International Journal of Parallel Programming, Vol. 15, No. 6, 1986

Optimal Parallel Algorithms for Constructing and Maintaining a Balanced *m*-way Search Tree

Eliezer Dekel,[1] Shietung Peng,[1] and S. Sitharma Iyengar[2]

Received July 1987; revised May 1987

We present parallel algorithms for constructing and maintaining balanced *m*-way search trees. These parallel algorithms have time complexity O(1) for an *n* processors configuration. The formal correctness of the algorithms is given in detail.

KEY WORDS(S): Parallel algorithms; MIMD; search trees.

1. INTRODUCTION

The use of tree structures to represent symbol tables, dictionaries has been extensively studied.(1) In all these structures we hae a collection of records that are to be manipulated with regard to a certain key field in a record. Common operations on these structures are SEARCH, INSERT, and DELETE. SEARCH(K) returns a pointer to the record that contains the requested key field K. If no record with key K is in the given collection, it returns a pointer to the location in which a record with such a key can be inserted. INSERT(R) inserts a new record into the collection. DELETE(K) removes the record with key K from the collection. The tree structure supports efficient INSERT, SEARCH, and DELETE operations. In some implementations the operations are designed so that a balanced tree is maintained through the process. Another approach is to periodically rebalance the tree.

[1] University of Texas at Dallas, Programs in Computer Science.
[2] Lousiana State University, Computer Science Dept.

0885-7458/86/1200-0503$05.00/0

In our discussion we refer to these structures as dictionaries. The operations INSERT, SEARCH, and DELETE will be referred to as basic dictionary operations.

When the data associated with the dictionary fit into main memory, the most common structure used would be a balanced binary search tree. When the data are too big to fit in main memory, a balanced m-way search tree would be used. Many type of balanced m-way search trees are reported in Refs. 1 and 2. In our discussion we refer to the following:

Definition 1.1. An m-way search tree, T, is a tree is which all internal nodes are of degree $\leqslant$M. If T is empty, then T is an m-way search tree. When T is not empty, it has the following properties:

1. T is a node of type A_0, (K_1, A_1), (K_2, A_2),..., (K_{m-1}, A_{m-1}) where the A_i, $0 \leqslant i < m$, are pointers to the subtrees of T and the K_i, $1 \leqslant i < m$, are key values.
2. $K_i < K_{i+1}$, $\quad 1 \leqslant i < m-1$.
3. All key values in subtree A_i are less than value K_{i+1}, $0 \leqslant i < m-1$.
4. All key values in the subtree A_{m-1} are greater than K_{m-1}.
5. The subtrees A_i, $0 \leqslant i \leqslant m-1$, are also m-way search trees.

As an example of a 4-way search tree consider the tree of Fig. 1. This tree is constructed for key values 2, 3, 10, 11, 14, 16, 17, 19, 22, 25, 30, 32, 35, 40, 47, 60, 65, 90. In order to search for any key value x in this tree, we start at the root T and look for keys K_i and K_{i+1} for which $K_i \leqslant x < K_{i+1}$ (for convenience we assume the existence of keys $K_0 = -\infty$ and

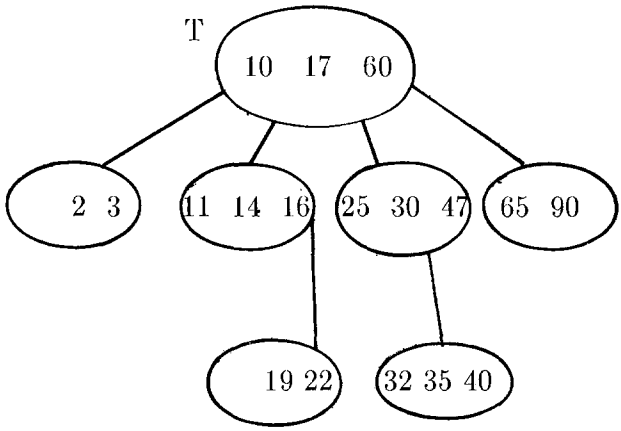

Fig. 1. Example of a 4-way search tree.

$K_{n+1} = \infty$). In case $x = K_i$ then the search is complete. Otherwise, by the definition of the m-way tree, x must be in sbstree A_i if it is in the tree. Clearly, the 4-way search tree in Fig. 1 is not the only possibility for a 4-way search tree construction with the given keys. In general we would prefer the construction with the minimal possible height.

Definition 1.2. A balanced m-way search tree is an m-way search tree minimal height.

In this paper, we present parallel algorithms for balancing and maintaining m-way search tree. Since the complexity of a parallel algorithm depends very much on the architecture of the parallel machine on which it runs, it is necessary to keep the architecture in mind when designing parallel algorithms. Many parallel architectures have been proposed and studied. In this paper, we deal directly with only the multiple-instruction stream, multiple-data stream (MIMD) model. Our technique and algorithms readily adapt to other models (e.g., single-instruction stream, multiple-data stream (SIMD) and data flow models). MIMD computers have the following characteristics.

(1) The consist of p processing elements (PEs). The PEs are indexed $0, 1, \ldots, p-1$, and an individual PE may be referenced as PE(i). Each PE is capable of performing the standard arithmetic and logical operations. In addition, each PE knows its index.

(2) Each PE has some local memory.

(3) The PEs operate asynchronously under the control of individual instruction streams.

(4) Different PEs can execute different instructions at any time.

During the computation the PEs communicate results to each other. In many MIMD models the time required to communicate data from PE to PE often dominates the overall complexity of the algorithm. Several interprocessor communication models for MIMD computers have been proposed in the literature.

The communication overhead of an algorithm varies from one communication model to another. To simplify the discussion we deal only with the shared-memory model (SMM) in this paper. This model has no communication delay. In a shared-memory computer there is a large common memory that is shared by all PEs. It is assumed that any PE can access any word in common memory in O(1) time. When two or more PEs access the same memory word simultaneously, we say that a conflict has occurred. As far as our discussion is concerned, no conflicts are allowed.

The advent of parallel processing (specifically VLSI) has led to the development of a number of special purpose parallel machines to support

dictionary structures.[3–10] Leiserson,[7] Bently and Kung,[5] Ottmann, *et al.*,[8] and Atallah and Kosaraju[4] proposed pipelined architectures based on a balanced binary tree. $O(N)$ PEs are used in order to support a search tree containing up to N elements. The machines vary in their wiring complexity and the variety of dictionary operations that they support. Generally, all of them can perform the basic dictionary operations in $O(\log N)$ time. They vary on the input pipeline intervals. The Atallah and Kosaraju's machine[4] provides $O(\log N)$ performance with a pipeline interval of $O(1)$ for a wide range of dictionary operations. (n is the actual number of elements stored in the $O(N)$ PEs machine). These designs maintain the dictionary elements in some sorted order. Somani and Agarwal[9] propose a binary tree machine with $O(N)$ PEs that does not require any sorted order for the dictionary elements. Their design supports all dictionary operations and provides an $O(\log n)$ time performance with constant pipeline interval. Armstrong,[3] Tanaka, *et al.*,[10] and Carey and Thompson[6] propose a pipelined architecture for maintaining a search tree of N elements with only $O(\log N)$ PEs. The most recent design, by Carey and Thompson supports the richest set of dictionary operations among these three designs. This design can perform the basic dictionary operations in $O(\log N)$ time with $O(1)$ pipeline interval. Fisher[11] developed an architecture based on the Trie structure.[1] In his design the number of PEs is proportional to the length of the maximum key. Like the Carey and Thompson's design,[6] his machine supports a smaller set of operations than the $O(N)$ PEs machines. Fishers scheme[11] is advantageous when the dictionary keys are long.

