Derivation of a
Maximally Parallel Algorithm for
Balancing Binary Search Trees

Abha Moitra
and
S. Sitharama Iyengar

TR 84-638
September 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*Department of Computer Science, Louisiana State University, Baton Rouge, Louisiana 70803
Derivation of a Maximally Parallel Algorithm for Balancing Binary Search Trees

Abha Moitra
Department of Computer Science
Cornell University
Ithaca, NY 14853

S. Sitharama Iyengar
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

ABSTRACT

A recent trend in program methodologies is to derive efficient parallel programs from sequential programs. This paper explores the question of transforming a sequential algorithm into an efficient parallel algorithm by considering the problem of balancing binary search trees. The derivation of the parallel algorithm makes use of stepwise refinement. We first derive a new iterative balancing algorithm that exploits the similarity of pointer restructuring required at all the nodes at the same level. From this we derive a parallel algorithm that has time complexity $O(1)$ on an $N$-processor configuration. This achieves the theoretical limit of speed-up possible in a multi-processor configuration.

1. INTRODUCTION

In recent years, researchers have sought to transform sequential programs for efficient execution on parallel machines. Such a transformation depends on the degree of parallelism sought and the model of parallel computation. The most common model of parallel computation is the shared memory model (SMM), which has the following characteristics:

1. There are $M$ processing elements (PEs) or processors. These are indexed 1, 2, ..., $M$ and the $i^{th}$ PE is referenced as $P_i$. Each PE has the capability of performing all the standard arithmetic and logic operations. Each PE also knows its own index.
2. There is a common memory that is shared among all the PEs. All PE's can read and write into this memory but if two or more PEs attempt to read from (write to) the same memory location, a read (write) conflict occurs.

3. The PEs are synchronized and operate under the control of a single instruction stream.

4. An enable/disable mask can be used to select any arbitrary subset of PEs to perform an instruction; all the other PEs remain idle. All enabled PEs execute the same instruction.

A number of parallel algorithms have been developed for this model. For example, various matrix and graph problems have been investigated by Agerwala and Lint [1], Arjomandi and Corneil [2], Csanky [5] and Hirschberg [9], sorting problems have been studied by Hirschberg [10], Muller and Preparata [13,15]. Evaluation of polynomials has been considered by Munro and Paterson [14], arithmetic expression evaluation has been studied by Brent [3]. The generation of postfix form of arithmetic expressions and scheduling algorithms have been studied by Dekel and Sahni [7,8].

A useful metric for analyzing parallel algorithms is to compute the effectiveness of processor utilization (EPU). For any problem $S$ with a parallel algorithm $T$, this is defined as follows:

$$EPU(S, T) = \frac{\text{complexity of the fastest sequential algorithm for } S}{\text{number of PEs used by } T \times \text{complexity of } T}$$

For any problem, $0 \leq EPU \leq 1$, and $EPU = 1$ corresponds to the theoretical limit of speed-up possible.

Another characteristic of a parallel algorithm is whether it can be effectively
"scaled" in terms of the number of processors being utilized. Typically, we would like a parallel algorithm to make effective use of all the available processors instead of being dependent on some particular number of processors that it can handle efficiently.

In this paper we consider the transformation of a recursive algorithm into an efficient parallel algorithm for the SMM model. Fig. 1 gives a general outline of how this can be achieved.

![Diagram](image)

Fig. 1. Transformation process.

In this paper we investigate the problem of balancing binary search trees. This is usually accomplished by recursive algorithms. We illustrate all the steps outlined in Fig. 1 by deriving a parallel balancing algorithm that requires only constant running time on an $n$-processor configuration, thus achieving the maximum speed-up
possible.

The second goal of this paper is to motivate the derivation of the iterative as well as the parallel algorithms and we do this by following the stepwise refinement technique advocated by Wirth [16]. Both the iterative balancing algorithm and the parallel balancing algorithm are developed by successive refinement from one version to another. This leads to considerable similarity between the algorithms developed.

The paper is organized as follows. In Section 2, we introduce a recursive algorithm for balancing a binary search tree. In Section 3, we derive a new iterative algorithm for growing balanced binary search trees. In Section 4, we derive the parallel algorithm for balancing binary search trees. In Section 5 we make some general comments on the transformation process.

