MAINTAINING B-TREES ON AN EREW PRAM*

LISA HIGHAM AND ERIC SCHENK

Efficient and practical algorithms for maintaining general B trees on an EREW PRAM are presented. Given a B tree of order b with m distinct records, the search (respectively, insert and delete) problem for n input keys is solved on an n processor EREW PRAM in $O(\log n + \log_b m)$ (respectively $O(b(\log n + \log_b m))$ and $O(b^2(\log_b n + \log_b m))$) time.

1 Introduction

To make the fullest use of large scale synchronous parallel machines, algorithms that efficiently maintain search trees in parallel are required. This paper presents algorithms for maintaining B trees on an n processor exclusive read exclusive write parallel random access machine (EREW PRAM).

A PRAM is a collection of independent synchronized sequential processors with unique identifiers and a shared global memory. In each time step, each processor can read a location in the global memory, perform a local computation, and then write to a location in the global memory. In an EREW PRAM no two processors may simultaneously access the same memory location for either reading or writing. Karp and Ramachandran [7] provide an overview of PRAM results.

A B tree of order b is a search tree satisfying: (1) every node has at most b children; (2) every node except the root and the leaves has at least $b/2$ children; (3) a non-leaf node with l keys has $l+1$ children; and (4) all leaves are at the same level of the tree. There are two common variants of the B-tree structure. In a general B tree, keys are stored in all nodes; in a B$^+$ tree, keys are stored only in the leaves. Comer [4] provides a general review of the B tree literature.

Throughout this paper the number of keys in the B-tree is denoted by m and the number of PRAM processors by n. Given keys $k_1, \ldots, k_n$ and a B tree B, we define three problems.

The n Key Search Problem. For each i, $1 \le i \le n$, find the node v of B such that: (1) if k_i is in B then k_i is in v, and (2) if k_i is not in B then v is the leaf of B at which the insertion of k_i would take place.

The n Key Insertion Problem. Modify B to a new B-tree whose keys are the keys of B union $\{k_1, \ldots, k_n\}$.

The n Key Deletion Problem. Modify B to a new B tree whose keys are the keys of B minus $\{k_1, \ldots, k_n\}$.

*This research was supported in part by a research grant and a postgraduate scholarship both from the Natural Sciences and Engineering Research Council of Canada.

The problem of allowing multiple asynchronous processes to concurrently access and update B trees has been addressed in the concurrent database literature. See for example: Kung and Lehman [9], Lehman and Yao [12], Sagiv [14], Lanin and Shasha [11], Shasha and Goodman [16] and Wedekind and Wang [20] as well as some earlier papers [2, 3, 15]. The approach taken in all of these papers is to allow each process to apply read and write locks to B-tree nodes to prevent conflicting actions that would lead to incorrect results. In the asynchronous setting the problem is complicated by the necessity of maintaining a searchable structure at all times despite conflicting actions that may be introduced by updates that are in progress. Therefore the emphasis in these papers is to guarantee correctness in the presence of asynchrony. However, the techniques presented in these papers are insufficient to guarantee high concurrency when the number of updating processes is large. Rather it is easily confirmed that in the worst case these algorithms require $O(n)$ steps to complete n simultaneous parallel insertions or deletions. Weihl and Wang [20] provide a framework for concurrent dictionary algorithms that gives high concurrency for reads, even in the presence of multiple updates. However the cost of pure updates can still be made linear. In contrast the techniques of this paper exploit synchrony to solve the aggregate problems of n simultaneous searches, insertions or deletions in a cooperative manner, while avoiding locks entirely. Since only one of these three problems will be solved at a time, further gains in efficiency are made by abandoning attempts to maintain a searchable structure during the execution of the insertion and deletion algorithms.

There has been less research on maintenance of B trees in a synchronous environment. Wang and Chen [18] consider the problem of constructing a n node B tree of order 1 (a *2-3 tree*) from a sorted array of n data elements. Wang, Chen and Yu [19] extend this construction to general B trees. They describe an $n/\log\log n$ processor CREW (concurrent read exclusive write) PRAM algorithm that takes $O(\log\log n)$ time, as well as an n processor CREW PRAM algorithm that takes $O(1)$ time. Their algorithm can be used to solve the n node insertion problem on an m node B tree in $O(\log m + \log n)$ time on an $(m+n)$-processor EREW PRAM by reconstructing the tree from scratch. (The extra time and processors are required to sort). Shibayama [17] gives an algorithm that merges two B$^+$ trees of order 3 (*2-3$^+$ trees*) with sizes m_1 and m_2 with $m_1 \le m_2$ into a single tree in $O(\log m_1 + \log m_2)$ time on an m_1 processor EREW PRAM. Paul, *et al* [13] present a processor

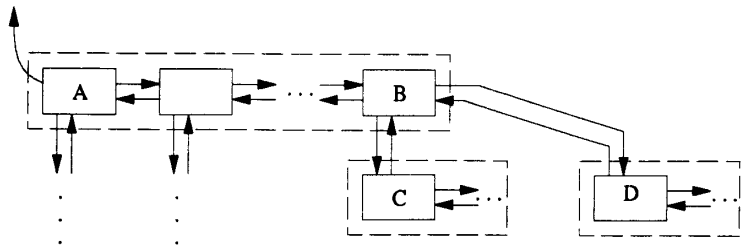

FIGURE 1. Representation of a B-tree node. Nodes are outlined with a dotted line. Records A, C and D, have the *head* flag set to true, record *B* has the *tail* flag set to true, and record *D* has the *last* flag set to true.

EREW PRAM algorithms for the n key search, insertion and deletion problems on a 2-3$^+$ tree. Each of their algorithms has time complexity $O(\log n + \log m)$.