These pipelined architectures achieve only an $O(\log N)$ throughput improvement over the serial balanced tree algorithms. When the number of records in the dictionary becomes bigger than N these designs will not function. That is, the hardware is tailored to the maximal possible number of elements in the tables. While to $O(N)$ PEs architectures can handle efficiently operations beyond basic dictionary functions, they have no advantage over the $O(\log N)$ designs when only the basic operations are considered.

These parallel designs follow the trend in most serial algorithms for search trees. They maintain the tree balanced through the INSERT and DELETE operations by splitting and combining nodes. Another strategy is to allow insert and delete to "unbalance" the tree and to periodically rebalance the entire tree.[12–14] Recently, Moitra and Iyengar[13] explored a technique of transforming a sequential algorithm for balancing a binary search tree into an efficient parallel algorithm. Furthermore, they have shown that the resulting parallel algorithm has a time complexity $O(1)$ when a tree with N elements is balanced with N PEs. An $O(\log N)$ time

set-up overhead is incurred when a new N is considered. For broader treatment of this, see Ref. 13.

Manber[15] discusses the concurrent maintenance of a variation of the binary search tree. He considers the basic dictionary operations in a concurrent environment. His approach is to allow the tree to become unbalanced as a result of an INSERT or DELETE operation. To allow rebalancing, he introduces maintenance processes.

In this paper, we consider the construction of parallel algorithms for balancing and maintaining a general m-way search tree. These parallel algorithms are examples of algorithms with no communication overhead. That is, once the assigned processors read in the input, they do not need to communicate until after the result is written out. Our algorithms have O(1) time complexity on an n-PEs configuration, where n is the number of elements in the tree. No setup overhead is required.

The complexity analysis is carried out on the assumption that as many PEs as needed are available. This assumption is of course unrealistic. A parallel algorithm will eventually be run on a machine with a finite number of PEs, say k. It should be easy to see that all our algorithms are easily adapted to the case of k PEs. If our algorithm has O(1) complexity using $O(N)$ PEs, then with k PEs $k < N$, its complexity is $O(N/k)$. Thus wth only 1 PE our tree balancing algorithm will have the same time complexity as the best serial algorithm for this problem.

The paper is organized as follows. In Section 2, we review some properties of m-way search trees. In Section 3 we develop the general m-way search tree rebalancing/construction algorithm, we discuss in detail the development of the algorithm and its correctness. I this section we also present algorithms for rebalancing after one insertion or one deletion. In Section 4, we discuss the implementation of the algorithms on an MIMD machine.

2. *M*-WAY SEARCH TREES

In this section we review some general properties of m-way search trees. As mentioned in the introduction, m-way trees are used to represent dictionaries that do not fit in internal memory, i.e., m-way search trees are structures tailored for external search. The choice of m is hardware dependent.

Dictionary searches are more efficient when they are done on a balanced tree. Thus the insertion and deletion algorithms are designed to leave the tree balanced. Because of this requirement, it is more convenient to consider balanced m-way search trees that are not necessarily of minimal height (e.g., B-trees). These trees lend themselves to easier splitting and

combining. While this is true in the serial case, we show that in the parallel case the complete m-way search tree proves to be an efficient choice.

We now present some definitions and review properties of m-way search trees.

Definition 2.1. A search tree of minimal height is called route-balanced.

It is easy to show the correctness of the following two lemmas:

Lemma 2.1. The maximum number of nodes on level i of an m-way search tree is $m^{(i-1)}$.

Lemma 2.2. The maximum number of nodes in an m-way search tree of height h is $(m^h - 1)/(m - 1)$.

Definition 2.2. A full m-way search tree of heigh h is an m-way tree of height h having $(m^h - 1)/(m - 1)$ nodes.

Definition 2.3. A level labeling of an m-way search tree is a labeling in which nodes are numbered sequentially, from top down and left to right. The root node is always labeled 1.

Definition 2.4. An m-way search tree with n nodes and of height h is complete if its nodes correspond to the nodes which are numbered 1 to n (by level labeling) in a full m-way search tree of height h.

Figure 2 shows a full 4-way tree of height 3. Its 21 nodes are labeled by level labeling. In Fig. 3, we show a complete 3-way tree.

Lemma 2.3. If the nodes of a complete m-way tree are labeled by level labeling, then the m children of node i (if they exist) are labeled $(i-1)*m+2+j$, $0 \leqslant j \leqslant m-1$.

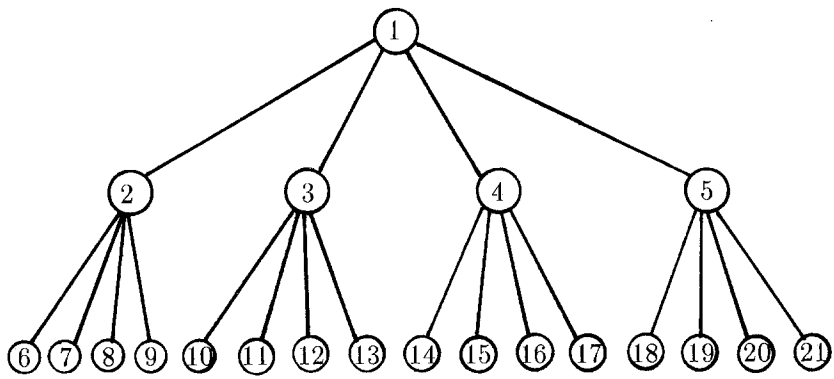

Fig. 2. A full 4-way tree.

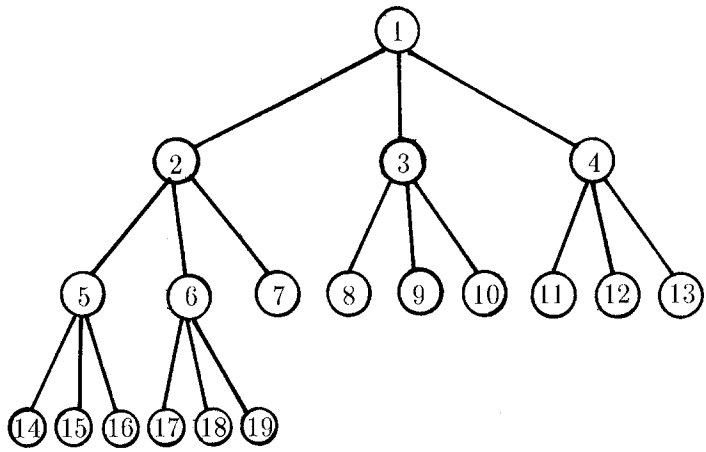

Fig. 3. A complete 3-way tree.