2. BALANCING BINARY SEARCH TREES

Binary search trees provide a method of data organization that is both flexible and efficient: records are retrieved from binary search trees in an average time proportional to $\log N$ (all logarithms in this paper will be to base 2), where $N$ is the number of nodes in the tree. The task of balancing a binary search tree is to adjust the left and right pointers of the nodes in the tree so that the height of the tree is minimized; the search time will approach $\lceil \log N \rceil$.

Algorithms that dynamically balance tree structure during insertion or deletion of nodes are described in detail by Knuth [11]. An algorithm due to Martin and Ness [12] re-organizes a tree with $n$ nodes by repetitively dividing $n$ by 2 and using the
result as a guideline for stepping through the framework of a perfectly balanced tree, i.e., for each node, the number of nodes in its left subtree and right subtree differ by 1 at most. An Inorder traversal is carried out concurrently to provide relative node positions for the pointer-restructuring procedure which fits them into proper places in the balanced tree structure. In this algorithm, a stack is required to save pointers during traversal. A balancing algorithm that needs no extra storage has been given in Day [6] where the input tree is right-threaded with negative backtrack pointers to allow stackless traversal.

Recently Chang and Iyengar [4] presented an algorithm (hereafter referred to as C-I algorithm) to balance binary search trees in \(O(N)\) time. The algorithm can be summarized as follows.

Phase 1: Traverse the tree in Inorder, storing pointers in a array \(\texttt{LINK}\).

Phase 2: Use a procedure \(\texttt{GROW(LOW:HIGH)}\) to recursively construct a balanced binary search tree from the nodes \(\texttt{LINK(LOW:HIGH)}\).

The interesting aspect of the C-I algorithm is that the left and right subtrees are grown simultaneously from the root. In particular, if the root is linked by the \(M\)'th cell in the array \(\texttt{LINK}\), then, for all \(K < M\), if \(\texttt{LINK(K)}\) has \(\texttt{LINK(j1)}\) and \(\texttt{LINK(j2)}\) as its left and right son, then \(\texttt{LINK(K+M)}\) has \(\texttt{LINK(j1+M)}\) and \(\texttt{LINK(j2+M)}\) as its left and right son. This is referred to as 'folding' with a folding factor \(M\). Calculation of \(M\) and other details can be found in [4].

This algorithm is recursive and, as indicated in Fig. 1, we could either try to first derive an iterative algorithm and then a parallel algorithm from it, or go directly from the recursive algorithm to a parallel algorithm.
In fact, we will be able to transform the recursive algorithm corresponding to Phase 1, which we call the Traversing Algorithm, directly into a parallel algorithm by keeping additional information with the binary search tree. The recursive algorithm corresponding to Phase 2, which we call the Growing Algorithm, will be transformed into an iterative algorithm and then into a parallel algorithm by reorganising the computation appropriately. To simplify the following presentation we will first concentrate on deriving a parallel algorithm for Phase 2 and after that we will consider deriving a parallel algorithm for Phase 1.

3. ITERATIVE GROWING ALGORITHM

Phase 1 of balancing a binary search tree can be achieved by the recursive algorithm A1.

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
We will now develop an iterative growing algorithm for Phase 2 of balancing binary search trees. In a recursive growing algorithm for Phase 2, a contiguous section of the array $LINK$ is converted into a balanced binary search tree by identifying the middle element as the root and then the two subproblems corresponding to its left and right subtrees are solved recursively.

In order to translate this into an iterative algorithm we should be able to identify the final position (in the balanced binary search tree) for any node/cell in the array $LINK$. There is no straightforward way of doing this and so in the spirit of stepwise refinement we will first consider a simpler problem.

3.1. Iterative Algorithm for Growing Complete Balanced Binary Trees

Let us first restructure a binary tree that has $N = 2^n - 1$ nodes. The resulting balanced binary search tree must then be a complete balanced binary search tree and the pointer restructuring required for all the nodes on the same level is similar, as shown by the following two theorems.

**Definition 1**: In a binary tree $T$, the $(i,j)$'th node refers to the $j$'th node from the left on the $i$'th level, if both exist.

**Definition 2**: In a complete balanced binary tree $T$ of $N$ nodes let $CVAL_N(<i,j>)$ denote the number of nodes with data values less than or equal to the data value of the $(i,j)$'th node.

**Theorem 1**: Consider a complete balanced binary tree $T$ with $N = 2^n - 1$ nodes.
If the \(<i,j>\)'th node exists in \(T\) then \(CVAL_N(<i,j>) = 2^{n-i} + (j-1)2^{n-i+1}\).