This paper presents n-processor EREW PRAM algorithms for the n key search, insertion and deletion problems for general B-trees of order b that achieve $O(\log n + b \log_b m)$ time for searching, $O(b(\log n + \log_b m))$ time for insertion, and $O(b^2(\log_b n + \log_b m))$ time for deletion. While our algorithms share some similarities with those of Paul, *et. al*, and achieve the same time complexity for constant b, they differ because our algorithms

(1) apply to standard B-trees (rather than B$^+$-trees),
(2) apply to B-trees of any order (rather than only order 3),
(3) require only $O(n)$ work space (rather than $\Theta(n \log n)$).

Our algorithms rely upon implementation details and input preprocessing outlined in section 2. Sections 3, 4 and 5 present, respectively, the search, insertion and deletion algorithms. Section 6 briefly summarizes the new contributions of this paper and adds some additional observations.

2. B-TREE REPRESENTATION AND INPUT PREPROCESSING

Throughout this paper, b denotes the order of the B-tree, and L denotes the maximum level, with the leaves at level 0 and the root at level L. A B-tree node v with keys c_1 through c_l and pointers q_0 through q_l is instantiated as a doubly-linked list of l records, r_1 through r_l. Each record r_i in this list contains the key c_i (stored in r_i.key), three flags (r_i.head, r_i.tail and r_i.last) and four pointers (r_i.data, r_i.child, r_i.pred and r_i.succ). Figure 1 illustrates this structure. Pointer r_i.data points to the data in global memory associated with key c_i. It is assumed that this field is set whenever r_i.key is set and is henceforth ignored. Pointer r_i.child points to the first record of the node pointed at by q_{i-1}. For $i > 1$, r_i.pred points to the record for key c_{i-1}. Pointer r_1.pred points to the record corresponding to the parent of node v. For $i < l$, r_i.succ points to the record for key c_{i+1}. Pointer r_l.succ points to the first record of the node pointed at by q_l. The flag r_i.head (respectively, r_i.tail) is set if and only if r_i is the first (respectively, last) record of a B-tree node. The last record of a node has two children that may compete for concurrent access to the parent record. The *last* flag is added to arbitrate control between two such children. Hence the flag r_i.last is set if and only if r_i is the head of node v and v is the second child of the last record of the parent of v. Each record also contains a scratch field used by our algorithms for marking and communication.

With just a reinterpretation of pointers but no change in structure, this representation of a B-tree with L levels converts to a binary tree with depth at most bL. When we wish to emphasize this interpretation we will use the names r_i.leftchild, r_i.rightchild and r_i.parent as synonyms for r_i.child, r_i.succ and r_i.pred respectively. Our search algorithm heavily exploits this correspondence.

By preprocessing with Cole's Merge Sort [3] to sort the input keys, and then with prefix sums [10] to remove duplicates, we can assume that input keys are sorted and distinct. This costs $O(\log n)$ time on an n-processor EREW PRAM. This is a non-restrictive assumption because our algorithms use $O(\log n + \log m)$ time and n-processors for B-trees with constant order b. Therefore, throughout the paper $k_1, \ldots, k_n$ denotes the sorted sequence of distinct inputs.

At the beginning of insertion, each processor is allocated one B-tree record, with the structure described above, to store the key it is inserting. Similarly, at the end of deletion, each processor returns the record used to store its deleted key to the memory pool. This is the only memory management required by our B-tree maintenance algorithms. The necessary allocations or deallocations can be performed by an $n / \log n$-processor EREW PRAM in $O(\log n)$ time using the method of Higham and Schenk [6].

3. SEARCH

3.1. Informal Overview of the Algorithm. Given the B-tree representation described in section 2, the search problem becomes a search of a binary tree of records.

The search algorithm has three stages. As soon as a processor completes a stage it continues with the next one. In stage 1 a subset of processors determine the required search result for packets of keys. In stage 2, broadcast is used to inform each remaining processor s of the search result for key k_s. In stage 3 synchronization is used to ensure a clean termination. Stage 1 does the major work of the algorithm and proceeds in rounds as follows. Initially, the sorted list of search keys forms a single packet residing at the root of the search tree.

Subsequently, each packet travels down the tree to its search destination splitting en-route whenever keys in the packet follow different paths. At each step a simple computation assigns one of four labels to each packet. It will be proved that at any step, for any node, there resides at most one packet of each label type. Hence, by scheduling one time slot for each label type, simultaneous access to a record is avoided.

Three arrays, POSITION, DIRECTION and FINISHED, each of size n, are used. If the search for key k_i is successful, POSITION[i] points to the record containing key k_i, otherwise it points to the leaf record of the binary tree where the search terminated. In the first case, DIRECTION[i] is set to `found`. Otherwise it is set the `left` or `right` depending on whether k_i is less than or greater than the key of POSITION[i]. The array FINISHED is used to coordinate termination.

3.2. Formal Description. Define a *packet* of keys, denoted pkt[$i..j$], to be the subsequence of keys k_i through k_j inclusive. Define a *labeling* function $\lambda : \{1,2,3,4\} \rightarrow \{1,2,3,4\}$ by:

$$\begin{aligned} \lambda(1) &= 1, \\ \lambda(2) &= 4, \\ \lambda(3) &= 1, \text{ and} \\ \lambda(4) &= 4. \end{aligned}$$

All actions ascribed to a packet pkt[$i..j$] are performed by the i^{th} processor. We say that processor i *manages* packet pkt[$i..j$].