Proof. By induction on the number of nodes in the complete m-way tree.

Definition 2.4. In an m-way tree T, the (i, j)'th node refers to the jth node, (from the left), on the ith level, if it exists. We refer to this indexing method for m-way search trees as two-dimensional indexing.

Lemma 2.5. A full m-way search tree of height h can accommodate at most $m^h - 1$ key elements.

Proof. This follows directly from Lemma 2.2 and the m-way search tree definition (Def. 1.1).

Definition 2.5. Inorder traversal of an m-way search tree is defined by the following recursive procedure:

```
procedure MINORDER(T)
{*T is an m-way search tree as defined in Def. 1.1*}
if T≠null then
    begin
    call MINORDER(A0);
    for i:=1 to m-1
        begin
        if no Ki key then exit;
        visit(Ki);
        MINORDER(Ai);
        end;
    end;
end. {*MINORDER*}
```

Definition 2.6. An index can be associated with each key in an m-way search tree. If this index corresponds to the order in which MINORDER visits the keys, it is called the inorder indexing.

Figure 4 shows a 3-way search tree. The inorder index is given for each key. The inorder index can be obtained by modifying MINORDER to execute the line $X_i := \text{count} + 1$ in place of the line visit(K_i). The variable count should be a global variable initialized to zero by the calling program, and X_i will be the inorder index associated with key K_i. It is easy to see that if the keys are printed out by their inorder index the result would be a sorted sequence of keys. In the following lemmas we consider the relation between the inorder indexing of a key and its location in the m-way search tree.

Lemma 2.6. Suppose node I is at level r of a full m-way search tree of heigh h. Let the inorder index associated with key K_i in I be X_i. Then $X_{i+1} - X_i = m^{h-r}$ for $1 \leqslant i < m-1$.

Proof. The keys visited by MINORDER between K_i and K_{i+1} are exactly the keys of the subtree pointed by pointer A_i. These keys will have inorder indexes starting from $X_i + 1$ to $X_{i+1} - 1$. Since the tree is a full tree, the height of the subtree A_i is $h-r$. The number of keys in a tree of height $h-r$ is $m^{(h-r)} - 1$ (Lemma 2.5). Hence $X_{i+1} - X_i = m^{(h-r)}$ for $1 \leqslant i < m-1$.

Lemma 2.7. a) Let nodes I and J be two adjacent sibling nodes at level r of a full m-way search tree of heigh h. Let $I.X_{m-1}$ be the inorder index of the last key in node I and $J.X_1$ the inorder index of the first key in node J. Then $J.X_1 - I.X_{m-1} = 2 * m^{(h-r)}$.

b) The previous claim is true for any two adjacent nodes at level r.

Proof. a) Observe that MINORDER, after traversing $I.K_{m-1}$ and assigning it an index $I.X_{m-1}$, goes on to traverse subtree $I.A_{m-1}$. When

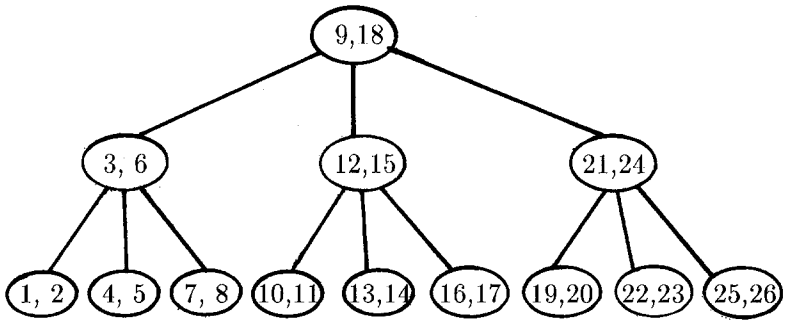

Fig. 4. A full 3-way search tree.

this tree is completed the procedure will traverse $S.K_f$, where $S.A_{f-1} = I$ and $S.A_f = J$ (S is the parent node of I and J). Next MINORDER will traverse subtree $J.A_0$ and only then key $J.K_1$. Thus indexes between $I.X_{m-1}$ and $J.X_1$ are associated with keys in two subtrees of height $h-r$ and an extra key in the parent node. Since each subtree has $m^{(h-r)}-1$ keys (Lemma 2.5), we get: $J.X_1 - I.X_{m-1} = 2 * (m^{(h-r)}-1) + 1 + 1 = 2 * m^{(h-r)}$.

b) One can readily observe that the only difference between sibling nodes at level r and nonsibling adjacent nodes is the level in which the extra key resides. For the sibling case the extra key is one level above. If the parents of the nodes are siblings then the extra node is two levels above, and so on.

Theorem 2.1. Consider a full m-way search tree of height h. The inorder indexes for keys in node i at level r (node(r, i)) are: $(i-1) * m^{(h-r+1)} + j * m^{(h-r)}$, where $1 \leqslant i \leqslant m^{(r-1)}$; $1 \leqslant j < m$.

Proof. The number of nodes at level r is $m^{(r-1)}$ (Lemma 2.1). Pointer A_0 in the first node (from the left) points to a subtree with $m^{(h-r)}-1$ key elements (Lemma 2.5). All these elements are traversed before the first element in this node. Hence the inorder index of this key is $m^{(h-r)}$. From Lemma 2.6 we know that the inorder index of the second element in that node should be $2 * m^{(h-r)}$. From Lemma 2.7 we get that the inorder index of the first element in the second node should be $x + 2 * m^{(h-r)}$, where x is the index of the last element in the first node. Thus the first m nodes should contain elements with the following indexes:

$$m^{(h-r)}, 2 * m^{(h-r)}, \ldots, (m-1) * m^{(h-r)}, \qquad i = 1 \text{ first node}$$

$$m^{(h-r+1)} + m^{(h-r)}, \ldots, m^{(h-r+1)} + (m-1) * m^{(h-r)}, \qquad i = 2$$

$$\ldots\ldots\ldots\ldots,$$

$$(m-1) * m^{(h-r+1)} + m^{(h-r)}, \ldots, (m-1) * m^{(h-r+1)} + (m-1) * m^{(h-r)}, \quad i = m$$

We may now generalize for arbitrary node (r, i), and conclude that it contains elements with indexes as claimed.