**Proof**: Let us consider a complete balanced binary tree structure \(T\) with its nodes labelled 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 \(<i,1>\)'th node (if it exists) is labelled \(2^{n-i}\) by the Inorder traversal. Further, if node \(<i,k>\) is labelled \(y\) then the node on the same level to its immediate right (if it exists) is labelled \(y + 2^{n-i+1}\). It then follows that if the \(<i,j>\)'th node exists, \(CVAL_N(<i,j>) = 2^{n-i} + (j-1)2^{n-i+1}\).

**Theorem 2**: Consider a complete balanced binary tree \(T\) with \(N = 2^n - 1\) nodes: if the \(<i,j>\)'th node, \(i < n\), has nodes \(<i1,j1>\) and \(<i2,j2>\) as its \(LSON\) and \(RSON\) respectively then \(CVAL_N(<i1,j1>) = k - 2^{n-i-1}\) and \(CVAL_N(<i2,j2>) = k + 2^{n-i-1}\) where \(k = 2^{n-i} + (j-1)2^{n-i+1}\).

**Proof**: From Theorem 1 we know that \(CVAL_N(<i,j>) = 2^{n-i} + (j-1)2^{n-i+1}\). We also know that in a complete balanced binary tree the \(<i,j>\)'th node \((i < n)\) has \(<i+1,2*j-1>\)'th and \(<i+1,2*j>\)'th nodes as it \(LSON\) and \(RSON\) respectively. So

\[
CVAL_N(<i+1,2*j-1>) = 2^{n-(i+1)} + ((2*j-1)-1)2^{n-(i+1)+1} = k - 2^{n-i-1}
\]

\[
CVAL_N(<i+1,2*j>) = 2^{n-(i+1)} + (2*j-1)2^{n-(i+1)+1} = k + 2^{n-i-1}
\]

This means that for balancing a binary search tree with \(N = 2^n - 1\) nodes, the node corresponding to the \(k'\)th cell in the array \(LINK\) will be the \(<i,j>\) node as determined by the equation
\[ k = 2^{n-i} + (j-1)2^{n-i+1} \]  \hspace{1cm} (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.

\[ k1 = k - 2^{n-i-1} \]  \hspace{1cm} (2)

\[ k2 = k + 2^{n-i-1} \]  \hspace{1cm} (3)

3.2. Efficient Sequencing

One way of translating this into an algorithm would be to sequence through the array \( LINK \) and set up the pointers defined by the above equations. This leads to a slight complication and some inefficiency as it would involve first determining \( i \) and \( j \) from \( k \) in equation (1). This problem can be avoided quite easily by having an algorithm that does not sequence through the array \( LINK \) but instead does the necessary computation for all the cells that have corresponding nodes to be positioned on level 1, and then all the cells that have corresponding nodes to be positioned on level 2, and so on. For all the cells with the corresponding nodes to be positioned on level \( i \), the cell with its corresponding node at position \( <i,j> \) will be processed before the node corresponding to the node at position \( <i,j+1> \). This would then correspond to determining \( k, k1 \) and \( k2 \) from \( i \) and \( j \), which is straightforward. We shall call this \textit{efficient sequencing}. Based on these observations; a binary search tree with \( N = 2^n - 1 \) nodes can be balanced 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.
Step 1 corresponds to Algorithm A1 and Step 2 can be accomplished by Algorithm A2 which is an iterative algorithm.

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 := \( 2^{n-1} + (J-1) \cdot 2^{n-I+1} \) // \( K = CVAL_N(<1,J>) \), using eq. (1) //
        if odd(K) then
            LSON(LINK(K')) := null
            RSON(LINK(K')) := null
        endif
        if even(K) then
            // interior nodes //
            K1 := \( 2^{n-I-1} + (2\cdot J - 2) \cdot 2^{n-I} \) // set up left link, using eq. (2) //
            LSON(LINK(K)) := LINK(K1)
            K2 := \( 2^{n-I-1} + (2\cdot J - 1) \cdot 2^{n-I} \) // set up right link, using eq. (3) //
            RSON(LINK(K)) := LINK(K2)
        endif
    endfor
endfor
end C-LEVEL-STRUCT

The C-I algorithm [4] is based on the folding method in which the similarity of the restructuring for the left and right subtrees of the root was exploited. In Algorithm
A2 we exploit the similarity of the restructuring for all the nodes on the same level and hence generalize the folding method. A straightforward analysis shows that the time complexity of each of the algorithms A1 and A2 is $O(N)$.

