A Maximally Parallel Balancing Algorithm for Obtaining Complete Balanced Binary Trees

ABHA MOITRA AND S. SITHARAMA IYENGAR

Abstract — **We present a new iterative balancing algorithm for binary trees of size $N = 2^n - 1$ by exploiting the similarity of pointer restructuring at each level. We also extract parallelism from this algorithm to yield a constant time complexity balancing algorithm for an N-processor configuration. This achieves the theoretical limit of speedup possible.**

Index Terms — **Balancing binary trees, binary search trees, complete binary trees, parallel algorithm.**

I. Introduction

Binary search trees provide a method of data organization which is both flexible and efficient. Furthermore, records are retrieved from binary search trees in an average time proportional to $\log N$ where N is the size number of nodes of the tree. The task of balancing a binary search tree is to adjust the left and right pointers of all the nodes in the tree so that the search path is optimized. For an overview on algorithms for balancing binary search trees see [1]–[6] and the references therein.

Recently Chang and Iyengar [2] presented an algorithm (hereafter referred to as the C–I algorithm) to balance binary search trees in $O(N)$ time. This algorithm can be summarized as follows.

Phase 1: Traverse the tree in Inorder, storing pointers in the array LINK.

Phase 2: Procedure GROW(LOW:HIGH) recursively constructs a balanced binary search tree for the nodes LINK(LOW:HIGH).

The interesting aspect of the C–I algorithm is that the left and right subtrees rooted at the root are grown simultaneously. In particular, if the root is linked by the Mth cell in the array link, then, for all $K < M$, if LINK($K + M$) has LINK($j1$) and LINK($j2$) as its LSON and RSON, then LINK($K + M$) has LINK($j1 + M$) and LINK($j2 + M$) as its LSON and RSON. This is referred to as the "folding" method with a folding factor M. Further details can be found in [2].

Throughout this correspondence we assume that the binary tree to be balanced is of size $N = 2^n - 1$. We present a new iterative balancing algorithm which exploits the similarity of pointer restructuring required for all the nodes at the same level, and this algorithm has a time complexity of $O(N)$. This is a generalization of the *folding* method [2] and is referred to as "level restructuring." More significantly, we describe how parallelism can be extracted from this new iterative algorithm to yield a constant time complexity algorithm. This work can be extended as in [7] to extract maximum parallelism for balancing arbitrary sized binary trees.

II. Balancing Binary Search Trees Through Level Restructuring

We now show that the pointer restructuring for all the nodes on the same level is similar.

Definition 1: In a binary tree T, the $\langle i,j\rangle$th node refers to the jth node from the left on the ith level, if both exist. □

Definition 2: In a complete balanced binary tree T of N nodes let $\text{CVAL}_N(\langle i,j\rangle)$ denote the number of nodes with data values less than or equal to the data value of the $\langle i,j\rangle$th node. □

Fact 1: Consider a complete balanced binary tree T with $N = 2^n - 1$ nodes. If the $\langle i,j\rangle$th node exists in T then $\text{CVAL}_N(\langle i,j\rangle) = 2^{n-i} + (j-1) * 2^{n-i+1}$.

Proof: Let us consider a complete balanced binary tree structure T with its nodes labeled by Inorder traversal. The label associated with each node corresponds to the number of nodes with data value less than or equal to its own data value. Thus, the $\langle i, 1\rangle$th node (if it exists) is labeled 2^{n-i} by the Inorder traversal. Further, if node $\langle i,k\rangle$ is labeled y, then the node on the same level to its immediate right (if it exists) is labeled $y + 2^{n-i+1}$. It then follows that if the $\langle i,j\rangle$th node exists, $\text{CVAL}_N(\langle i,j\rangle) = 2^{n-i} + (j-1) * 2^{n-i+1}$. □

Fact 2: Consider a complete balanced binary tree T with $N = 2^n - 1$ nodes: if the $\langle i,j\rangle$th node, $i < n$, has nodes $\langle i1, j1\rangle$ and $\langle i2, j2\rangle$ as its LSON and RSON, respectively, then $\text{CVAL}_N(\langle i1,j1\rangle) = k - 2^{n-i-1}$ and $\text{CVAL}_N(\langle i2,j2\rangle) = k + 2^{n-i-1}$ where $k = 2^{n-i} + (j-1) * 2^{n-i+1}$.