(1) **(Find)** Stage 1 proceeds in rounds and is described inductively. At the beginning of round 1, pkt[$1..n$] has *label* 1 and *resides* at the root of the tree. Suppose that at the beginning of round t some pkt[$s..s+l$] resides at record r. Then either
   (a) $k_{s+l} < r.\text{key}$ in which case the search for all keys in pkt[$s..s+l$] should continue in the left subtree of r,
   (b) $k_s > r.\text{key}$ in which case the search for all keys in pkt[$s..s+l$] should continue in the right subtree of r, or
   (c) $k_s \leq r.\text{key} \leq k_{s+l}$ in which case pkt[$s..s+l$] is said to *straddle* r.

   In case (a) (respectively, case (b)), if the left (respectively, right) subtree of r is non-empty, then in round t processor s changes the residence of pkt[$s..s+l$] to r.leftchild (respectively, r.rightchild), and updates the label of pkt[$s..s+l$] by applying the labeling function λ to its current label. If the left (respectively, right) subtree is empty then processor s sets DIRECTION[s] to `left` (respectively, `right`), sets POSITION[s] to r, and is then finished stage 1. In case (c), if $l \geq 1$ then pkt[$s..s+l$] is split into two approximately equal packets: pkt[$s..s+\lfloor l/2 \rfloor$] with label 2 and pkt[$s+\lfloor l/2 \rfloor + 1..s+l$] with label 3. Processor s alerts processor $s+\lfloor l/2 \rfloor + 1$ to manage pkt[$s+\lfloor l/2 \rfloor + 1..s+l$] in round $t+1$. Each packet remains resident at record r. If $l = 0$ then processor s sets DIRECTION[s] to `found`, sets POSITION[s] to r, and is then finished stage 1. To avoid read conflicts each round of stage 1 is executed in four steps. A processor managing a packet with label i executes in step i only.

(2) **(Broadcast)** Suppose processor s manages pkt[$s..s+l$] at the beginning of stage 2. Then POSITION[s] and DIRECTION[s] contain the search results for all keys in pkt[$s..s+l$]. Using a standard broadcast mechanism POSITION[s] and DIRECTION[s] are copied into POSITION[i] and DIRECTION[i] for every i such that $s < i \leq s+l$ as follows.
   Say that processor s manages some packet pkt[$s..s+l$] at round t of stage 2. If $l = 0$ then processor s is finished stage 2. Otherwise, s splits the packet pkt[$s..s+l$] into two approximately equal packets pkt[$s..s+\lfloor l/2 \rfloor$] and pkt[$s+\lfloor l/2 \rfloor + 1..s+l$], notifies processor $s+\lfloor l/2 \rfloor + 1$ to manage pkt[$s+\lfloor l/2 \rfloor + 1..s+l$] in round $t+1$ of stage 2, and copies DIRECTION[s] into DIRECTION[$s+\lfloor l/2 \rfloor + 1$], and POSITION[$s$] into POSITION[$s+\lfloor l/2 \rfloor + 1$].

(3) **(Termination)** Various processors typically complete stages 1 and 2 at different times. A simple synchronization mechanism is employed to ensure the simultaneous termination of all processors, even when the size of the B-tree is not known. Specifically, processor s marks location s of a global FINISHED array only when it has completed stage 2 and processors $2s$ and $2s+1$, if they exist, have marked their corresponding positions in the FINISHED array. The algorithm terminates (using a broadcast) when FINISHED[1] is marked.

3.3. Correctness. Given that the search algorithm is free of read conflicts, correctness for an EREW PRAM would follow directly from the following easily established claims.

Claim 3.1. *If a processor s completes stage 1 managing pkt[$s..s+l$] for some $l \geq 0$, then* POSITION[s] *and* DIRECTION[s] *contain the required search result for all keys in pkt[$s..s+l$].*

Claim 3.2. *Every process s completes stage 2, and upon completion* POSITION[s] *and* DIRECTION[s] *contain the required search result for key k_s.*

Claim 3.1 follows from the well established correctness of the usual binary tree search algorithm. Claims 3.2 is immediate from the algorithm.

Stages 2 and 3 are clearly free of concurrent reads and writes. Stage 1 has no concurrent writes. It remains to be shown that stage 1 is free of read conflicts. The following labeling scheme is easily extracted from the algorithm.

(1) Initially, pkt[$1..n$] is labelled 1.
(2) If pkt[$a..d$] straddles a record in round t then in round $t+1$ the new packets pkt[$a..\lfloor (a+d)/2 \rfloor$] and pkt[$\lfloor (a+d)/2 \rfloor + 1..d$] are labelled 2 and 3 respectively.
(3) If a packet descends one level of the binary tree in round t, then

(a) a packet labelled 1 retains label 1.
(b) a packet labelled 2 is relabelled 4.
(c) a packet labelled 3 is relabelled 1.
(d) a packet labelled 4 retains label 4.

The algorithm schedules four steps per round such that a processor with label i executes in step i of the round. To ensure that the n-key search algorithm with this labeling and scheduling is correct for an EREW PRAM, it suffices to establish the following theorem.

Theorem 3.3. *For each round of stage 1, and for each record r, the packets residing at r have distinct labels chosen from* $\{1, 2, 3, 4\}$.

The notation $\text{pkt}[a..b] < \text{pkt}[c..d]$ denotes that the largest key in $\text{pkt}[a..b]$, k_b, is less than the smallest key in $\text{pkt}[c..d]$, k_c. A packet $\text{pkt}[a..b]$ *left-bounds* (respectively, *right-bounds*) a collection of packets if for every $\text{pkt}[c..d]$ in the collection, either $\text{pkt}[a..b] = \text{pkt}[c..d]$ or $\text{pkt}[a..b] < \text{pkt}[c..d]$ (respectively, $\text{pkt}[a..b] > \text{pkt}[c..d]$). The following lemma is used in the proof of Theorem 3.3.