Corollary 2.1. A key with inorder index t is at level r of a full m-way search tree if and only if $t \bmod m^{(h-r+1)} \neq 0$ and $t \bmod m^{(h-r)} = 0$.

Proof. This readily follows from Theorem 2.1.

Corollary 2.2. Let t be the inorder index associated with a key at level r of an m-way tree of height h, and let $q = \lfloor t/m^{(h-r+1)} \rfloor$. Keys with the same r value (Cor. 2.1) and q value are in the same node, in the m-way search tree. Moreover, the two-dimensional indexing of this node will be $(r, q+1)$.

Proof. Integer division of $(i-1) * m^{(h-r+1)} - j * m^{(h-r)}$ by $m^{(h-r+1)}$ yields $(i-1)$, where i is the node number in the previous representation (Theorem 2.1).

Corollary 2.3. Let t be the inorder index associated with a key at level r of an m-way tree of height h. The position of the key within the node is given by s, $1 \leqslant s \leqslant m-1$, where $s = \lfloor (t \bmod m^{(h-r+1)})/m^{(h-r)} \rfloor$.

Proof. This readily follows from Theorem 2.1.

Lemma 2.8. Let h be highest level in a complete m-way tree. The inorder indexes associated with keys in node (h, p) are $(p-1) * m + j$, where $1 \leqslant j < u$. (h, v) is the last node in this level, $v \leqslant m^{(h-1)}$. $u = m-1$ for all nodes except (h, v).

Proof. Observe that the leftmost key in this level (K_1 of node $(h, 1)$) is the first key to be visited by MINORDER. Hence the inorder index for this key is 1. Since all the pointers in level h are set to null, the next key to be visited by MINORDER will be key K_2 of node $(h, 1)$, the index for this key is 2. In the same manner the inorder index for the $m-1$ keys of node $(h, 1)$ (if they exist.) are assigned inorder indexes 1, 2,..., $m-1$. The first key of node $(h, 2)$ is visited after key K_f in the parent node is visited, where pointers A_{f-1} and A_f are pointing at nodes $(h, 1)$ and $(h, 2)$ respectively. It follows that the inorder index of K_1 in node $(h, 2)$ is $m+1$. Observe that MINORDER will visit a key in a node at level $s < h$ after the last key of node (h, p) and before the first key of node $(h, p+1)$. The key visited is in a node that is the nearest common ancestor to nodes (h, p) and $(h, p+1)$. For example the key visited after the last key of node (h, m) and before the first key of node $(h, m+1)$ is in a node at level $h-2$. Thus the keys within a node at level n are indexed by consecutive integers. The inorder index associated with the first key in node (h, p) is $g+2$, where g is the inorder index of the last key in node $(h, p-1)$. Hence the inorder indexes associated with node (h, p) are $(p-1) * m + j$, where $1 \leqslant j \leqslant u$ and u is the number of keys in this node.

Corollary 2.4. Consider a complete m-way search tree of height h. Let w be the inorder index of the last key in the rightmost node, (h, v), in level h. Furthermore let the number of keys in node (h, v) be u. Then $w = (v-1) * m + u$.

Proof. This follows immediately from Lemma 2.8.

3. ALGORITHMS FOR REBALANCING AN M-WAY SEARCH TREE

Operations of m-way search tree are most efficient when the tree is balanced. In this section we develop parallel algorithms for balancing a general m-way tree. We develop first a rebalancing algorithm for the special case where the number of keys in the m-way search tree fit exactly into a full m-way search tree of height h. this algorithm is then generalized to rebalance an m-way search tree with any number of keys. Finally we present rebalancing algorithms for the basic dictionary operations: INSERT and DELETE. A discussion about SEARCH is given in Section 4.

It is instructive to look first at the case where the number of keys in the tree is $m^h - 1$ for some $h \geq 1$. This is the maximal number of keys that can be accommodated in a full m-way search tree of height h (Lemma 2.5). We assume that the input m-way search tree is unbalanced. The tree could become unbalanced as a result of some insertions and deletions of records.

Our first approach will be to assign a PE to each key. The PE computes the location of the key in the balanced m-way tree. This approach to the problems is similar to the approach taken by Moitra and Iyenger[13] in their algorithm for balancing a binary tree. We follow Moitra and Iyenger and allow for an extra field to be associated with each key. The content of this field is the inorder index of the key. If no inorder index exist, it can be computed using a parallel version of the MINORDER procedure from Section 2. The parallel procedure is based on the Euler path technique,[16] and can perform the operation in $O(\log n)$ time using n PEs, where n is the number of nodes in the tree. It is easy to see that assuming the availability of the inorder index, is equivalent to assuming that the keys are sorted in a nondecreasing order of key values and stored in an array. Key K_i is in location j in the array if and only if its inorder index is j. PE(i) is assigned to the key with inorder index i. The PE uses the inorder index of the key in order to compute the two-dimensional index of the node in which the key is to be stored. It is assumed that each PE knows the height of the tree and its degree. (These might be passed to the PE as procedure parameters.) We now present the algorithm:

3.1. Algorithm 1

(*The input keys can be accessed by their inorder index, i.e., by key K_j we mean the key with associated inorder index j, $1 \leq j \leq m^h - 1$.*)

Step 1. (*Find the level (r) for each node.*)

for each key K_t; $1 \leq t \leq m^h - 1$ do
find i, such that

$t \bmod m^i \neq 0$ and $t \bmod m^{(i-1)} = 0$
$r := h - i + 1$;

Step 2. (*Find the second index (q) for two-dimensional indexing of each node.*)

for each key K_t; $1 \leqslant t \leqslant m^h - 1$ do
in K_t is at level r then $q := \lfloor t/m^{(h-r+1)} + 1 \rfloor$;

Step 3. (*Find the position of key K_t in node (r, q).*)

for each key K_t; $1 \leqslant t \leqslant m^h - 1$ do
$s := \lfloor (t \bmod m^{(h-r+1)})/m^{(h-r)} \rfloor$

Step 4. (*Elements with equal r and q are grouped together in the same node.*)

for each key K_t; $1 \leqslant t \leqslant m^h - 1$ do
Assign key K_t to node (r, q) as key s of the m keys associated with the node;

Step 5. (*Compute the pointer values*)

for each node(r, q) do
create m pointers $A_0, A_1, \ldots, A_{m-1}$;
if $r = h$ then $A_j :=$ null
else $A_j :=$ node$(r + 1, (q - 1) * m + j + 1)$;

Theorem 3.1. Algorithm 1 correctly constructs the required full m-way search tree.

Proof. The correctness of the algorithm follows from the discussion in Section 2. In Step 1, PE(i) correctly computes the level r of key K_i (the key with inorder index i) in the m-way tree (Cor. 2.1). The specific node q in level r in which key K_i is stored is computed in Step 2 (Cor. 2.2). The exact position s of key K_i in node (r, q) of the search tree is computed in Step 3 (Cor. 2.3). In Step 4, PE i uses the values r, q, and s to position element K_i in its correct location in the tree. Step 4 establishes the values of the pointers. One can readily observe that the m children of node (r, q) are $(r + 1, (q - 1) * m + j)$ where $1 \leqslant j \leqslant m$.