Proof: From Fact 1 we know that $\text{CVAL}_N(\langle i,j\rangle) = 2^{n-i} + (j-1) * 2^{n-i+1}$. We also know that in a complete balanced binary tree the $\langle i,j\rangle$th node ($i < n$) has $\langle i+1, 2*j-1\rangle$th and $\langle i+1, 2*j\rangle$th nodes as its LSON and RSON, respectively. So,

$$\begin{aligned}\text{CVAL}_N(\langle i+1, 2*j-1\rangle) &= 2^{n-(i+1)} + ((2*j-1)-1) * 2^{n-(i+1)+1} \\ &= k - 2^{n-i-1}\end{aligned}$$

$$\begin{aligned}\text{CVAL}_N(\langle i+1, 2*j\rangle) &= 2^{n-(i+1)} + (2*j-1) * 2^{n-(i+1)+1} \\ &= k + 2^{n-i-1}.\end{aligned}$$

□

This means that for balancing a binary search tree with $N = 2^n - 1$ nodes, the node corresponding to the kth cell in the array LINK will be the $\langle i,j\rangle$th node as determined by the equation

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

Further, if $i < n$, then it will have nodes corresponding to the $k1$th and $k2$th cell as its LSON and RSON, as determined by the following equations:

$$k1 = k - 2^{n-i-1} \tag{2}$$

$$k2 = k + 2^{n-i-1}. \tag{3}$$

We can therefore balance a binary tree of size $N = 2^n - 1$ as follows.

Step 1: Traverse the tree in Inorder, storing the pointers in the array LINK.

Step 2: Visit the nodes in increasing level order from left to right, setting up appropriate links to construct a balanced binary search tree.

Steps 1 and 2 are accomplished by the algorithms A1 and A2, respectively.

Algorithm A1: Recursive Inorder traversal for storing the pointers in the array LINK.

```
procedure MAIN (T)                    //The tree with root T is to be
                                        traversed//
  declare T, LINK()
  procedure TRAVERSE (P, N)
    if P = null then return
    TRAVERSE (LSON(P), N)             //LSON(P) is the pointer to the
                                        left son of P//
    N := N + 1
    LINK(N) := P
    TRAVERSE (RSON(P), N)             //RSON(P) is the pointer to the
                                        right son of P//
  end TRAVERSE
  TRAVERSE (T, 0)
end MAIN
```

Manuscript received March 30, 1984; revised December 3, 1984.

A. Moitra is with the Department of Computer Science, Cornell University, Ithaca, NY 14853.

S. S. Iyengar is with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803.

0018-9340/85/0600-0563$01.00

Algorithm A2: Iterative growing algorithm where $N = 2^n - 1$. Level restructuring is done by visiting the nodes from left to right in increasing level order, setting up links to construct a balanced binary tree.

```
procedure C-LEVEL-STRUCT
  declare I, J, K, DIFF, n, N, LINK(1:N), LSON(1:N), RSON(1:N)
  for I := 1 to n do                                  //for each level//
    for J := 1 to 2^(I-1) do                          //for all nodes on a level//
      K := 2^(n-I) + (J - 1) * 2^(n-I+1)              //K = CVAL_N(<I,J>), using (1)//
      if odd(K) then                                  //leaves//
        LSON(LINK(K)) := null
        RSON(LINK(K)) := null
      endif
      if even(K) then                                 //interior nodes//
        K1 := 2^(n-I-1) + (2 * J - 2) * 2^(n-I)       //set up left link, using (2)//
        LSON(LINK(K)) := LINK(K1)
        K2 := 2^(n-I-1) + (2 * J - 1) * 2^(n-I)       //set up right link, using (3)//
        RSON(LINK(K)) := LINK(K2)
      endif
    endfor
  endfor
  end C-LEVEL-STRUCT
```

The time complexity of Algorithm A1 is $O(N)$. Algorithm A2 is constructed so that each node is visited exactly once, and hence it also has an $O(N)$ time complexity.

III. Parallel Balancing Algorithm

In this section we develop a parallel balancing algorithm for binary search trees with $N = 2^n - 1$ nodes. What we are seeking is a decomposition of the balancing problem so that the resulting algorithm can be run on an N-processor configuration. The model of parallel computation that we will be using is a simplification of the shared memory model (SMM). This model can be characterized as follows.

1) There are M processing elements (PE's) or processors. These are indexed $1, 2, \cdots, M$, and the ith PE is referenced as P_i. Each PE has the capability of performing all the standard arithmetic and logic operations.