3.3. Iterative Algorithm for Growing Balanced Binary Trees

Algorithm A2 was developed under the assumption that $N = 2^n - 1$. If this condition is not satisfied, then a complete balanced binary search tree cannot be constructed. For an arbitrary number of nodes there does not exist either a unique route balanced binary search tree (i.e. one in which the length of the longest and the shortest paths from the root to a leaf differs at most by 1) or a unique perfectly balanced binary search tree. In the remainder of this paper, a balanced binary tree will correspond to a route balanced binary search tree in which all the leaves of the highest level are located at the leftmost side of the tree, as in Fig. 2 which is annotated with $VAL$ value for the corresponding node. Algorithm A1 works for any arbitrary binary tree; and we will now show how Algorithm A2 can be modified to construct route balanced binary search trees for an arbitrary number of nodes.

![Fig. 2. Balanced binary trees.](image-url)
As before, to construct a balanced binary tree, for any arbitrary node \(<i,j>\) in the balanced binary tree we should be able to infer where the node is stored in the array \(LINK\).

**Definition 3**: Let \(VAL_N(<i,j>)\) denote the number of nodes with data values less than or equal to the data value of the node \(<i,j>\) in a binary tree of size \(N\).

From Definition 2 and 3 it follows that \(CVAL_{2^*-1}(<i,j>) = VAL_{2^*-1}(<i,j>)\).

Consider the balanced binary trees in Fig. 2. Fig. 2a corresponds to a balanced binary tree with 10 nodes and Fig. 2b corresponds to a complete balanced binary tree which is obtained from Fig. 2a by adding 5 nodes. To extend Algorithm A2, we must be able to calculate \(VAL_N(<i,j>)\) for arbitrary \(N\). The following theorem relates \(VAL_N(<i,j>)\) and \(VAL_{2^*-1}(<i,j>)\) where \(n = \lceil \log(N+1) \rceil\).

**Theorem 3**: For any balanced tree of \(N\) nodes where \(n = \lceil \log(N+1) \rceil\)

\[ VAL_N(<i,j>) = \min [ VAL_{2^*-1}(<i,j>), [VAL_{2^*-1}(<i,j>) + N - S + 1]/2 ] \]

where \(S = 2^n - 1 - N\), which is the minimum number of nodes that can be added to the balanced binary tree to obtain a completely balanced binary tree.

**Proof**: Let \(L <i,j>\) denote the number of leaves that precede the \(<i,j>\) node (in Inorder traversal) in a balanced binary tree of size \(2^n - 1\). Then the following recurrence relation is satisfied

\[ L <i,j> = L <i,j-1> + 2*L <i,1> \quad j \geq 2 \]

\[ L <i,1> = 2^{n-i-1} \]

This recurrence can be solved to obtain
\[ L <i,j> = (2*j - 1)*2^n - i - 1 \]

In a completely balanced binary tree of size \(2^n - 1\) we know that \(VAL_{2^n-1}(<i,j>) = 2^n - i + (j-1)*2^n - i - 1\). Now in any balanced binary tree of size \(N\) we know that \(S\) leaves at the rightmost side of the tree at the highest level are missing. If some of the leaves that precede the node \(<i,j>\) (Inorder traversal) in a binary tree of size \(2^n - 1\) are missing in a balanced binary tree of size \(N\) then \(VAL_{2^n-1}(<i,j>)\) should be adjusted accordingly to obtain \(VAL_N(<i,j>)\). That is

if \(S \leq 2^n - 1 - L <i,j>\)

\[ VAL_N(<i,j>) = VAL_{2^n-1}(<i,j>) \]

else \( VAL_N(<i,j>) = VAL_{2^n-1}(<i,j>) - [S + L <i,j> - 2^n - 1] \)

The above equation can be simplified to

\[ VAL_N(<i,j>) = \min [VAL_{2^n-1}(<i,j>), \ [VAL_{2^n-1}(<i,j>) - S - (2*j - 1)*2^n - i - 1 + 2^n - 1] ] \]

\[ = \min [VAL_{2^n-1}(<i,j>), \ [VAL_{2^n-1}(<i,j>) + N - S + 1]/2 ] \]

(4)

\[ \square \]

This theorem gives us a method for determining which location in the array corresponds to the \(<i,j>\)'th node in the balanced binary tree. The only remaining thing to be done to develop a general iterative growing algorithm is to be able to determine under what conditions the \(<i,j>\)'th node has null \(LSON/RSON\) pointers. A node has null pointers as specified below.