Lemma 3.4. *Any packet labelled* 1 *(respectively,* 4*) residing at record r left-bounds (respectively, right-bounds) the set of packets in the subtree rooted at r.*

Proof. The proof is given for packets with label 1. The proof for packets labelled 4 follows the same reasoning. According to the algorithm, a packet labelled 1 in round t was labelled either 1 or 3 in round $t-1$. Proof is by induction on the number of rounds that a fixed packet has label 1. For the basis, suppose that in round t_0 packet $\text{pkt}[c..d]$, residing at record r, is first labelled 1. Then either $t_0 = 1$ or in round $t_0 - 1$ $\text{pkt}[c..d]$ was labelled 3 and resided at p, the parent of r. If $t_0 = 1$ then the lemma follows trivially. Otherwise, it follows from the algorithm that in round $t_0 - 1$:

(1) there was also a packet $\text{pkt}[a..b]$ labelled 2 residing at p, and
(2) $\text{pkt}[a..b]$ and $\text{pkt}[c..d]$ were formed by splitting $\text{pkt}[a..d]$, and
(3) $a \le p.key$, and
(4) $p.key < c$.

Thus, there is no search key k outside of $\text{pkt}[a..b]$ satisfying $p.key < k < c$. So $\text{pkt}[c..d]$ left-bounds the set of packets in the subtree rooted at r. For the inductive step, suppose that at round t $\text{pkt}[a..b]$ residing at r and labelled 1 left-bounds the packets in the subtree rooted at r. Then $\text{pkt}[a..b]$ left-bounds both the packets in the left subtree and the packets in the right subtree of r. So if $\text{pkt}[a..b]$ descends a level to record q it left-bounds the set of packets in the subtree rooted at q, where q is either child of r. If $\text{pkt}[a..b]$ does not descend then it is split in two and the two new packets are labelled 2 and 3. ■

Proof of Theorem 3.3. Suppose the theorem is false. Let t be the first round in which two packets with the same label reside at the same record, and suppose r is one such record. If the label is 2, then in round $t-1$ two packets must have resided at r each straddling $r.key$. Since packets never overlap, this is impossible. The same argument applies to label 3. If the label is 1, then there must have been one packet labelled 1 and one labelled 3 both of which descend from p, the parent of r, to r in round $t-1$. The existence of a packet $\text{pkt}[c..d]$ with label 3 implies the simultaneous existence of a packet $\text{pkt}[a..b]$ with label 2 also residing at p. Since packet $\text{pkt}[a..d]$ straddled and $\text{pkt}[c..d]$ descended, it must be that $a \le p.key < c$. But by Lemma 3.4, in round $t-1$, the packet with label 1 left-bounds the packets in the subtree rooted at p. Thus the packet labelled 1 must have moved to the left subtree in round $t-1$ and the packet labelled 3 must have moved to the right subtree in round $t-1$. So they could not have descended to the same record. The same type of argument applies to packets labelled 4. ■

3.4. Complexity. At each round in stage 1, each packet either splits into two halves or moves down to a child record. Thus, all processors must complete stage 1 after at most $\log n + (b-1)\log_{b/2} m$ rounds. Each round uses four steps and each step takes constant time. After stage 1 there are a further $\log n$ steps for each of stages 2 and 3. Thus the algorithm completes in $O(\log n + b \log_b m)$ time.

The results of this section are summarized by the following theorem.

Theorem 3.5. *The n-Key Search Problem for an order b B-tree containing m nodes can be solved on an n-processor EREW PRAM in* $O(\log n + b \log_b m)$ *time.*

Notice that the search algorithm determines the required node in the B-tree and the exact location within the node where an absent record would be inserted.

4. INSERTION

4.1. Informal Overview of the Algorithm. The algorithm has four stages. Stage one applies the search algorithm to find the insertion point for each record. Stage two arranges each packet of records that share a common insertion point into a balanced binary tree. The major work of the insertion algorithm is accomplished in stage 3, which proceeds in rounds. In each round, the root of each binary tree is inserted into a leaf node thus creating distinct insertion points for the roots of each subtree. Each node that exceeds the size limit of $b-1$ after the record insertions is reconfigured into a root node with one record and two children, and the root is scheduled for insertion into the next level of the tree in the next round. To avoid simultaneous access of records, the progressive insertions of the roots of the binary insertion trees are separated by 2 rounds. Stage 4 ensures a clean termination.

4.2. Formal Description.

(1) **(Search)** Apply the search algorithm of section 3. We assume that no insertion key is already in the B-tree. (Otherwise, follow the search stage with a constant time step to do the updates, and a logarithmic time parallel prefix operation to compact the remaining insertion keys.) Each processor p records key k_p in the record r_p, which is allocated as discussed in section 2. The pointer fields r_p.child and r_p.succ are initialized to **null**. The pointer r_p.pred and the flags r_p.head, r_p.tail and r_p.last will be initialized during remaining steps of the algorithm.

(2) **(Insertion scheduling)** Define an *insertion interval* $I[i,j]$ to be the maximum interval $[i,j]$ such that the insertion points determined by search for keys k_i through k_j are identical. That is, $I[i,j]$ is the maximal interval such that for all p, q in $I[i,j]$, POSITION$[p]$ = POSITION$[q]$, and DIRECTION$[p]$ = DIRECTION$[q]$. All processors determine if they are an endpoint of an insertion interval by examining the POSITION and DIRECTION arrays. Processors at the head and tail of an insertion interval communicate through the scratch field of the record at their insertion point to determine the insertion interval. Consider any insertion interval $I[i,j]$. The processors with indices in $[i,j]$ proceed inductively, as follows, to arrange the records r_i through r_j inclusive into a balanced binary search tree called an *insertion tree* and denoted $T[i,j]$. If DIRECTION$[i]$ = `left` (respectively, `right`), then initially processor i *manages* processors in the interval $[i,j]$ to construct a left (respectively, right) subtree of the record, say $r_{i,j}$, pointed to by POSITION$[i]$. Let $z = \lceil (i+j)/2 \rceil$. Processor i alerts processor z to doubly link (using the binary tree interpretation) record r_z as a left (respectively, right) child of $r_{i,j}$. If $j > z$, then processor i alerts processor $z+1$ to manage processors in the interval $[z+1, j]$ to construct a right subtree of r_z. Finally, processor i manages processors in the interval $[i, z-1]$, if any, to construct a left subtree of r_z.