2) There is a common memory that is shared among all the PE's. All the M PE's can read and write into this memory at any time instance. If two or more PE's attempt to read (write) from the same memory location, a read (write) conflict occurs.

In the algorithms we will develop, we will ensure that no read or write conflict occurs.

A. Parallel Traversal of Binary Tree

If N-processors are available, each node of the binary search tree can be associated with a unique processor. We assume that with every node in the input binary search tree an additional field NUM(I) gives the number of nodes with the key values less than or equal to that associated with node I. The value of this new field is altered only when entries are added or removed from the binary search tree. The value of this field can be updated as entries are added and deleted from the binary search tree without increasing the time complexity of performing these operations.

With this additional information, the traversal of the binary search tree can be accomplished simply by linking node I with LINK(NUM(I)). Furthermore, the processing associated with node I does not depend upon, or interfere with, the processing associated with any other node. This is due to the fact that all nodes are linked to unique cells in array LINK. So, all the computation associated with node I is done by processor P_I, and the traversal of the entire binary search tree can be accomplished in a constant amount of time. This leads directly to Algorithm A3.

Algorithm A3: Parallel traversal algorithm by processes $P_1, \cdots, P_N$, one for each node of the tree executing simultaneously.

```
procedure P-TRAVERSE
  declare K, LINK(1:N), NUM(1:N)
  for each PE P_K do in parallel
    LINK(NUM(K)) := K
  endfor
  end P-TRAVERSE
```

B. Parallel Algorithm for Growing Complete Balanced Binary Trees

The straightforward approach for transforming Algorithm A2 into a parallel algorithm would be to allocate one processor per cell in the array LINK and to have each processor set up its own links to construct a balanced binary tree. For this, a processor P_k associated with LINK(k) should be able to determine the following.

R1. i, j such that $\text{CVAL}_N(\langle i, j\rangle) = k$; that is its final position in the balanced binary tree.

R2. If it has nonnull LSON, then determine $k1$ such that $\text{CVAL}_N(\langle i + 1, 2 * j - 1\rangle) = k1$.

R3. If it has nonnull RSON, then determine $k2$ such that $\text{CVAL}_N(\langle i + 1, 2 * j\rangle) = k2$.

We are interested in a constant time complexity parallel algorithm, and under that restriction, we wish to determine all of the above information.

If we allocate processor P_k to process the node with CVAL equal to k, then P_k may correspond to different indexes $\langle i + H, j\rangle$, $H = 0, 1, 2, \cdots$ (note j will be constant) depending on the size of the complete balanced binary tree. For example, $\text{CVAL}_7(\langle 2, 2\rangle) = 6$, and also, $\text{CVAL}_{15}(\langle 3, 2\rangle) = 6$. But what are invariant over the size of the complete balanced binary tree are the indexes h, j where $h = n - i$. So if h, j values are permanently associated with a processor P_k and if the total number of nodes in the tree is passed as an argument/parameter, then the processor can determine its indexes $\langle i, j\rangle$, and hence requirement R1 can be met in constant time. It is this observation, by which we can assign computation to the processes in such a way that the indexes h and j are kept constant as the size of the array changes, that allows us to derive a constant time complexity parallel algorithm. (If it is not possible to associate the constants h, j permanently with a processor, then the indexes i, j can be determined for each process P_k, but not in constant time.)

The requirements R2 and R3 can be satisfied very easily in constant time since from Fact 2 we know how to calculate $\text{CVAL}_{2^{n-1}}(\langle i, j\rangle)$ for any arbitrary i and j. The Algorithm A4 follows immediately. This algorithm has a constant time complexity when run on an N-processor configuration; and it involves no possibility of read or write conflicts.

Algorithm A4: Parallel growing algorithm for the construction of a balanced binary tree by simultaneously executing processes $P_1, \cdots, P_N$, one for each cell in the array LINK, where $N = 2^n - 1$.

```
procedure PC-LEVEL-STRUCT
  declare N, LINK(1:N), LSON(1:N), RSON(1:N)
  for each PE P_K do in parallel
    declare n, H, J, K
    constant H, J
    n := ⌈log(N + 1)⌉
    I := n - H
    if odd(K) then                                  //leaves//
      LSON(LINK(K)) := null
      RSON(LINK(K)) := null
    endif
    if even(K) then                                 //interior
                                                      nodes//
      LSON(LINK(K)) := LINK(K - 2^(n-I-1))
      RSON(LINK(K)) := LINK(K + 2^(n-I-1))
    endif
  endfor
end PC-LEVEL-STRUCT
```