1. All the leaves on the highest level have null \(LSON\) and \(RSON\) pointers.

\(<i,j>\)'th node is a leaf on the highest level if \(CVAL_{2^n-1}(<i,j>)\) is an odd value and \(CVAL_{2^n-1}(<i,j>) = VAL_N(<i,j>)\).
2. On level \( n-1 \), the rightmost \( \lfloor S/2 \rfloor \) nodes have null \( RSON \) pointers and the rightmost \( \lfloor (S+1)/2 \rfloor \) nodes have null \( LSON \) pointers. If node \( \langle i, j \rangle \) is on level \( n-1 \) then both \( CVAL_{2^n-1}(\langle i+1, 2^j-1 \rangle) \) and \( CVAL_{2^n-1}(\langle i+1, 2^j \rangle) \) have odd values. Further, if \( CVAL_{2^n-1}(\langle i+1, 2^j-1 \rangle) \neq VAL_{2^n-1}(\langle i+1, 2^j-1 \rangle) \) then the \( \langle i, j \rangle \)'th node has null \( LSON \) pointer; and if \( CVAL_{2^n-1}(\langle i+1, 2^j \rangle) \neq VAL_{2^n-1}(\langle i+1, 2^j \rangle) \) then the \( \langle i, j \rangle \)'th node has null \( RSON \) pointer.

These observations immediately give rise to Algorithm A3 which has \( O(N) \) time complexity. Note that Algorithm A2 can be obtained from Algorithm A3 as a special case by setting \( S = 0 \).

4. DERIVATION OF PARALLEL ALGORITHM

We will first consider the development of a parallel algorithm to grow a balanced binary search tree from the array \( LINK \) and after that we will consider the problem of a parallel traversal of a binary search tree. The model of parallel computation that we will be using is a simplification of the shared memory model (SMM) and can be characterized as follows:

1. There are \( M \) processing elements (PEs) or processors. These are indexed 1, 2, \( \ldots \), \( M \) and the \( i^{th} \) PE is referenced as \( P_i \). Each PE has the capability of performing all the standard arithmetic and logic operations. Each PE knows its own index.

2. There is a common memory that is shared among all the PEs. All PEs can read and write into this memory at any time instance but if two or more PEs attempt to read (write) from (to) the same memory location, a read (write) conflict occurs.
Algorithm A3: Iterative growing algorithm. 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 LEVEL-STRUCT

declare S, N, n, I, J, K, K', K1, K1', K2, K2', LINK(1:N), LSON(1:N), RSON(1:N)

S := 2^n - 1 - N

for I := 1 to n-1 do  // for each level //
    for J := 1 to 2^{I-1} do  // for all nodes on a level //
        K := 2^{I-1} + (J-1) * 2^{I-1}
        K' := (K + N - S + 1) / 2
        if K' > K then K' := K
        if odd(K) ∧ (K = K') then
            LSON(LINK(K')) := null
            RSON(LINK(K')) := null
        endif
        if even(K) then
            K1 := 2^{I-1} + (2∗J - 2) * 2^{I-1}
            K1' := (K1 + N - S + 1) / 2
            if K1' > K1 then K1' := K1
            if odd(K1) ∧ (K1 ≠ K1')
                then LSON(LINK(K1')) := null
                else LSON(LINK(K1')) := LINK(K1')
            endif
            K2 := 2^{I-1} + (2∗J - 1) * 2^{I-1}
            K2' := (K2 + N - S + 1) / 2
            if K2' > K2 then K2' := K2
            if odd(K2) ∧ (K2 ≠ K2')
                then RSON(LINK(K2')) := null
                else RSON(LINK(K2')) := LINK(K2')
            endif
        endif
    endfor
endfor
end LEVEL-STRUCT
3. The PEs need not be synchronized.

