NEW ALGORITHMS FOR BALANCING BINARY SEARCH TREES

E. Haq*, Y. Cheng and S. S. Iyengar***

*Department of Computer Science, ** Department of Mathematics
Louisiana State University, Baton Rouge, Louisiana 70803

Abstract

A simple iterative algorithm is presented for balancing an arbitrary binary search tree in linear time. An efficient parallel algorithm is developed from the iterative algorithm using shared memory model. The number of processors used is equal to N, the number of nodes in the tree. The proposed algorithm has time complexity of $O(1)$.

I. Introduction

Binary search trees provide an efficient method of data storage and organization. If a binary search tree with N nodes is completely balanced, then a delete or an insertion operation on such a tree can be performed in $O(\log N)$ time. If the tree is highly unbalanced, this time may be $O(N)$. There are two approaches of balancing binary trees, one during each operation[4,5] on the tree and the other to balance the tree periodically [1,2,3]. Sequential and parallel algorithms for balancing binary search tree (BST) have been developed in [2,3]. The iterative algorithm developed in [3], uses two separate algorithms for balancing binary search trees with nodes $N = 2^n - 1$ and $N < 2^n - 1$, where n is the number of levels of the tree. We propose a new algorithm for balancing BST, which trades space for computational time than that in [3]. We first present a sequential iterative algorithm for balancing a BST and then we convert the sequential algorithm into a parallel algorithm.

We use shared memory model (SMM) for the parallel algorithm. Each processing element (PE) is capable of performing all the standard arithmetic and logical operations. Moreover, there is a common memory which is shared by all the processors. All the PE's can read and write into this memory, but care should be taken to avoid read - write conflict. The PE's are synchronized and they operate under the control of a single instruction.

The paper is organized as follows. In section II we introduce the iterative algorithm for balancing BST. In section III, we convert the sequential algorithm to a parallel algorithm. Conclusions are given in section IV.

II. Iterative algorithm for balancing BST

The task of balancing a binary search tree is to adjust the left and right pointers of the nodes in the tree, in such a way that the height of the tree is minimized. In a balanced tree, with N nodes, the search time will be $O(\lceil \log N \rceil)$ as opposed to $O(N)$ in a highly unbalanced BST. In this paper all logarithms are on base 2. Algorithms that dynamically balance tree structure during insertion or deletion of nodes are described by Knuth [4] and Tarjan [5]. Several other recursive algorithm for balancing BST can be found in [6,7,8]. In [3], two separate algorithms were presented to balance BST with nodes, $N = 2^n - 1$ and $N \leq 2^n - 1$, where $n = \lceil \log(N+1) \rceil$. In this paper, we propose a new iterative algorithm for balancing any arbitrary BST.

The first phase of the algorithm is to traverse the tree in inorder and store the pointers in an array. We can perform this traversal by a recursive routine. If we have $N = 2^n - 1$ nodes in a tree, then we get a balanced BST with n levels. But if we have $N < 2^n - 1$ nodes in the tree, then the tree will not be completely balanced. However, the missing nodes will be at the highest level (n-th level) and the tree will be completely balanced upto the $(n-1)$-th level. The number of missing nodes at the n-th level is given by

$$S = (2^n - 1) - N \tag{1}$$

Moreover, if we number the nodes of a completely balanced

88CH2571-8/88/0000-0378$1.00©1988IEEE

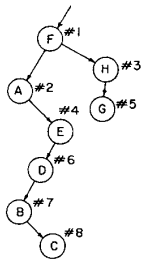
BST according to the inorder traversal of the tree, then the nodes at the highest level are numbereed with odd numbers $1, 3, 5...$

If we have $N = 2^n - 1$ nodes in the original tree, then we just traverse the tree in inorder and store the pointers in an array. In this case the length of the array will be N. But if we have $N < 2^n - 1$ nodes in the original tree, then we use an array of length $(S + N)$. We also note that the missing nodes in a balanced tree are at the highest level and are numbered with odd numbers. Thus when we traverse the tree, we need to skip the positions corresponding to the numbers assigned to the missing nodes in the array. This can be done as follows.