Clearly Steps 2–4 can be executed in constant time. ach PE computes the q, r, and s values for its associated key. Since $m - 1$ PEs end up being associated with any node, it is easy to see that Step 4 can also be completed in constant time. In Step 1 PE t has to compute the level r in which the key with inorder index t will reside. To compute r PE(t) searches for a power i of m such that $t \bmod m^i \neq 0$ and $t \bmod m^{(i-1)} = 0$. This i can be found in $O(\log n)$ time by conducting a search over the possible values. The search

has to be conducted once for given tree size. The i values computed can be used in future rebalancing operations. Thus the $O(\log n)$ search time can be considered as a set-up cost. Having observed that, we can conclude that the time required for rebalancing a full m-way search tree with $m^h - 1$ keys is $O(1)$ when using $m^h - 1$ PEs.

Observe that when executing the algorithm, the PEs are only involved in computation. Each PE has all the information that it needs for computing the required values. Thus Algorithm 1 has no communication overhead. When the number of available PEs is smaller than the number of keys, we can let each PE compute the values for several keys. If only p PEs are available, where $1 \leqslant p < m^h - 1$, then we can associate $\lceil (m^h - 1)/p \rceil$ keys with each PE. The time complexity of the algorithm with only p PEs available will be then $O((m^h - 1)/p)$, not counting the set-up overhead.

We can avoid the set-up overhead altogether if we approach the problem differently. For our next algorithm we assume the same input as for Algorithm 1. This time we associate a PE with each node in the m-way balanced search tree. We now let each PE calculate the inorder indexes for keys that should reside in the node. As in the previous algorithm we consider a full m-way search tree and assume that each PE knows the height and degree of the tree.

3.2. Algorithm 2

(*Assume that there are $(m^h - 1)/(m - 1)$ PEs. The number of keys that are to be associated with this tree are $m^h - 1$ (Lemma 2.5).*)

Step 1. (*Each PE computes the two dimensional index of the node it is associated with, i.e., this is a mapping from i, the PE index, to (r, q), where $1 \leqslant i \leqslant (m^h - 1)/(m - 1)$, $1 \leqslant q \leqslant m^{(h-1)}$, $1 \leqslant r \leqslant h$.*)

```
for each PE i do
    begin
    j := ⌊log_m i⌋ (*i is the PE index*)
    if i > (m^(j+1) − 1)/(m − 1) then
        begin
        r := j + 2;
        q := i − (m^(j+1) − 1)/(m − 1);
        end
    else
        begin
        r := j + 1;
        q := i − (m^j − 1)/(m − 1);
        end
    end
```

Step 2. (*Each PE represents a node (r, q). Node (r, q) has $m-1$ inorder index associated with it. Each inorder index is associated with a unique key. The inorder index for key K_s in node (r, q) is held in X_s, where $1 \leqslant s \leqslant m-1$.*)

```
for each node (r, q) do
    begin
    for s:= 1 to m − 1 do
    X_s := ((q − 1) * m + s) * m^(h−r);
    end
```

Step 3. (*The m pointers for node (r, q) are stored at A_s, where $0 \leqslant s \leqslant m-1$.*)

```
for each node (r, q) do
    begin
    for s := 0 to m − 1 do
        if r < h then
            A_s := node(r + 1, (q − 1) * m + s + 1)
        else
            A_s := null
    end
```

In the first step of Algorithm 2 we map level labeling (Def. 2.3) to two-dimensional labeling (Def. 2.4). That is, PE i is assigned to the node with level label i in the m-way search tree. The value of i is then used in order to compute the level of the node in the tree, r, and its displacement from the left on this level, q. One can readily observe the correctness of this step.

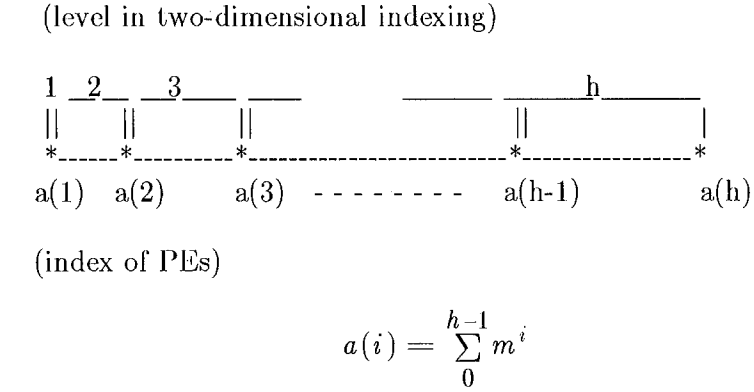

Fig. 5. The relation between level labeling and two-dimensional indexing.

There are at most $(m^h - 1)/(m - 1)$ nodes in a tree of height h (Lemma 2.2). Each PE find the height of the full tree that could be constructed using indexes that are smaller than its own index. Hence it finds the level of the node with level labeling i. Once the level of the node is known the displacement of the node within that level can be computed by subtracting the number of nodes in the full tree above the node from the level label of the node (Fig. 5).

The correctness of the other two steps in Algorithm 2 can be easily observed. Step 2 follows directly from Theorem 2.1, and Step 3 is the same as Step 5 in Algorithm 1. If one prefers to work with level labeling, then Step 3 should be (Lemma 2.3):

```
for each node i do (*i is the level label for the node*)
    begin
    for s := 0 to m − 1 do
        if r < h then
            A_s := (i − 1) * m + 2 + s
        else
            A_s := null
    end
```

Theorem 3.2. Algorithm 2 correctly constructs the required full m-way search tree.

Proof. The correctness of this theorem follows from the previous discussion.

As far as the complexity of Algorithm 2 is concerned, it should be clear that there is no set-up overhead. With $(m^h - 1)/(m - 1)$ PEs Steps 1, 2, and 3 of the algorithm can be carried in constant time. If one wants to minimize the processing time within a node (Steps 2 and 3) then up to $m - 1$ PEs can be utilized in each node. This, however, does not change the overall O(1) time complexity of our parallel algorithm.

While it is instructive to go over Algorithms 1 and 2, in practice it will be more often the case that the number of keys will not be of the form $m^h - 1$. We now consider the general case where the number of keys can be positive integer. Algorithm 3 will produce a complete m-way search tree for the given number of keys. Only the last node in the tree (greatest level label) can have less then $m - 1$ keys associated with it. We assume that each PE knows the number of keys and the degree of the search tree. These values might be passed to the PE as procedure parameters.