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

The straightforward approach for transforming Algorithm A3 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 $VAL_N(\langle i, j \rangle) = k$; that is its final position in the balanced binary tree

R2. if it has a non-null $LSON$ then determine $k1$ such that $VAL_N(\langle i + 1, 2*j - 1 \rangle) = k1$

R3. if it has a non-null $RSON$ then determine $k2$ such that $VAL_N(\langle i + 1, 2*j \rangle) = k2$

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

4.1. Parallel Algorithm for Growing Complete Balanced Binary Trees

Following the stepwise refinement methodology, we first consider the simpler case where a complete balanced binary tree is to be constructed. First, for any processor $P_k$ we should be able to determine in a constant time the indices $i, j$ such that $VAL_{2^*-1}(\langle i, j \rangle) = k$. Consider Fig. 3 where, as before, the nodes are annotated
with their $VAL$ values. It is obvious that a processor $P_k$ may correspond to different indices $<i+M,j>$, $M = 1, 2, ...$ (note $j$ will be constant) depending on the size of the complete balanced binary tree. But what is invariant over the size of the complete balanced binary tree are the indices $h, j$ 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 indices $<i,j>$ and hence requirement R1 can be met in constant time. It is this observation, that we can assign computation to the processes in such a way that the indices $h$ 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 indices $i, j$ can be determined for each process $P_k$, but not in constant time.)

![Diagram](image)

(a) node with value 6 has $j=2$, $h=2$   
(b) node with value 6 has $j=2$, $h=2$

Fig. 3. Node identification which is invariant over size of complete balanced binary tree.

The requirements R2 and R3 can be satisfied very easily in constant time since from Theorem 3 we know how to calculate $VAL_{2^r-1}(<i,j>)$ for any arbitrary $i$ and $j$. 

- 17 -
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 $P_1, \ldots, P_N$, one for each cell in the array LINK, where $N = 2^n - 1$.

```plaintext
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
            LSON(LINK(K)) := null
            RSON(LINK(K)) := null
        endif
        if even(K) then
            LSON(LINK(K)) := LINK(K-2^n-1-1)
            RSON(LINK(K)) := LINK(K+2^n-1-1)
        endif
    endfor
end PC-LEVEL-STRUCT
```

4.2. **Parallel Algorithm for Growing Balanced Binary Trees**

When considering arbitrary balanced binary trees, a cell $LINK(k)$ may correspond to different indices $<i,j>$ and $<i',j'>$ where $i \neq i'$ and $j \neq j'$, as the size of the tree changes. The calculation of the indices becomes much more complicated (and time consuming) in this case but this can be avoided in the following way.
Let process $P_k$ be associated with $LINK(k)$. Instead of allowing a process $P_k$ to determine the indices $<i,j>$ to set up links for $LINK(h)$, we will allow the possibility of some other process $P_m$, $m \neq h$ setting up the links for $LINK(h)$. The choice of which processor $P_{k'}$ to set up links for $LINK(h)$ must be made so that $P_{k'}$ can easily determine the $<i,j>$ indices required for setting up links for $LINK(h)$. We will allow a process $P_{k'}$ to set up links for $LINK(h)$ where

$$h = \min \left[ h', [h' + N - S + 1]/2 \right]$$

This corresponds to 'reorganising the computation'.

To translate this into an algorithm would require the use of $2^n - 1$ processors where $n = \lceil \log(N+1) \rceil$ for a tree of size $N$ nodes, as indicated in Algorithm A5. The structure of Algorithm A3 consists of two nested loops. We have derived a parallel algorithm A5 from this by assigning computation to processors in such a way that each processor can determine its own distinct pair of indices corresponding to the two loops in constant time. Once this is possible, the rest of the computation can be accomplished in constant time as it amounts to having each processor execute the body of the inner loop of Algorithm A3 once.

In Algorithm A5, we require at times more processors than the size of the corresponding tree. As an illustration, Fig. 4 shows the details of the computation done for a tree of size 10. To avoid using more processors than the size of the tree, we would like to be able derive a new algorithm A6 that would reorganise the computation as shown in the last column of Fig. 4. The translation from A5 to A6 to achieve this is straightforward; a processor $P_k$ that was not setting up any links in A5 is made to take over the responsibility of processor $P_{k+S}$. Both Algorithms A5 and A6 run in constant time on an appropriate multiprocessor configuration and no read or write conflicts are possible in their execution.
Algorithm A5: Parallel growing algorithm for the construction of a balanced binary tree by simultaneously executing processes $P_1,...,P_n$, where $n = \lceil \log(N+1) \rceil$.