Let node i be marked l by inorder traversal. If NL is the number of nodes at the n-th level (leaves), then,

$$NL = (N - (2^{n-1} - 1)) \tag{2}$$

If $l \leq 2NL$ then we use the l-th position in the array to store pointers of the i-th node, otherwise we place the pointers of the i-th node at $2(l - NL)$-th position of the array. Positions corresponding to the missing nodes are marked NULL in the array. Algorithm A1 recursively traverses the tree in inorder and stores the pointers in the array $LINK$. An example of inorder traversal when $N < 2^n - 1$ is given in Figure 1.



ORDER	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ITEM	#2	#7		#8		#6		#4		#1		#5		#3	
KEY	A	B		C		D		E		F		G		H	

Figure 1. Inorder tree traversal storing the information in the array.

ALGORITHM A1: Recursively traverses the tree in inorder and stores the pointers in array $LINK$.

```
PROCEDURE TRAVERSE(T,N)
   // traverses tree in Inorder //
   DECLARE T,LINK( ),LSON( ),RSON( )
   PROCEDURE AUXTRAVERSE(T,N,NL)
   BEGIN
      IF T=NULL THEN RETURN
      AUXTRAVERSE(LSON(T),N,NL)
      N:=N+1
      IF N ≤ 2*NL THEN
         LINK(K):=T
      ELSE
         BEGIN
         K:=2(N-NL)
         LINK(K):=T
         LINK(K-1):=NULL
         LINK(K+1):=NULL
         END
      AUXTRAVERSE(RSON(T),N,NL)
   END   // end of auxtraverse //
BEGIN   // begin traverse //
   n := ⌈log(N + 1)⌉
   NL := (N - (2^(n-1) - 1))
   AUXTRAVERSE(T,0,NL)
END   // end of traverse //
```

We next develop an iterative algorithm for balancing the tree by using the information stored in the array $LINK$. In a completely balanced binary tree, the j-th node from the left on i-th level, denoted by (i, j) has $2^{n-i} + (j-1) * 2^{n-i+1}$ nodes of key value less than or equal to it. It is also observed that in such a tree, if the (i, j)-th node, $i < n$, has (i_1, j_1) and (i_1, j_2) as leftson and rightson respectively, then

$$K_1 = KVLE(i_1, j_1) = K - 2^{n-i-1} \tag{3}$$

$$K_2 = KVLE(i_1, j_2) = K + 2^{n-i-1} \tag{4}$$

where,

$$K = 2^{n-i} + (j-1) * 2^{n-i+1} \tag{5}$$

and $KVLE(i, j)$ denotes number of nodes with key value less than or equal to the j-th node from the left on i-th level.

Therefore for balancing a BST, the node corresponding to the K-th cell in the array $LINK$ will be (i, j)-th

node. Moreover, if $i < n$ then it will have nodes corresponding to the K_1-th and K_2-th cells in the array $LINK$ as its leftson and rightson, as determined by equations (3) and (4), respectively. Furthermore, if K is odd, then, it must be a leaf and therefore, its leftson and rightson will be NULL. Algorithm A2 iteratively balances any arbitrary BST.

This algorithm is similar to that in [3] for balancing BST with nodes, $N = 2^n - 1$. The way we use the array $LINK$, algorithm A2, will balance any arbitrary BST. This algorithm is simple compared to the algorithm given in [3] for balancing BST with nodes $N < 2^n - 1$. However, our algorithm will require extra spaces for the array $LINK$. The amount of extra space for the array $LINK$ is equal to the number of missing nodes (S) for a completely balanced BST and this extra space may be at most $O(N)$. However, in the proposed algorithm, we trade space for less computational time than that in [3]. For balancing tree with nodes $N < 2^n - 1$, the alogrithm in [3] requires,

