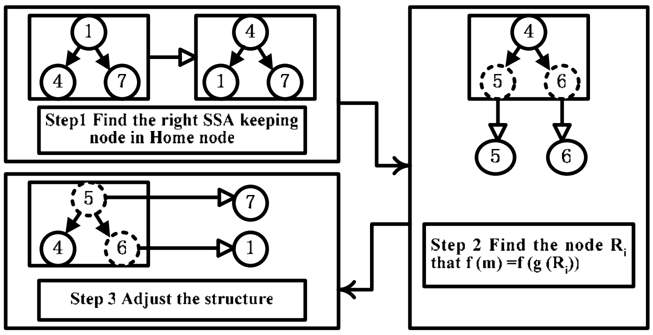
Step1 Find the right SSA keeping node in Home node
Step 2 Find the node R$_i$ that f (m) =f (g (R$_i$))
Step 3 Adjust the structure

Fig. 3 The process of string searching

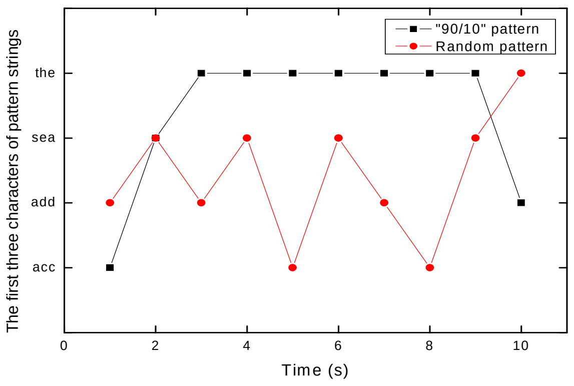

Fig. 4 Two different access patterns

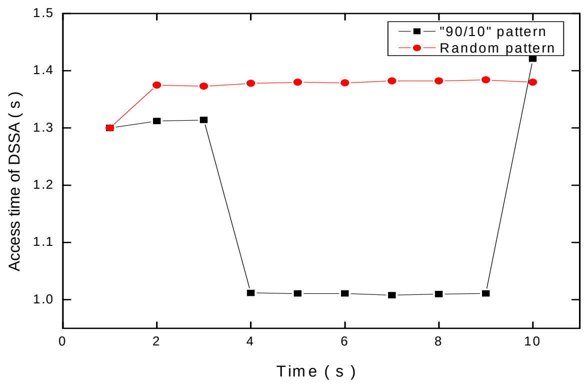

Fig. 5 The relationship between DSSA and access pattern

VI. CONCLUSION

We propose a new structure for string searching. The structure, called Splay Suffix Array (SSA), combines the features of splay trees and suffix arrays into a new data structure. The purpose of this is to improve the performance of suffix arrays when the access pattern follows the"90/10" rule and reduce the amount of pointer manipulations for efficiently processing. DSSA offer substantial advantages over both string splay trees and suffix arrays in terms of memory space and time.

The complexity of DSSA is very amenable to performance improvement through several parameters, including efficient implementation of DSSA, exploitation of parallelism, and the high-performance capabilities of computer cluster architecture.

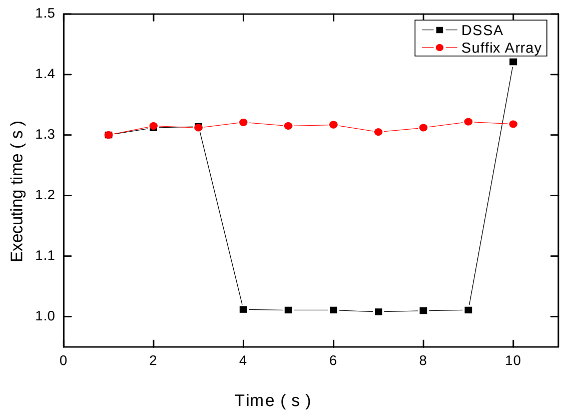

Fig. 6 Comparison between DSSA and Suffix Array

REFERENCES

[1] I. H. Written, A. Moffat, and T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Van Nostrand Reinhold, 1994.
[2] W. B. Cavnar. Using an n-gram based document representation with a vector processing retrieval model. In Proc. Of the 3rd Text Retrieval Conference (TREC-3), pages 269-277. NIST special publication, 1995.
[3] G. A. Stephen. String Searching Algorithms. World Scientific Publishing, 1994.
[4] M. Crochemore and W. Rytter. Text Algorithms. Oxford Univ. Press, New York, 1994
[5] D. Gusfield. Algorithms on Strings, Trees, and Sequences. Cambridge Univ. Press, 1997
[6] E. M. McCreighl. A Space-Economical Suffix Tree Construction Algorithm. Journal of the ACM, 23, pp. 262-272, 1976.
[7] U. Manber and G. Myers Suffix Arrays: A New Method for On-Line String Searches. SIAM J. Comput. 22(5), pp 935-948, 1993.
[8] G. H. Gonnet, R. A. Baeza-Yates, et al. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. A. Baeza-Yates, editors, Information Retrieval: Data Structure and Algorithms, pages 66-82. Prentice-Hall, New Jersey, 1992.
[9] M. Nagao and S. Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrase form large text data of Japanese. In Proc of COLING'94, pages 611-615,1994.
[10] A. Apostolico. The myriad virtues of subword trees. In Combinatorial Algorithms on Words, pages 85-96. Springer-Verlag, 1985.
[11] A. Fellah. Concurrent and Distributed Data Structures for Multikeys Sorting on Computer Clusters. IEEE Proc. Of the 16th Intern. Symp. On High Performance Comput. Systems and Applications. Pp. 281-, Moncton, Canada, 2002.
[12] A. Fellah, A. Maamir and M. Abaza. Distributed Data Structures for Multikey Sorting. Intern. J. of Parallel and Distributed Systems and Networks, Vol. 2(2), pp. 62-68, 1999.
[13] Fellah, A. and Mawson, R. Distributed multidimensional suffix arrays for string search. Communications, Computers and signal Processing, 2003. PACRIM. 2003 IEEE Pacific Rim Conference on , Volume: 2, pp. 792-795, 2003.
[14] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In Proc. DCC'98, pages 129-138, 1998.
[15] D. E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973.
[16] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In the 8th Annual ACM-SIAM Sympo. On Descrete Algorithms, pages 360-369, 1997.
[17] J. Kitajima, G. Navarro, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays: a quicksort-based approach. In Proc. WSP'97, pages 53-69. Carleton University Press, 1997.
[18] J. Kitajima, B. Ribeiro, and N. Ziviani. Network and memory analysis in distributed parallel generation of PAT arrays. In Proc. 8th Brazilian Symp. On Comp. Arch. – High-Performance Processing, pages 193-202. Brazilian Comp. Soc., 1996.
[19] G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In Proc. CPM'97, LNCS 1264, pages 102-115, 1997.
[20] Kitajima, J.P., Navarro, G. A fast distributed suffix array generation algorithm. String Processing and
[21] Information Retrieval Symposium,1999 and International Workshop on Groupware, Sept,1999,22-24 Pages:97-104
[22] D. D. Sleator and R. E. Tarjan. Self-adjusting Binary Search Trees. Journal of the ACM 32, pp. 652-686, 1985.

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.
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
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.
Liu Gang was born in 1978. He is a Ph.D. student in the Department of Computer Science and Technology, USTC. His research interests include interprocessor communication and mobile computing.
Dong Wan li 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