C. Analysis of Algorithms A3 and A4

Algorithms A3 and A4 both run in constant time independent of the size of the input tree. The performance of the algorithm can also be described by another complexity measure called EPU (effectiveness of processor utilization). This is defined as follows:

$$\text{EPU} = \frac{\text{complexity of fastest known sequential algorithm}}{\text{complexity of parallel algorithm} * \text{number of processors}}$$

$$= \frac{n}{1 * n} = 1.$$

Thus, the new parallel algorithm achieves the maximum possible speedup for this particular problem. Furthermore, if we have K ($K \leq N$) processors, then the computations can be rescheduled over these processors (processing time = N/K) with EPU still being equal to one.

IV. Conclusions

In this correspondence we have presented algorithms for balancing binary trees of size $N = 2^n - 1$. We were able to develop a maximally efficient parallel algorithm by exploiting the similarity of the pointer restructuring of all the nodes at the same level. Of course, a complete analysis should also consider the question of balancing binary search trees of arbitrary size; we do that in [7].

Acknowledgment

The authors would like to thank the referees for their comments on this correspondence.

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 509–517, Sept. 1975.

[2] H. Chang and S. S. Iyengar, "Efficient algorithm to globally balance binary search trees," *Commun. Ass. Comput. Mach.*, vol. 27, pp. 695–702, July 1984.

[3] A. C. Day, "Balancing a binary tree," *Comput. J.*, vol. 19, pp. 360–361, Nov. 1976.

[4] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Potomac, MD: Computer Science Press, 1976, pp. 442–456.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.

[6] W. A. Martin and D. N. Ness, "Optimal binary trees grown with a sorting algorithm," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 88–93, Feb. 1972.

[7] A. Moitra and S. S. Iyengar, "Derivation of a maximally parallel algorithm for balancing binary search trees," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. 84-638, Sept. 1984.

Insertion Networks

M. DAVIO AND C. RONSE

Abstract **— The problem of inserting an item in a list is frequently encountered in various application fields such as sorting, compiling, etc. It is shown that the insertion of an item in a list of N members may be realized at a cost proportional to N and within a delay proportional to $\log N$.**

Index Terms **— Computational complexity, memories, permutation networks.**

I. Introduction

The problem of inserting an item in an ordered list is frequently encountered in various fields of computer science, for example in sorting or compiling. Most frequently, insertion is considered as a sequential task, to be performed step by step on a conventional general-purpose processor. It is, however, rewarding to attempt to design a special-purpose circuit dedicated to the insertion task; such circuits will provide us with interesting information on various complexity measures of the insertion problem.

We shall consider information-lossless insertion: in this case, the problem amounts to inserting a distinguished item of the list in an arbitrary position in that list, and to shifting or relabeling accordingly the other items of the list. If the list consists of N elements numbered $0, 1, \cdots, N-1$, the problem is also equivalent to generating the N permutations $\pi_0, \pi_1, \cdots, \pi_{N-1}$ where

i) π_0 is the identity permutation;

ii) for $i = 1, \cdots, N-1$,

$$\pi_i \text{ maps } j \text{ on } j+1 \quad \text{if } j < i,$$
$$i \text{ on } 0,$$
$$\text{and fixes every } j > i. \tag{1}$$

Insertion networks are thus a particular case of networks generating a certain class of permutations. Such problems have been dealt with in many papers, especially the ones concerning the design of networks generating *all* permutations; here the asymptotic lower bounds for the component cost and delay are $O(N \log N)$ and $O(\log N)$, respectively [3]. These bounds are obtained by Benes-type networks [1], [4]. Another specialization consists of *rotator networks* [2], which generate the N rotations $\pi_k : i \rightarrow i \oplus k$ (mod N); here the asymptotic cost and delay are again $O(N \log N)$ and $O(\log N)$, respectively.

For insertion networks, standard arguments from complexity theory show that the asymptotic lower bounds for the cost and delay are $O(N)$ and $O(\log N)$.

Manuscript received January 15, 1984; revised June 5, 1984.

The authors are with Philips Research Laboratory, 2, Ave. Van Becelaere, Box 8, 1170 Brussels, Belgium.

0018-9340/85/0600-0565$01.00