3.3. Algorithm 3

(*The input keys have inorder indexes in the range $[1{:}n]$, where n can be any positive integer.*)

Step 1. (*Each PE computes the two-dimensional index of the node it is associated with. This is, a mapping from i, the PE index, to (r, q), $1 \leqslant i \leqslant \lceil n/(m-1) \rceil$, $1 \leqslant q \leqslant m^{(h-1)}$, $1 \leqslant r \leqslant k$.*)

```
for each PE i do
    begin
    j := ⌊log_m i⌋ (*i is the PE index*)
    if i > (m^(j+1)) − 1)/(m − 1) then
        begin
        r := j + 2;
        q := i − (m^(j+1) − 1)/(m − 1);
        end
    else
        begin
        r := j + 1;
        q := i − (m^j − 1)/(m − 1);
        end
    end
```

Step 2. (*This computes the parameters of the tree.*)

```
for each PE i do
    begin
    h := ⌊log_m n⌋;
    if n = m^(h+1) − 1 then (*full tree*)
        (*The height of the m-way search tree is h + 1.
        It is a complete m-way search tree*)
        execute steps 2 and 3 of algorithm 2 and stop;
    c := n − (m^h − 1); (*number of keys for the last level*)
    u := c mod(m − 1); (*number of keys in the last node*)
    v := ⌊c/(m − 1)⌋; (*number of full nodes at last level*)
    if u = 0 then w := v*m − 1 else w := v*m + u;
    (*w is the inorder index of the right ost key at highest (last) level.*)
    end
```

Step 3. (*Compute the inorder indexes that are directly influenced by the extra (last) level. These will be associated with nodes in the left part of the tree.*)

```
for each node (r, q) do
    begin
    s := 1;
        loop
        temp := ((q − 1) *m + s) * m^(h − r + 1);
        if temp > w or s = m then exit;
        X_s := temp;
        s := s + 1;
        forever
    end
```

Step 4. (*Compute the inorder indexes for the rest of the tree.*)

```
for each node (r, q) with r < h + 1 do
    begin
    s := m − 1;
        loop
        temp : = ((q − 1)*m + s) * m^(h − r) + c;
        if temp ⩽ w or s = 0 then exit;
        X_s := temp;
        s := s = 1;
        forever
    end
```

Step 5. (*Compute pointer values*)

```
for each node (r, q) do
    begin
    if u = 0 then q_1 := v else q_1 := v + 1;
    (*Node h + 1, q_1) contains w.*)
    q_2 := ⌊(q_1 − 1)/m⌋ + 1; (*(h, q_2) is the parent of (h + 1, q_1)*)
    j := (q_1 − 1) mod m; (*A_j of (h, q_2) points (h + 1, q_1)*)
    for s := 0 to m − 1 do
        if r < h or (r = h and q < q_2) or (r = h and q = q_2 and i ⩽ j)
            then
                A_s := node(r + 1, (q − 1) * m + s + 1)
            else
                A_s := null;
    end.
```

Before we proceed to show the correctness of this algorithm, it is helpful to look at an example.

Example 3.1

Input: A set of n keys and their associated inorder indexes, the degree of the search tree, m. Let $n = 19$, $m = 3$.

Step 1. 10 PEs are utilized to form the nodes of the tree

PE	1	2	3	4	5	6	7	8	9	10
node	(1, 1)	(2, 1)	(2, 2)	(2, 3)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)

Step 2. $h = 2$, $c = 11$, $u = 1$, $v = 5$, $w = 16$.

Step 3. The set of elements of node (r, q) is $\{t = (3 * (q-1) + s) * 3^{(3-r)}$, where $t \leqslant 16$ and $s = 1$ or $2\}$.

PE i	node (r, q)	content (inorder indexes) X_1	X_2
0	(1, 1)	9	—
2	(2, 1)	3	6
3	(2, 2)	12	15
4	(2, 3)	—	—
5	(3, 1)	1	2
6	(3,2)	4	5
7	(3, 3)	7	8
8	(3, 4)	10	11
9	(3, 5)	13	14
10	(3,6)	16	—

Step 4. The set of elements for node (r, q) $r < 3$ is $\{t = (3 * (q+1) + s) * 3^{(2-r)} + 11$, where $t > 16$ and $s = 1$ or $2\}$.

PE i	node (r, q)	content (inorder indexes) X_1	X_2
1	(1, 1)	9	17*
2	(2, 1)	3	6
3	(2, 2)	12	15
4	(2, 3)	18*	19*

* was assigned value in this step

Step 5. $q_1 = 6$, $q_2 = 2$, $j = 2$.

PE i	node (r, q)	pointers (children) A_0	A_1	A_2
1	(1, 1)	(2, 1)	(2, 2)	(2, 3)
2	(2, 1)	(3, 1)	(3, 2)	(3, 3)
3	(2, 2)	(3, 4)	(3, 5)	(3, 6)
4	(2, 3)	null	null	null
5	(3, 1)	null	null	null
6	(3, 2)	null	null	null
7	(3, 3)	null	null	null
8	(3, 4)	null	null	null
9	(3, 5)	null	null	null
10	(3, 6)	null	null	null

In Fig. 6 we show the 3-way complete search tree for the example.

While Algorithm 3 seems complicated, it is easy to show its correctness.

Lemma 3.1. The following are true for a complete m-way search tree:

(i) The right most node of level $h+1$ is node$(h+1, v)$ when $u = 0$ and node$(h+1, v+1)$ otherwise.

(ii) The last element of the right most node at level $h+1$ is w.

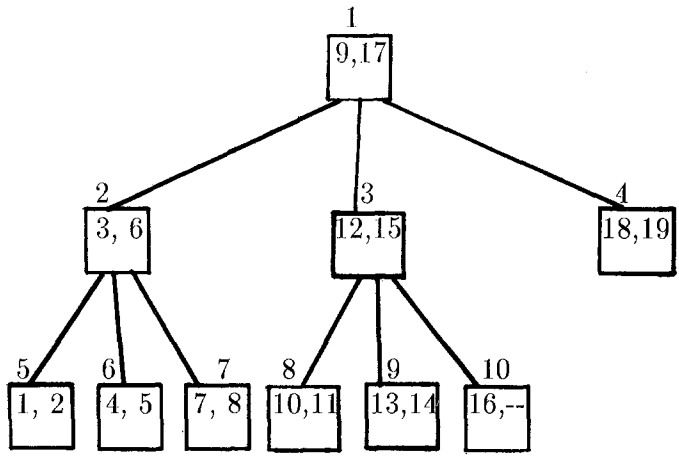

Fig. 6. The 3-way complete tree for Example 3.1.

Proof. (i) For the resulting tree to be complete, it has to accommodate c keys in its last level $h+1$. When c is a multiple of $m-1$ ($u=0$), v nodes at level $h+1$ are sufficient. When c is not an exact multiple of $m-1$, an extra node $(h+1, v+1)$ is required. Node $(h+1, v+1)$ will be only partially full, i.e., it will have less then m-1 keys associated with it. (ii) This follows immediately from Corollary 2.4.

