Nordic Journal of Computing 1(1994), 402–432.

PARALLEL DYNAMIC LOWEST COMMON ANCESTORS*

ERIC SCHENK†
Department of Computer Science, University of Toronto
Toronto, Ontario, M5S–1A4, Canada
schenk@cs.toronto.edu

Abstract. This paper gives a CREW PRAM algorithm for the problem of finding lowest common ancestors in a forest under the insertion of leaves and roots and the deletion of leaves. For a forest with a maximum of n vertices, the algorithm takes $O(m/p + r \log p + \min(m, r \log n))$ time and $O(n)$ space using p processors to process a sequence of m operations that are presented over r rounds. Furthermore, lowest common ancestor queries can be done in worst case constant time using a single processor. For one processor, the algorithm matches the bounds achieved by the best sequential algorithm known.

CR Classification: E.1

Key words: data structures

1. Introduction

Finding lowest common ancestors in trees is a frequently occurring problem in the literature and has found application in such diverse problems as computing dominators in reducible flow graphs [Aho *et al.* 1976], detecting negative cycles in sparse graphs [Maier 1979], planarity testing [Já Já and Simon 1982], and computing weighted matchings [Gabow 1990]. This paper gives a Concurrent Read Exclusive Write Parallel Random Access Machine (CREW PRAM) algorithm for the problem of finding lowest common ancestors in a forest of trees under the insertion of leaves and roots, and the deletion of leaves. As part of the solution to dynamic lowest common ancestors, this paper also defines and solves the dynamic restricted range minimum problem: a restricted version of the problem of finding the minimal valued element between two elements in a collection of lists under the insertion and deletion of elements.

Both problems considered in this paper are treated as Abstract Data Types with a fixed set of operation types. Operations are classified as either *cooperative* or *independent*. Cooperative operations must be processed together as a group; independent operations can be performed independently by a

* A preliminary version of this paper appeared in the Proceedings of the 4th Scandinavian Workshop on Algorithm Theory [Schenk 1994].

† This research was supported in part by an NSERC postgraduate scholarship.

Received May 1994. Accepted September 1994.

single processor provided the data structure is not being updated. An external agent is assumed to make rounds of one or more requests for cooperative operations, where each round consists of requests of the same type and must be dealt with on-line before the next round. Between rounds, the external agent can freely make independent requests, which must be processed on line before the next round can proceed.

The *dynamic lowest common ancestor problem* is to maintain a forest of trees of unbounded degree under the following operations.

initialize(T, x)**:** Add a new tree T to the forest, where T contains the single vertex x.

lca(T, x, y)**:** For any x and y in T, find the vertex of greatest depth in T that is an ancestor of both x and y.

add-leaf(T, x, y)**:** Add the new vertex y as a child of the vertex x in T.

add-root(T, x)**:** Make x the root of the tree T and the old root a child of x.

delete-leaf(T, x)**:** Remove the leaf x from the tree T.

We place the further restriction that, within a round, at most one *add-leaf* operation can be specified at each existing vertex, and at most one *add-root* operation can be specified for each tree in the forest. The *lca* operation is *independent*, all other operations are *cooperative*.

Let n denote the maximum number of vertices ever contained in the forest. When a total of m cooperative operations are requested over r rounds, we achieve $\mathrm{O}(m/p + r \log p + \min(m, r \log n))$ time and $\mathrm{O}(n)$ space on a p processor CREW PRAM. For $p \in \mathrm{O}(n/r \log n)$ the time-processor product is $\mathrm{O}(m)$, which is optimal. Also, any *lca* query can be performed in worst case constant time using one processor without performing any write operations. This is the first parallel algorithm for the dynamic problem. This algorithm can be used to substantially improve the PRAM simulation of the Parallel Asynchronous Recursion Model (PAR Model) given by Higham and Schenk [1992] and Schenk [1992], reducing the worst case work in the simulation from $\mathrm{O}(w^2 \log p)$ to $\mathrm{O}(w \log p)$, where w is the work performed by the PAR Model algorithm, and p is the total number of processes used by the PAR Model algorithm.