(3) **(Progressive insertions)** This stage proceeds in rounds starting with round 0. Each round t has a *record insertion* step and a *node reconstruction* step. For $i \le x \le j$, define $\delta(x)$ to be the depth of record r_x in the insertion tree $T[i,j]$, where $\delta(\lceil (i+j)/2 \rceil) = 0$. Initially, processor p is *scheduled* for round $2\delta(p)$ and *manages* record r_p for *insertion* at level 0 of the B-tree.

(a) **(Record insertion)** For each processor p scheduled for round t, processor p inserts the record it is managing, r_p, into the B-tree at the position determined by r_p.parent and r_p.last, which were set during the construction of $T[i,j]$. First left children are inserted then right children. Each of these steps requires only local redirection of pointers and updating of flags, and is specified by procedure 1 in the appendix.

(b) **(Overflow correction)** For every node v that received at least one insertion in the record insertion step of round t, the set of processors that inserted into node v cooperate to elect a leader, L_v, as follows. Each processor that inserted a left child into the node list, marks for election the record it inserted. Next each processor that inserted a record walks up the node list toward the head of the node until either it encounters a marked record, other than the one it inserted, or the head of the node. The processor reaching the head of the node is the leader. The other participants in the leader election are finished stage 3. For each node v that received at least one insertion, processor L_v determines the new size s_v of v, removing the election marks in the process. If $s_v \le b$ then L_v is finished stage 3. Otherwise, v has grown too large so L_v splits v into three pieces: the middle record, say r', those records that precede r', and those records that follow r'. These pieces are restructured into a depth 2 B-tree with root r'. Again this involves only local redirection of pointers and setting of flags, and is specified by procedure 2 in the appendix. Let $l(v)$ be the level of v. If $l(v) = L$ then v was the root of the B-tree so L_v updates the global root information and is then finished stage 3, otherwise processor L_v is scheduled for round $t+1$ and manages r' for insertion at level $l(v)+1$.

(4) **(Termination)** Apply the same termination procedure as used in stage 3 of the search procedure.

4.3. Correctness. We first show that for every round t, all records scheduled for insertion in round t have distinct insertion points. In round 0, each root of an insertion tree is scheduled for insertion and has a distinct insertion point among the leaves of the B-tree. Given the insertion of a depth d record r of an insertion tree during round t, the depth $d+1$ children of r, which are scheduled for insertion at round $t+2$, have distinct insertion points because they lie on either side of r. Insertions into internal nodes are also at distinct locations, because the overflow correction step promotes at most one record of a node for insertion into its parent.

Right insertions always follow Left insertions so no simultaneous access of the same record can occur within one level. The algorithm separates successive insertions into the leaves of the B-tree by two rounds. Thus if there are insertions at level i in round t then there are none at level $i-1$ or level $i+1$ in round t. Thus insertions at adjacent levels do not occur in the same round. So simultaneous modification of the same pointer is avoided and the algorithm has no read or write conflicts.

Finally we show that the B-tree structure is maintained. Consider any collection of insertions into a correct B-tree node v, at level l with $z \le b-1$ records. There are at most $z+1$ insertion points for a total of at most $2b-1$ records. Hence, after overflow correction there is either one node v at level l with at most $b-1$ records, or there are two new nodes of size

at most $b-1$ at level l, and one parent record, promoted for insertion into level $l+1$. This property suffices to show (by a simple inductive argument) that when the insertion algorithm terminates, the B-tree data structure is reestablished for the whole tree.

4.4. Complexity. A complexity of $O(\log n + b \log_b m)$ has been established in section 3 for stage 1. Clearly, stage 2 completes in $O(\log n)$ steps. The maximum depth of any insertion tree is $\log n$. Thus in stage 3 each record of each insertion tree is scheduled for insertion at level 0 by round $2 \log n$. Any insertion at level l at round t that results in a node splitting, creates a new insertion at level $l+1$ scheduled for round $t+1$. Hence after at most $2 \log n + \log_b(m+n)$ rounds the last insertion is complete. Since each round completes in $O(b)$ time, the insertion algorithm takes $O(b(\log n + \log_b m))$ time.

The results of this section are summarized by the following theorem.

Theorem 4.1. *The n-Key Insertion Problem for an order b B-tree containing m nodes can be solved on an n-processor EREW PRAM in* $O(b(\log n + \log_b m))$ *time.*

5. DELETION

5.1. Informal Overview of the Algorithm. The algorithm has four stages. Stages 1 and 2 find and mark each record that is to be deleted by applying the search algorithm. For each marked internal record, the succeeding record (which is necessarily the head record of a leaf node) is also found. The deletions and required rebalancing are performed in stage 3 which proceeds in rounds. In each round, each marked internal record swaps its key and data pointer with those of the first unmarked record in the succeeding leaf node if one exists. Otherwise, to avoid simultaneous access or records, the marked internal record waits three rounds before trying again to swap. All marked records in leaf nodes are removed. If any node becomes too small after the deletions, then that node, its parent and its siblings restructure themselves so that the children are large enough. If the parent is now too small, it schedules itself as a child needing restructuring in the next round. In the case that local restructuring results in an empty parent, *dummy nodes* containing only a parent and a child pointer and a flag to indicate that it is a dummy node serve to propagate the necessary restructuring upwards. Stage 4 ensures a clean termination.