Lemma 3.2. Steps 3, 4 and 5 of algorithm 3 correctly assign the values of indexes associated with nodes and the value of pointers among the nodes.

Proof. Observe that key with inorder index i, where $i \leqslant w$ should be assigned using the construction for an m-way tree of height $h+1$. Only these keys are candidates for level $h+1$ (Lemma 2.8). These keys are assigned to nodes in Step 3. Notice also that keys with inorder index i, $i > w$ are candidates for levels 1 through h in the m-way tree. The index for these keys should be biased in order to compensate for the keys that were assigned in step 3. One can easily show that this bias should be c. That is, the keys with inorder index i, $i > w$, should be assigned as keys with inorder index i-c in a full m-way tree with h levels. This is done in Step 4 of the algorithm. It follows that the assignment of keys to nodes is performed correctly.

As for the correctness of Step 5, assignment of pointers among the nodes, only the case where $r = h$, needs to be justified. But in this case, we only need to find the pointer A_j of node (h, q) such that A_j points to the node containing the key with inorder index w. By the definition of a complete tree, all pointers to the right of this pointer should be set to null. This is exactly what is done in the algorithm.

Theorem 3.3. Algorithm 3 correctly constructs the required complete m-way search tree.

Proof. The correctness of this theorem follows from the previous discussion.

As far as the time complexity of Algorithm 3 is concerned, the mapping of level labeling to two-dimensional indexing (Step 1) can be performed in constant time. The number of nodes in the tree and, correspondingly the maximal number of PEs that can be effectively utilized for Step 1 is $\lceil n/(m-1) \rceil$, where n is the number of keys (length of input). Next, each PE computes the values of c, u, v, and w. This can be done in constant time using the same number of PEs. With one PE assigned to a node, Steps 3–5 can be performed in $O(m)$ time each. Since m is fixed, we can conclude that the overall time complexity of Algorithm 3 is $O(1)$. As in

the previous algorithm, up to $m-1$ PEs can be utilized in each node (for Steps 3–5). These additional PEs will not change the overall time complexity of the algorithm.

When algorithm 3 available to rebalance the tree periodically, we can allow the insertion and deletion operation to leave the tree unbalanced. The INSERT (DELETE) procedure will use SEARCH to identify the point of insertion (deletion) and insert (delete) the key at that point. Obviously, the m-way search tree will become unbalanced after a few such INSERT and DELETE operations. Algorithm 3 can then be utilized to rebalance the tree.

In an environment where insertion and deletion are not common, it is more efficient to insert or delete keys while maintaining the tree balanced. To do that we need to transform a balanced m-way search tree with n keys to a balanced m-way search tree with $n+1$ or $n-1$ keys. We consider a transformation from n to $n-1$ keys (a delete, Algorithm 4), and from n to $n+1$ keys (an insert, Algorithm 5). Each of the transformations described here is simpler than the operation of rebalancing the whole tree described earlier in this section. Observe that after a direct insertion or deletion we need to update the structure and the content of the complete m-way search tree.

In both Algorithms we modify first the structure to reflect the change in the number of keys, and then move the keys into their correct locations in the new balanced tree. We assume that the following parameters are kept with the data structure:

n—the number of keys in the tree.
u—the number of keys in the last node of the highest level.
v—the number of full nodes in the highest level.
w—the inorder index of the last key in the last node.
$h+1$—the height of the tree.

3.4. Algorithm 4

(*Insert key X into a balanced m-way search tree of height $h+1$. The tree remains balanced after the insertion.*)

Step 1. (*This transforms a given complete m-way search tree with n keys to an m-way search tree with $n+1$ keys.*)

$n := n+1;$
$u := (u+1) \bmod (m-1);$
if $u = 1$ then
begin
(*A new node is required.*)

```
        if n = m^(h+1)
            begin
            (*The new node is at a new level.*)
            h := h + 1
            v := 0;
            w := 1;
            end
        else
            begin
            v := v + 1
            w := w + 2;
            end
        create a new node(h + 1, v + 1) that contains only
        one key, this key has inorder index w;
        j := v mod m;
        A_j of node (h, ⌊v/m⌋ + 1) := node(h + 1, v + 1);
        end
    else
        begin
        w := w + 1;
        just add the key with inorder index w to node(h + 1, v + 1);
        end
```

Step 2. (*Reset the indexes effected by the increase in number of keys.*)

```
        for each node(r, q) do
        for each inorder index t associated with node(r, q) do
            if t > w then t := t + 1;
```

Step 3. (*Insert key X.*)

```
        for all keys K_i with inorder index i ≤ w and K_i > X do
            (*Assume K_0 = −∞ and K_w = ∞*)
            if K_(i−1) > X
                K_i := K_(i−1)
            else
                K_i := X;
        for all keys K_i with inorder index i ≥ w and K_i ≤ X do
            (*Assume K_(n+1) = ∞.*)
            if K_(i+1) < X
                K_i := K_(i+1)
            else
                K_i := X;
```

3.5. Algorithm 5

(*Delete key X from a balanced m-way search tree of height $h+1$. The tree remains balanced after the deltion.*)

Step 1. (*Delete key X from the tree*)

```
for all keys K_i do (*i is the inorder index of the key*)
    if K_i ⩾ X then K_i := K_{i+1};
```

Step 2. (*This transforms a given complete m-way search tree with n keys to an m-way search tree with $n-1$ keys.*)

```
n := n − 1;
u := (u − 1) mod(m − 1);
if u = 0 then
    begin
    if n = m^h − 1 then
        (*The only node at level h + 1 should be deleted.*)
        begin
        h := h − 1;
        v := m^h
        w := n;
        delete node(h + 2, 1);
        A_0 of node(h + 1, 1) := null;
        end
    else
        begin
        w := w − 2;
        v := v − 1;
        delete node(h + 1, v + 2);
        j := (v + 1) mod m;
        A_j of node(h, ⌊(v + 1)/m⌋ + 1) := null;
        end
    end
else
    begin
    just delete w from the right most node at level h + 1;
    w := w − 1;
    end
```

Step 3. (*Update inorder indexes effected by the decrease in number of keys*)

for each node(r, q) do
    for each inorder index t associated with node(r, q) do
        if $t > w$ then $t := t - 1$;

The correctness of these algorithms can be easily shown. The time complexity of these algorithms is O(1). In Fig. 7 we show the tree of Example 3.1 after 2 insertion operations. The values of the variables n, u, v, w, and h before the first insertion are 19, 1, 5, 16 respctively. After the first insertion the value of these variables would be 20, $0 = 2 \bmod (3-1)$, 5, 17 respectively, and after the second insertion their values would be 21, 1, 6, 19.