ALGORITHM A2 : Iteratively balances any arbitrary BST

```
PROCEDURE BALANCE(n,NL)
    // uses information stored in array LINK
    and iteratively balances the tree //
    // n = [logN + 1] //
    // NL = (N - (2^(n-1) - 1)) = leaves at n-th level //
    INTEGER I,J,KK,K1,K2,K
    BEGIN
    FOR I:=1 TO n DO   // for each level do //
        BEGIN
        IF (I=n ) AND (NL ≠ 0) THEN KK:=NL
        ELSE
            KK:=2^(I-1)
        FOR J:=1 TO KK DO   //for all nodes at a level do //
            BEGIN
            K := 2^(n-I) + (J - 1) * 2^(n-I+1)
            IF ODD(K) THEN   // Leaf //
                LSON(LINK(K)):=NULL
                RSON(LINK(K)):=NULL
            ELSE   // interior nodes //
                BEGIN
                K1 := K - 2^(n-I-1)
                K2 := K + 2^(n-I-1)
                LSON(LINKK)):=LINK(K1)
                RSON(LINK(K)):=LINK(K2)
                END
            END   // end of inner for loop //
        END   // end of outer for loop //
    END   // end of procedure Balance //
```

6 comparisons for each node to be placed not at the highest level and 2 comparisons for each node to be placed at the highest level of the final balanced tree. On the other hand, to determine the final position of any node in the balanced tree, the proposed algorithm requires only 2 comparisons. Therefore, even though both algorithms have the same time complexity $O(N)$, our algorithm will require less computational time.

III. Parallel algorithm for balancing BST

We propose two parallel algorithms for balancing any arbitrary BST in constant time. In the first version of the parallel algorithm, we use $2^n - 1$ processors, where $n = \lceil \log(N+1) \rceil$. If the tree is highly unbalanced, then we shall be using more processors than the number of nodes. The number of extra processors is given by equation (1). However, only one processor per node will be active. Processors are numbered 1 to 2^{n-1}. The processors assigned to the missing nodes will not be active and these processors are at the highest level and are numbered with odd numbers. In the second version of the parallel algorithm, we use N processors, where N is the number of nodes in the tree.

First we describe the parallel algorithm for inorder tree traversal. As in [3], we now assume an extra field 'KNUM' with each node of the tree. The value of this field for node i, $KNUM(i)$, gives the number of nodes in the BST with key value less than or equal to the key value associated with node i. With each node of the BST we attach a processor. Now, let node i has $KNUM(i) = K$. If $K \leq 2NL$, where NL is given by equation (2), then the node i will be linked to the cell $LINK(K)$; otherwise it will be linked to the cell $LINK(2(K - NL))$. Algorithm A3 performs parallel inorder traversal of the tree and it has a constant time complexity.

Now we can convert algorithm A2 into a parallel algorithm. We allocate one processor per cell in the array $LINK$ and each processor should be able to set up its own links to construct a balanced binary search tree in constant time. Now with each processor P_K we store the height (h) of node K in the balanced BST. If we pass the total number of nodes in the tree as a parameter to the processors, then each processor will be able to find the final level i , where $i = n - h$, of the node it is processing,in the balanced tree . Each processor P_K will set up the links

to the left and right sons of node K, using equations (3), (4) and the array $LINK$ in constant time.

Algorithm A4 balances any arbitrary BST in parallel by executing all the processors simultaneously. This algorithm is very simple compared to that in [3] and is sufficient for handling both the cases ($N = 2^n - 1$ and $N < 2^n - 1$) of balancing binary search tree. It has a constant time complexity.

ALGORITHM A3 : Parallel Traversal Algorithm by Processors $P_1, P_2, ...P_{2^n-1}$; One processor for each node of the tree.

```
PROCEDURE PTRAVERSE(NL)
    DECLARE K,LINK( ),KNUM( )
    FOR EACH PROCESSOR PK DO IN PARALLEL
        BEGIN
        IF KNUM(K) ≤ 2NL THEN
            LINK(KNUM(K)):=K
        ELSE
            BEGIN
            N:=2(KNUM(K)-NL)
            LINK(N):=K
            LINK(N-1):=NULL
            LINK(N+1):=NULL
            END  //end of else //
        END  // end of for clause //
    END  // end of procedure Ptraverse //
```

ALGORITHM A4: Parallel algorithm for balancing arbitary BST using $2^n - 1$ processors, $n = \lceil \log(N+1) \rceil$.

```
PROCEDURE PBALANCE1
    // parallel algorithm for balancing BST //
    DECLARE N,LINK( ),LSON( ),RSON( )
    FOR EACH PROCESSOR P_K DO IN PARALLEL
        DECLARE n,H,j,K
        CONSTANT H,j
        n:=⌈log(N + 1)⌉
        I:=n-H
        IF ODD(K) AND (LINK(K)≠NULL ) THEN  // leaf //
            BEGIN
            LSON(LINK(K)):=NULL
            RSON(LINK(K)):=NULL
            END
        IF EVEN(K) THEN  // interior nodes //
            BEGIN
            LSON(LINK(K)):=LINK (K - 2^(n-I-1))
            RSON(LINK(K)):=LINK (K + 2^(n-I-1))
            END
        END  // end of the for loop //
    END  // end of Pbalance1 //
```