**procedure** P-LEVEL-STRUCT

```
declare N, LINK(1:N), LSON(1:N), RSON(1:N)
for each PE $P_K$ do in parallel
    declare n, N, H, I, J, S, K, K', K1, K1', K2, K2'
    constant H, J
    n := $\lceil \log(N+1) \rceil$
    I := n - H
    S := $2^n - 1 - N$
    K := $2^{n-1} + (J-1) \cdot 2^{n-1}$
    K' := $(K + N - S + 1) / 2$
    if K' > K then K' := K  // $K' = \text{VAL}_N(<I,J>)$  //
    if odd(K) $\land$ (K = K') then  // leaves on the highest level //
        LSON(LINK(K')) := null
        RSON(LINK(K')) := null
    endif
    if even(K) then  // not on the highest level //
        K1 := $2^{n-1} + (2\cdot J - 2) \cdot 2^{n-1}$  // set up left link //
        K1' := $(K1 + N - S + 1) / 2$
        if K1' > K1 then K1' := K1  // $K1' = \text{VAL}_N(<I+1,2\cdot J-1>)$  //
        if odd(K1) $\land$ (K1 $\neq$ K1')  // odd(K1) $\Rightarrow$ K' on level n-1 //
            then LSON(LINK(K1')) := null
        else LSON(LINK(K1')) := LINK(K1')
        endif
    K2 := $2^{n-1} + (2\cdot J - 1) \cdot 2^{n-1}$  // set up right link //
    K2' := $(K2 + N - S + 1) / 2$
    if K2' > K2 then K2' := K2  // $K2' = \text{VAL}_N(<I+1,2\cdot J>)$  //
    if odd(K2) $\land$ (K2 $\neq$ K2')  // odd(K2) $\Rightarrow$ K' on level n-1 //
        then RSON(LINK(K2')) := null
    else RSON(LINK(K2')) := LINK(K2')
    endif
endfor
```

end P-LEVEL-STRUCT
Algorithm A6: Parallel growing algorithm for the construction of a balanced binary tree by executing simultaneously processes $P_1, \ldots, P_N$.

procedure P-LEVEL-STRUCT1

declare $N$, LINK(1:N), LSON(1:N), RSON(1:N)
for each PE $P_K$ do in parallel

declare $n$, $N$, $H$, $I$, $J$, $S$, $K$, $K'$, $K1'$, $K2'$
constant $H$, $J$
$n := \lceil \log(N+1) \rceil$
$I := n - H$
$S := 2^a - 1 - N$
$K := 2^{a-1} + (J-1) \cdot 2^{a-1}$
$K' := (K + N - S + 1) / 2$
if $K' > K$ then $K' := K$ // $K' = VAL_N(<I,J>)$ //
if odd($K$) \& (K = K') then // leaves on the highest level //
    LSON(LINK(K')) := null
    RSON(LINK(K')) := null
endif
if odd($K$) \& (K \neq K') then // compute for $P_{K+S}$ //
    $K := K + S$
    $K' := (K + N - S + 1) / 2$
    if $K' > K$ then $K' := K$
endif
if even($K$) then
    $K1 := 2^{a-1} - (2J - 2) \cdot 2^{a-1}$ // set up left link //
    $K1' := (K1 + N - S + 1) / 2$
    if $K1' > K1$ then $K1' := K1$ // $K1' = VAL_N(<I+1,2J-1>)$ //
    if odd($K1$) \& (K1 \neq K1') // odd($K1$) $\Rightarrow$ $K'$ on level n-1 //
        then LSON(LINK(K1')) := null
        else LSON(LINK(K1')) := LINK(K1')
    endif
    $K2 := 2^{a-1} + (2J - 1) \cdot 2^{a-1}$ // set up right link //
    $K2' := (K2 + N - S + 1) / 2$
    if $K2' > K2$ then $K2' := K2$ // $K2' = VAL_N(<I+1,2J>)$ //
    if odd($K2$) \& (K2 \neq K2') // odd($K2$) $\Rightarrow$ $K'$ on level n-1 //
        then RSON(LINK(K2')) := null
        else RSON(LINK(K2')) := LINK(K2')
    endif
endfor
end P-LEVEL-STRUCT1
<table>
<thead>
<tr>
<th>Processor</th>
<th>Alg. A5 sets up links for</th>
<th>Alg. A6 sets up links for</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{15}$</td>
<td>—</td>
<td>not used</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>LINK(10)</td>
<td>not used</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>—</td>
<td>not used</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>LINK(9)</td>
<td>not used</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>—</td>
<td>not used</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>LINK(8)</td>
<td>LINK(8)</td>
</tr>
<tr>
<td>$P_{9}$</td>
<td>—</td>
<td>LINK(10)</td>
</tr>
<tr>
<td>$P_{8}$</td>
<td>LINK(7)</td>
<td>LINK(7)</td>
</tr>
<tr>
<td>$P_{7}$</td>
<td>—</td>
<td>LINK(9)</td>
</tr>
<tr>
<td>$P_{6}$</td>
<td>LINK(6)</td>
<td>LINK(6)</td>
</tr>
<tr>
<td>$P_{5}$</td>
<td>LINK(5)</td>
<td>LINK(5)</td>
</tr>
<tr>
<td>$P_{4}$</td>
<td>LINK(4)</td>
<td>LINK(4)</td>
</tr>
<tr>
<td>$P_{3}$</td>
<td>LINK(3)</td>
<td>LINK(3)</td>
</tr>
<tr>
<td>$P_{2}$</td>
<td>LINK(2)</td>
<td>LINK(2)</td>
</tr>
<tr>
<td>$P_{1}$</td>
<td>LINK(1)</td>
<td>LINK(1)</td>
</tr>
</tbody>
</table>