Gabow [1990] gives a sequential algorithm to process sequences of m *lca*, *add-leaf* and *add-root* operations creating a tree of maximum size n in $\mathrm{O}(m)$ time and $\mathrm{O}(n)$ space. This matches the sequential complexity of our algorithm. As well, the method used here to deal with deletions applies directly to Gabow's construction.

Both Gabow's algorithm and our algorithm are based upon algorithms for the *static lowest common ancestor problem*, which is to pre-process a fixed tree T of size n such that on-line *lca* queries can be answered efficiently. Harel and Tarjan [1984] give an algorithm that achieves $\mathrm{O}(n)$ time and space for preprocessing and constant time to answer queries. Gabow [1990] bases his algorithm for the dynamic lowest common ancestor problem on this algorithm. Under the assumption that the tree is given together

with an Euler tour, Berkman and Vishkin [1989] use a completely different approach to derive a constant time, n processor, preprocessing algorithm for the static problem on a special variant of the CRCW PRAM. (If the Euler tour is not available, then the preprocessing requires $O(\log n)$ time on $n/\log n$ processors.) This results in a data structure that allows *lca* queries to be answered in constant time using a single processor. Our solution to the dynamic lowest common ancestor problem is based upon Berkman and Vishkin's algorithm.

Our construction for the dynamic lowest common ancestor problem is a reduction to the *dynamic restricted range minimum problem*, which is a variation of a problem defined by Berkman and Vishkin [1989] to help compute lowest common ancestors. This problem is to maintain a collection of lists of elements, where each element x has an associated value $value(x)$. A total ordering is defined on the elements of a list such that for two elements x and y, $x < y$ if (1) $value(x) < value(y)$ or (2) $value(x) = value(y)$ and y precedes x in the list. The following operations must be supported.

initialize-min$(X, x_1, x_2, \ldots, x_{n'})$**:** Add a new list X to the collection, where X contains $[x_1, x_2, \ldots, x_{n'}]$ in that order.

list-insert$(X, x, y, side)$**:** Insert a new element x into the list X either to the left or right of y as indicated by the value of *side*. It must be the case that $x > y$, and that for at least one neighbor v of x, there is a constant $k > 0$ such that $|value(x) - value(v)| \leq k$.

list-delete(X, x)**:** Remove the element x from the list X. The element x must be a local maximum in the list. That is, if y is a neighbor of x in the list, then $x > y$.

collect(X, C)**:** Put the elements of the list X into the array C in the order they occur in X.

rmin(X, x, y)**:** For any x and y in X, find the rightmost minimum valued element in X between x and y inclusive.

prec(X, x, y)**:** For any x and y in X, determine if x precedes y in the list X.

Within a round, at most one insertion can be specified at each element currently in a list X. For the purposes of analysis, the initialization of a list of n' elements is counted as n' operations; likewise a collect operation on a list of n' elements is counted as n' operations. The *rmin* and *prec* operations are *independent* operations, all other operations are *cooperative*.

Let n denote the maximum number of elements contained in the collection of lists at any one time. When a total of m cooperative operations are requested over r rounds, we achieve $O(m/p + r \log p + \min(m, r \log n))$ time and $O(n)$ space on a p processor CREW PRAM. The work is optimal for $p \in O(n/r \log n)$. Also, any *rmin* or *prec* query can be performed in worst case constant time using one processor without performing any write operations.

REMARK 1. The *list-delete* operation specified here is not used in the reduction from the dynamic lowest common ancestors problem. It is included

only to extend the possible operations on the dynamic restricted range minimum data structure as far as possible. The purpose of the *collect* operation is to allow access to the current contents of the data structure.

The remainder of this paper is organized as follows. Section 2 briefly defines the PRAM model and gives some basic results that are used in this paper. Section 3 gives the reduction from the dynamic lowest common ancestor problem to the dynamic restricted range minimum problem. Section 4 gives a simple but not very efficient algorithm for dynamic restricted range minimum. Section 5 shows how to make this algorithm optimal for very small lists. Section 6 combines the algorithms of section 4 and 5 to obtain a general optimal algorithm. Section 7 presents some concluding remarks.