5.2. Formal Description.

(1) **(Search)** Apply the search algorithm of section 3. If deletion key k_p is not in the B-tree then processor p terminates. Each remaining processor p *manages* the record r_p specified by POSITION$[p]$.

(2) **(Partial scheduling)** Each processor p, marks r_p for deletion. If r_p is in an internal node then p finds the first record that follows r_p in sorted order and retains a pointer π_p to it. The details are specified in procedure 3 in the appendix. Before proceeding to stage 3, processors are synchronized as in stage 3 of the search algorithm.

(3) **(Progressive deletions)** Stage 3 proceeds in rounds beginning with round 0. Initially all processors are *scheduled* for round 0. Each round t has a *record deletion* step and a *node reconstruction* step.

(a) **(Record deletion)** Each processor p managing a record in an internal node, examines the node v_p whose head is pointed to by π_p. If any record in v_p is not marked for deletion, then p swaps the key and data in record r_p with the key and data in the first unmarked record, say s_p in v_p, becomes the manager of record s_p, unmarks r_p for deletion and marks s_p for deletion. Otherwise all records in v_p are already marked for deletion so p sets π_p to π_p.pred and schedules itself to continue at the start of round $t+4$.

Next all processors managing a record in a common leaf node v cooperate to elect a leader, L_v as in stage 3 step 2 of the insertion algorithm. Each processor that does not become a leader is finished stage 3. Each leader L_v deletes from v the records marked for deletion, removing the election marks in the process. If all records in v are deleted, then L_v replaces v with a dummy node constructed out of v's first record.

(b) **(Node reconstruction)** All leaders of nodes with a common parent, say μ cooperate to elect a chief C_μ in three steps as follows. Let L_v be any such leader of a node v and let r_v be the record in node μ containing the pointer to node v. First, if v is the last child of μ then L_v marks r_v for election. Second, if v is not the last child of μ and r_v is unmarked for election then L_v marks r_v for election. Finally, all leaders that marked a record for election elect a chief as in the leader election in stage 3 step 2 of the insertion algorithm. Each leader that does not become a chief is finished stage 3. Each chief C_μ collects all records in node μ and its immediate children (except dummy nodes), removing election marks in μ in the process. These records are formed into a B-tree B_μ of height at most 2. If B_μ has height 1 and μ is not the root of the B-tree, then a dummy root node with no keys is used to pad to height 2. All grandchildren of μ are reattached to B_μ in order. Let μ' denote the root of B_μ. If μ was not the root of the B-tree, then chief C_μ attaches μ' in place of μ at μ's parent. Otherwise, C_μ updates the global root information to μ'.

Finally, if μ' has fewer than $b/2-1$ keys, and μ was not the root, then C_μ is the leader of node μ' in round $t+1$. Otherwise, C_μ is finished stage 3.

(4) **(Termination)** Synchronize the processors by applying the termination procedure of stage 3 of the search procedure. Each processor marked exactly one record for deletion that was successfully deleted. Before terminat-

ing the algorithm, the processors cooperate to return these records to the memory pool. As noted in section 2, this can be accomplished in $O(\log n)$ time.

5.3. Correctness. Correctness of stage 1 (Search) and stage 4 (Termination) are already established in section 3 so we consider only stages 2 and 3. Let B_t denote the structure existing at the end of round t of stage 3. First consider the operation of the algorithm when all records to be deleted reside in the leaves of the B-tree.

Lemma 5.1. *If all records to be deleted reside in the leaves of the B-tree, then at the end of round t of stage 3, all leaders reside at level $t+1$ and a node has a leader if and only if it has fewer than $b/2-1$ records.*

Proof. Initially all leaders are at level 0. During round t, either a leader at level t becomes a chief at level $t+1$ or is finished stage 3 and is therefore permanently out of contention. Next a chief at level $t+1$ becomes a leader for round $t+1$ if it has fewer than $b/2-1$ records or it is finished stage 3 and again is permanently out of contention. ■

Lemma 5.2. *If all records to be deleted reside in the leaves of the B-tree, then at the end of round t of stage 3 for any $t \leq L-2$, each subtree at level t of B_t is a correct B-tree of height $t+1$ whose root contains between $b/2-1$ and $b-1$ records.*

Proof. The proof is by induction on the round number. Since the basis ($t=0$) is clear from the algorithm, assume that at the end of round $t \leq L-3$, each subtree at level t is a B-tree of height $t+1$ whose root contains between $b/2-1$ and $b-1$ records. By lemma 5.1 each node at level $t+1$ either has fewer than $b/2-1$ records and a resident leader or has between $b/2-1$ and $b-1$ records and no resident leader. Consider a node v at level $t+2$. If v has no child with a resident leader then there is no restructuring within the subtree rooted at v during round $t+1$. Hence at the end of round $t+1$, each child of v is a B-tree of height $t+2$ whose root contains between $b/2-1$ and $b-1$ records.

Otherwise v has at least one child with a resident leader. Thus a chief will be elected for v during round $t+1$. In this case we need to show that the chief can reconstruct the subtree B_v rooted in the place of v in such a way that each child of v', the new root, has size between $b/2-1$ and $b-1$, and each grandchild of v has a unique attachment point and sorted order of all records in the subtree rooted at v is maintained. Since no restructuring has progressed beyond level $t+1$, node v contains at least $b/2-1$ records. Thus there are enough records to construct at least one non-root B-tree node. In the sorted order of the records, each record in B_v falls between the records of two adjacent grandchildren of v. Thus the sorted order of the structure B_{t+1} is maintained and the number of attachments points in B_v is exactly equal to the number of grandchildren of v. ■

Correctness in the case when all deletions are at the leaves follows immediately from lemma 5.2. Now consider the general case where records in internal nodes are being deleted.