CONCLUSION

In Section 3 we presented optimal parallel algorithms for rebalancing or constructing belanced m-way search trees. If n keys are to be associated with the tree then the construction can be carried out in O(1) time using O(n) PEs. While our algorithm is more general then Moitra and Iyengar's binary tree algorithm,[13] we can compare the two when $m = 2$. For this case our algorithm is more efficient since it does not require any set-up overhead. Notice also that the complexity of our algorithm is independent of the degree of the tree, m. Hence m can be chosen to fit best with the external storage hardware characteristics.

The problem of constructing a balanced m-way search tree was chosen to demonstrate a parallel algorithm where communicating overhead is

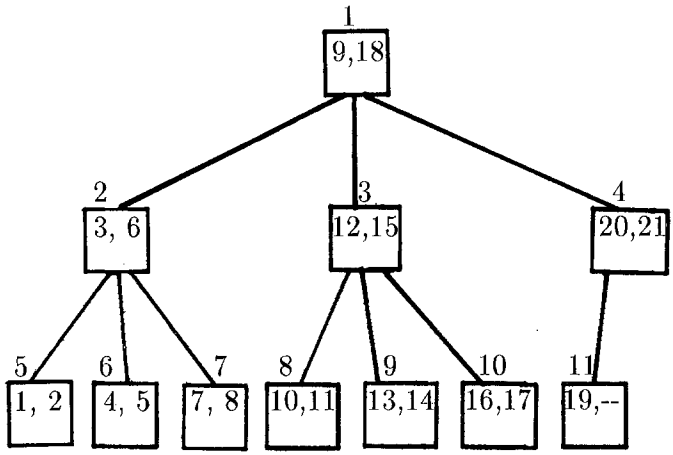

Fig. 7. The 3-way search tree of Fig. 6, after 2 insertions.

completely eliminated. While in Section 3 we treated the problem from the "Design of Algorithm" point of view, it is important to consider the environment in which such algorithms can be useful.

In this context, let us examine the basic distionary operation (SEARCH, INSERT, AND DELETE). Using straightforward information-theoretic arguments, one can show that at least $\log_k n$ parallel steps are required for searching a sorted array of n elements with k PEs. SEARCH is required as an initial operation for both INSERT and DELETE. It is clear that once a location for insertion (deletion) is found, the insertion of an element can be done in constant time. Hence the complexity of insertion is bounded below by the complexity of searching.

Consider the case where the dictionary information fits in internal memory. Using a "fan in" argument, we can see that there is no advantage to using more than one PE for a search operations. This argument is based on the practical assumption that a PE can send or receive information concurrently from only a fixed number of ports. In our analysis we assume that only one communication port can be active at a time. Assume that we have k PEs and we need to search for a specific element in an ordered set of size n. We will need $O(\log k)$ time to transmit the key for the search to the k PEs. As observed, the search can be conducted in $\log_k n$ parallel steps. After each step the search location for the search step is transmitted among the PEs. Hence each search step will require $O(\log k)$ communication overhead. Thus the overall time complexity of a search is $(\log_2 k) * (\log_k n) = \log_2 n$. Since this search can be conducted using binary search and only one PE in $O(\log n)$ time, the argument follows. Notice that this analysis provides a lower bound for any data structure or number of PEs. The observation made for the case of one PE is the only one that assumed ordered keys.

Having established the $O(\log n)$ lower bound, it is not surprising that all the spcial purpose architectures have this time complexity for searching no matter how many PEs they use, see Refs. 3–10, 14. While those solutions achieve the lower bound complxity, it was observed in Ref. 11 that they are "processor-profligate." Most of these architectures use $\theta(n)$ PEs to achieve only an $O(\log n)$ throughput improvement over the serial balanced tree algorithm.

When the dictionary is stored in external memory, the optimization criteria are different. The storage structure is chosen so that the number of I/O operations are minimized. An m-way search tree is a popular choice. The degree m is selected to fit the physical characteristics of the external storage.[2]

These observations can be translated quite effectively to practice in our MIMD environment. The system can initiate any number of searches in a

"pipelined" fashion. Each search is conducted using only one PE leaving one machine cycle between consecutive requests. Search results can be obtained in a pipeline interval of O(1). While some PEs are conducting searches, other PEs are free to perform other tasks.

Our solution is applicable for a general purpose machine environment. The m-way search tree is kept in external storage. At any time k PEs are available, where $0 \leqslant k \leqslant P$ (P is the maximal number of PEs available on the a machine.). In such a machine the operating system can be instructed to allocate only one processor for a search operation and as many PEs as available or required (whichever is the minimum), in case a new tree has to be constructed or an existing tree rebalanced.

REFERENCES

1. D. E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching. Addison-Wesley, reading, Mass. (1973).
2. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Potomac, Md. (1982).
3. P. K. Armstrong, U. S. Patent 4131947 (December 26, 1978).
4. M. J. Attallah and S. R. Kosaraju, A generalized dictionary machine for VLSI, *IEEE Trans. on Comput.* **C-34**(2):151–155 (February 1985).
5. J. L. Bentley and H. T. Kung, Two papers on tree-structured paralel computer, Dep. Comput. Sci. Carnegie Mellon University, Pittsburge, PA, Report CMU-CS-79-142 (1979).
6. M. J. Carey and C. D. Thompson, An efficient implementation of search trees on $\lceil \log N + 1 \rceil$ processors, *IEEE Trans. on Comput.* **C-33**(11):1038–1041.
7. C. E. Leiserson, Systolic priority queues, Dep. Comput. Sci. Carnegie Mellon University, Pittsburge, PA, Report CMU-CS-79-115 (1979).
8. T. A. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, A dictionary machine (for VLSI), *IEEE Trans. on Comput.* **C-31**:892–897 (September 1982).
9. A. K. Somani and V. K. Agarwal, An efficient VLSI dictionary machine, *Proc. 11th Annu. ACM Intl. Symp. on Comput. Arch.*, pp. 142–150 (June 1984).
10. Y. Tanaka, Y. Nozaka, and A. Masuyama, Pipeline searching and sorting modules as components of data flow database computer, *Proc. Int. Fed. Inform. Processing*, pp. 427–432 (October 1980).
11. A. L. Fisher, Dictionary Machines with a small number of processors, *Proc. 11th Annu. ACM Int. Symp. on Comput. Arch.*, pp. 151–156 (June 1984).
12. H. Chang and S. S. Iyengar, Efficient algorithms to globally balance a binary search tree, *Com. ACM* **27**:695–702 (1984).
13. A. Moitra and S. S. Iyengar, A maximally parallel balancing algorithm for obtaining complete balanced binary trees, *IEEE–T-SE*, pp. 442–449 (1986).
14. Q. F. Stout and B. L. Warren, Tree rebalancing in optimal time and space, U. of Michigan Computing Research Laboratory, Ann Arbor, MI, CRL-TR-42-84.
15. U. Manber, Concurrent Maintenance of Binary Search Trees, *IEEE Trans. on Soft. Engineering* **SE-10**(6):777–784 (November 1984).
16. R. E. Tarjan and U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time *FOCS* (1984).