Fig. 4. Reorganising computation.

4.3. Parallel Traversal

In this section we derive a parallel traversal algorithm from the recursive algorithm A1. We could first try to rewrite A1 as an iterative algorithm and then derive a parallel algorithm from that. Such an iterative algorithm could somehow try to mimic the recursive algorithm A1 by explicitly maintaining a stack. Instead, we are going to directly derive a parallel algorithm from algorithm A1 by keeping extra
information with the binary search tree. Of course, we can always keep a complete
copy of the entire binary search tree at each node and then be able to come up with
a parallel algorithm for traversal. But such a solution is impractical both in terms of
the extra time and space required so that each node has a global view of the entire
binary search tree, and also in terms of the time needed to make use of the extra
information to do the traversal of the tree. The extra information we would like to
keep should be simple and easy to update as values are added and removed from
the binary search tree.

When considering a binary search tree of size $N$, if $N$ processors are available then
each processor can be associated with a unique node of the binary search tree. The
extra information kept at each node should ideally allow the processor associated
with that node to infer the cell in the array $LINK$ to which it will be linked. Now a
node $i$ will be linked to the cell $LINK(k)$ if there are exactly $k$ nodes in the binary
search tree with values less than or equal to the value associated with node $i$. So, if
with each node we kept the information about the number of nodes that have
values less than or equal to its value, the traversal of the binary search tree can be
accomplished in constant time.

We will therefore assume that with every node in the binary search tree an addi-
tional field $NUM$ is associated. The value of this field for node $i$, $NUM(i)$, gives
the number of nodes in the binary search tree that have values less than or equal to
the value associated with the node $i$. It is straightforward to see that the value of
this new field is altered only when entries are added or removed from the binary
search tree and that this field value can be updated as entries are added and deleted
from the binary search tree without any increase in the time complexity of perform-
ing these operations.
Algorithm A7 allows the parallel traversal of a binary search tree under the assumption that extra information is stored with each node; it also has a constant time complexity.

Algorithm A7: Parallel traversal algorithm by processes $P_1, \ldots, P_N$; 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

5. CONCLUSIONS

In this paper we have shown how the problem of balancing a binary search tree can be solved efficiently on a multiprocessor configuration. The algorithm developed achieves the maximum possible speed-up. The algorithms that we have presented can also be easily scaled so that a tree of size $N$ can be balanced in time $constant \times \lceil N/k \rceil$ if $k$ processors are available. The various theorems can also be extended to cover any $n$-ary tree and hence parallel algorithms to balance $n$-ary trees can be derived. Another straightforward extension is to derive algorithms to obtain perfectly balanced trees. We have also shown how the algorithms are developed by the stepwise refinement methodology of Wirth. The relationship between the various algorithms is shown in Fig. 5.
Development of Traversing Algorithm

keep additional information

Development of Growing Algorithm

recursive growing algorithm C-I [4]

efficient sequencing

special case iterative algorithm A2

compensate for missing leaves on level n
allocate processors so that the do loop indices can be determined in constant time

special case parallel algorithm A4

compensate for missing leaves on level n and reorganise computation for efficiency

parallel algorithm A5
reorganise computation to remove the useless processes

parallel algorithm A6

Fig. 5. Development of parallel algorithms for balanced binary trees.
ACKNOWLEDGEMENTS

We are grateful to Professors J. L. Bentley, M. Joseph and F. B. Schneider for their useful comments on previous versions of this paper.

REFERENCES