Algorithm A5 balances any arbitray BST in constant time using N processors, where N is equal to the number of nodes in the tree. Instead of keeping the information of the height of the nodes in the balanced tree with the processors, a separate array is used to keep this information. In the algorithm, this array is denoted by H. The length of this array should be $2^n - 1$, where $n = \lceil \log(N+1) \rceil$. With this information available to us, we can use N processors to balance any arbitrary BST. Each processor will access one cell of the array $LINK$. For each processor, P_K, we calculate the height for the node it is processing. If $K > 2NL$, then we calculate a number, $M = 2(K - NL)$, otherwise $M = K$. Now the processor P_K will access the M-th cell of the array $LINK$. It will fetch the height information from the M-th cell of the array H. Thus each processor will be able to calculate the final position of the leftson and the rightson of the node it is processing, in the balanced tree and accordingly restructures the pointers in constant time.

ALGORITHM A5 : Parallel algorithm for balancing any arbitrary BST using N processors.

```
PROCEDURE PBALANCE2
    DECLARE M,N,LINK( ),LSON( ),RSON( )
    FOR EACH PROCESSOR P_K DO IN PARALLEL
    DECLARE n,K,H( ),L
    IF K > 2NL THEN
        M:=K+(K-2NL)
    ELSE
        M:=K
    IF ODD(M) THEN  // leaves //
        BEGIN
        LSON(LINK(K)):=NULL
        RSON(LINK(K)):=NULL
        END
    ELSE IF EVEN(M) THEN
        BEGIN
        I:=n-H(M)
        L = 2^(n-I-1)
        LSON(LINK(M)):=LINK(M-L)
        RSON(LINK(M)):=LINK(M+L)
        END
    END  // end for //
END  // end of Pbalance2 //
```

IV. Conclusions

In this paper, we have presented new algorithms for balancing any arbitrary binary search tree with nodes $N = 2^n - 1$ or $N < 2^n - 1$, in sequential and in parallel modes. The time complexity of the parallel algorithm is $O(1)$ using N processors. This algorithm uses $O(N + S)$ space , where S is the number of missing nodes of the completely balanced tree at the highest level. Proposed algorithms are simple and they require less comparisons to determine the final position of a node in the balanced tree.

References

[1] S. S. Iyengar and H. Chang, "Efficient Algorithms to create and maintain Balanced and Threaded Binary Search Trees", Software-Practice and Experience, vol 15 (10), 925-941, Oct. 1985.

[2] A. Moitra and S. S. Iyengar, "A maximally parallel Balancing Algorithm for obtaining complete Balanced Binary Trees", IEEE Trans. on Computers, Vol. C-34, No. 6, pp. 563 -565, June 1985.

[3] A. Moitra and S. S. Iyengar, "Derivation of a Parallel Algorithm for Balancing Binary Trees," IEEE Trans. on Software Engineering, vol. SE-12, No. 3, pp. 442 - 449, March 1986.

[4] D. E. Knuth, The art of computer programming, vol 3: sorting and searching, Addison-Wesley, 1973.

[5] R. E. Tarjan, Data Structure and Network algorithm, Soc. for Ind. and Appl. Math., Philadelphia, 1987.

[6] A. C. Day, "Balancing a binary tree", Computer Journal, 19, 4 (Nov 1978), 360-361.

[7] W. A. Martin and D. N. Ness, "Optimal binary trees grown with a sorting algorithm", Commun. ACM 15,2 (Feb 1972), 88-93.

[8] F. S. Quentin and L. W. Bette, "Tree Balancing in optimal time and space", Commun. ACM 29, 9 (Sep. 1986), 902-908.

New Address of Dr. Y. Cheng

AT & T Bell Laboratories
Crawfords Corner Road
Holmdel, New Jersey 07733