Lemma 5.3. *At the beginning of every round t, if r_p is marked for deletion and r_p is in an internal B-tree node then π_p points to the first record s_p of some node v_p, where (a) s_p is the successor of r_p in the sorted order of records and (b) v_p is a leaf node.*

Proof. First observe that since sorted order is given by the inorder traversal, each internal record is followed in the sorted order by an external node. Thus it suffices to establish (a), which is done by induction on the round number. The basis ($t=0$) is clear from the algorithm, since π_p is set to the successor of r_p in the sorted order during Partial scheduling. Assume that r_p is marked for deletion and resides in an internal node γ_p at the beginning of round t and that π_p points to the first record s_p of some node v_p, where s_p is the successor of r_p in the sorted order. If the contents of r_p are swapped with the contents of a record in v_p during round t, then the deletion is completed, r_p is unmarked for deletion, and the lemma holds.

Otherwise all records in v_p are marked for deletion. There are two case to consider, depending on the level of r_p. If r_p resides at some level greater than level 1, then after round t, r_p still resides in an internal node. During round t pointer π_p is set to the first record q_p of the parent of v_p. But since all records in v_p are deleted in round t, record q_p becomes the immediate successor of r_p and is guaranteed to be in a leaf node after the node reconstruction of round t so again the lemma holds. If r_p resides at level 1 and r_p is the last record in a node, then after node reconstruction, r_p is guaranteed to be in a leaf node for round $t+1$, so the lemma holds. If r_p is not the last record in a node, then π_p is reset to r_p.succ, which is the successor of r_p in the sorted order since all records in the node between these two were deleted during round t. Furthermore, after node reconstruction, either r_p or r_p.succ will be in a leaf node for round $t+1$, so the lemma holds. ■

It follows from lemma 5.3 that each record marked for deletion is eventually deleted. The algorithm can be viewed as waves of deletions pipelined through the B-tree. There are three levels between successive waves since when an internal record fails to swap with a leaf record during round t, it reschedules itself to try again at the start of round $t+4$. Since in round t, a chief at a node at level i updates records only at levels $i-2$ through $i+1$ inclusive the separation of three levels ensures that the pipelined waves do not interfere with each other. Thus the output of the algorithm is the same as running each successive wave to completion, and hence by lemma 5.2, the B-tree structure in maintained.

It remains to show that the algorithm is free of read and write conflicts. Stage 1 is conflict free by the proof in section 3. Stage 2 is conflict free because for each internal record r_p, π_p points to a unique record. The first two steps of the chief election prevent simultaneous marking by the two children of

the last record in the node μ (all other records have one child). The leader election and the remainder of the chief election are clearly conflict free. The separation of successive waves of deletion by three levels guarantees the remainder of stage 3 is conflict free.

5.4. Complexity. Stage 1 requires $O(\log n + b \log_b m)$ time. Stage 2 requires $O(\log_b m)$ time to find candidates for swapping, and $O(\log n)$ time to synchronize processors. Stage 4 requires $O(\log n)$ time. Processors that first become leaders in round t of stage 3 finish the stage no later than round $t + \log_b m$. New leaders are introduced only when a swap of an internal record and a leaf occurs.

Claim 5.4. *All internal records marked for deletion have been swapped with records in leaf nodes by round* $4\lceil \log_{b/2-1} n \rceil$.

Proof. Any internal record unable to swap must be blocked by at least $b/2 - 1$ records that will be deleted in the current round. Therefore, all swap are achieved within $\lceil \log_{b/2-1} n \rceil$ attempts and a new attempt is made every fourth round. ■

It follows that stage 3 completes in $O(\log_b m + \log_b n)$ rounds and it is easily established that each round completes in $O(b^2)$ time.

Theorem 5.5. *The n-Key Deletion Problem for an order b B-tree containing m nodes can be solved on an n-processor EREW PRAM in* $O(b^2(\log_b n + \log_b m))$ *time.*

6. Concluding Remarks

Our algorithms differ from that of Paul, *et. al* [13] in several respects. Our search algorithm searches a general B-tree; theirs is restricted to a 2-3$^+$ tree. While their algorithm could be extended to a B-tree of order b, the description of the algorithm would become more complex as b increases. We achieve a simpler description that is simultaneously general for B-trees of any order, by exploiting the binary tree interpretation of our B-tree representation. Another difference is that in each round of stage 1 a packet either splits or moves to a new location. In their algorithm packets can both split and move in the same round. These simplifications facilitate a precise description and proof of the scheduling required to avoid read conflicts. Although Paul, *et. al* prove that in each round there are most four packets at a node of the 2-3$^+$ tree, they do not show explicitly how to schedule operations on these four packets without read conflicts. The presence of data in internal nodes complicates insertion and deletion over that of Paul, *et. al* so our algorithms for these operations are more elaborate than theirs. Our insertion algorithm allocates only one record to each processor. Thus the work space is only $O(n)$ (rather than $\Theta(n \log n)$).

In our algorithms the nodes of the B-tree are structured as lists. Further efficiency could be achieved for large values of b by structuring the nodes themselves as a 2-3 tree (a B-tree with $b = 2$), however, using a 2-3 tree initially, instead of a large value for b, provides better constants. This points out a potentially interesting open problem. In the sequential setting large values of b are used to offset the expense of obtaining data from secondary storage at an increase in the expense of searching in fast primary storage. Can similar reasoning be applied in the parallel case? What is meant by secondary storage on a PRAM? The constant in the insertion algorithm can be improved somewhat at the expense of more involved proofs of correctness by structuring the insertion tree on the fly, causing the computation of stage 2 to overlap that of stage 3 after one step. As well, in stage 3 of the deletion algorithm, successive waves of deletion can be scheduled with a separation of only two levels, as opposed to three, but more care must be taken to ensure that pointers are not simultaneously modified.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data structures and algorithms*, Addison–Wesley Publishing Company, 1983.
2. R. Bayer and K. Unterauer, *Prefix B-trees*, ACM Trans. Database Systems **1** (1977), no. 2, 11–26.
3. Richard Cole, *Parallel merge sort*, SIAM J. Comput. **17** (1988), no. 4, 770–785.
4. Douglas Comer, *The ubiquitous B-tree*, Comput. Surveys **11** (1979), no. 2, 121–137.
5. G. Held and M. Stonebraker, *B-trees reexamined*, Comm. ACM **21** (1978), no. 2, 139–143.
6. Lisa Higham and Eric Schenk, *Memory initialization and allocation on an EREW PRAM*, In preparation.
7. Richard M. Karp and Vijaya Ramachandran, *A survey of parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. A, Elsevier Science Publishers, Amsterdam, The Netherlands, and The MIT Press, Cambridge, Massachusetts, U.S.A., 1990.
8. Donald E. Knuth, *Sorting and searching*, The Art of Computer Programming, vol. 3, Addison–Wesley Publishing Company, 1973.
9. H. T. Kung and Philip L. Lehman, *Concurrent manipulation of binary search trees*, ACM Trans. Database Systems **5** (1980), no. 3, 354–382.
10. R. E. Ladner and M. J. Fischer, *Parallel prefix computation*, J. Assoc. Comput. Mach. **27** (1980), 831–838.
11. Vladimir Lanin and Dennis Shasha, *A symmetric concurrent B-tree algorithm*, Proceedings of the Fall Joint Computer Conference (Washington, DC, USA), IEEE Comput. Soc. Press, November 1986, pp. 380–389.
12. P. L. Lehman and S. Bing Yao, *Efficient locking for concurrent operations on B-trees*, ACM Trans. Database Systems **6** (1981), no. 4, 650–670.
13. W. Paul, U. Vishkin, and H. Wagener, *Parallel dictionaries on 2-3 trees*, Lecture Notes in Computer Science 143: Proceedings of the 10th Colloquium on Automata, Languages and Programming, Springer Verlag, 1983, pp. 597–609.
14. Yehoshua Sagiv, *Concurrent operations on B*-trees with overtaking*, J. Comput. System Sci. **33** (1986), 275–296.
15. B. Samadi, *B-trees in a system with multiple views*, Inform. Process. Lett. **5** (1976), no. 4, 107–112.
16. Dennis Shasha and Nathan Goodman, *Concurrent search structure algorithms*, ACM Trans. Database Systems **13** (1988), no. 1, 53–90.
17. Etsuya Shibayama, *A fast parallel merging algorithm for 2-3 trees*, Proceedings of the RIMS Symposia on Software Science and Engineering II, Lecture Notes in Computer Science, vol. 220, Springer Verlag, Berlin, New York, 1983/1984.
18. Biing-Feng Wang and Gen-Huey Chen, *Cost-optimal parallel algorithms for constructing 2-3 trees*, J. Parallel and Distributed Comput. **11** (1991), 257–261.

19. Biing-Feng Wang, Gen-Huey Chen, and M. S. Yu, *Cost-optimal parallel algorithms for constructing B-trees*, Proceedings of the International Conference on Parallel Processing (Washington, DC, USA), IEEE Comput. Soc. Press, August 1991, pp. 294–295.
20. William E. Weihl and Paul Wang, *Multi-version memory: Software cache management for concurrent B-trees*, Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing (Washington, DC, USA), IEEE Comput. Soc. Press, December 1990, pp. 650–655.

7. APPENDIX

Procedure 1: Inserting records into a node.

if r_p is a left child (r_p.last is false) then
  Set r_p.head to r_p.pred.head.
  Set r_p.tail to false.
  Set r_p.pred.head to false.
  Set r_p.last to r_p.pred.last.
  Set r_p.pred.last to false.
  Set r_p.succ.pred to r_p.pred.
  Set r_p.pred.child to r_p.succ.
  Set r_p.succ to r_p.pred.
  Set r_p.pred to r_p.pred.pred.
  Set r_p.succ.pred to point to r_p.
  if r_p.last is false and r_p.head is true then
    Set r_p.pred.child to point to r_p.
  else
    Set r_p.pred.succ to point to r_p.
  end if
end if
if r_p is a right child (r_p.last is true) then
  Set r_p.pred.tail to false.
  Set r_p.tail to true.
  Set r_p.last to false.
  Set r_p.head to false.
end if

Procedure 2: Splitting a B-tree node.

Let r be the first record in the node and let r' be the middle record in the node.
Set r'.succ.head to true.
Set r'.succ.last to true.
Set r'.last to r.last.
Set r.last to false.
Set r'.pred.tail to true.
Set r'.head to true.
Set r'.tail to true.
Set r'.pred.succ to r'.child.
Set r'.child.pred to r'.pred.
Set r'.pred to r.pred.
Set r.pred to point to r'.
Set r'.child to point to r.

Procedure 3: Finding the next record in sorted order given a record r_p in an internal node.

if r_p.last is true then
  Set π_p to r_p.succ.
else
  Set π_p to r_p.succ.child.
end if
while π_p.child is not **null** do
  Set π_p to π_p.child.
end while