2. Some PRAM Basics

We describe briefly the Parallel Random Access Machine (PRAM) model of computation, and some basic results that we make use of.

A PRAM is a collection of synchronized independent sequential processors with unique identifiers. These processors communicate through a global memory. Each memory cell stores $O(\log n)$ bits, where n is the size of the input. 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 (Exclusive Read Exclusive Write) PRAM no two processors may simultaneously access the same memory location for either reading or writing; in a CREW (Concurrent Read Exclusive Write) PRAM, simultaneous reading, but not writing, is permitted. (See Karp and Ramachandran 1990 for an overview of PRAM models and results.)

2.1 Self Simulation on Parallel Random Access Machines

In the algorithms presented in the remainder of this paper, the number of processors is a fixed value p. However, sometimes it is convenient to present an algorithm as though it uses some larger number of processors, say p'. In these cases we make use of the observation that any PRAM computation taking t time using p' processors, can be performed using $p \leq p'$ processors in $O(tp'/p)$ time by having each processor perform the tasks of at most $\lceil p'/p \rceil$ of the original p' processors [Karp and Ramachandran 1990].

2.2 Prefix Sums

The prefix sums computation is one of the basic tools in the design of efficient parallel algorithms [Karp and Ramachandran 1990]. Let $*$ be an associative operation over a domain D. Given an array $X = x_1, \ldots, x_s$ of s elements from D, the prefix sums problem is to compute the values $s_i = x_1 * x_2 * \ldots x_i$ for $i \in \{1, \ldots, s\}$. Ladner and Fischer [1980] give an optimal prefix sums algorithm for the EREW PRAM with a running time of $O(s/p + \log p)$ using p processors.

2.3 Array Compaction

One common application of prefix sums is to compact an array containing "dead" elements into a smaller array containing only "live" elements. To compute the position of "live" elements in this new array assign a 1 to each "live" element and a 0 to each "dead" element, and then compute the prefix sums with addition as the $*$ operator. Suppose, all "dead" elements are identifiable by some property that can be tested by a single processor in constant time. Then using Ladner and Fischer's algorithm these elements can be removed from an array of length s in $O(s/p + \log p)$ time using p processors.

2.4 List Ranking

The list ranking computation is similar to prefix sums computation. Given a linked list on s elements, compute the suffix sums of the last i elements of the linked list for all $i = 1, \ldots, s$. Cole and Vishkin [1988] give an optimal list ranking algorithm for the EREW PRAM with a running time of $O(s/p + \log p)$ using p processors.

2.5 Task Scheduling

In section 5 we will need to solve the following static scheduling problem. Consider a list of tasks $t_1, \ldots, t_e$ of known size. Let s_i denote the size of task i. Let $s = \sum_{i=1}^{e} s_i$, and let $M = \max_{i=1}^{e} s_i$. A task of size s_i can be completed in $O(s_i)$ time using one processor. Furthermore, a task cannot be broken down into sub-tasks that can be performed on more than one processor. We wish to assign sets of tasks to processors to minimize the time to complete all the tasks. Let $S_i = \sum_{j=1}^{i} s_j$ (the prefix sums of the sizes). We say a task t_i is a *group leader* if $\lfloor S_i/(s/p) \rfloor > \lfloor S_{i-1}/(s/p) \rfloor$. Also, task t_1 is a group leader by definition. The group leaders partition the list of tasks into sublists such that the total size of the tasks in each sublist is at most $O(s/p + M)$. If we assign one processor to each sublist of tasks the tasks can be completed in $O(s/p + M)$ time using p processors. Adding in the time to compute the prefix sums and the partitioning of the tasks, the tasks can be completed in $O(s/p + \log p + M)$ time using p processors.

2.6 Memory Allocation and Deallocation

The algorithms presented in this paper allocate and deallocate memory blocks of widely varying sizes. To avoid memory fragmentation we make use of a brute force relocating memory management scheme.

All memory is partitioned into *working* and *storage* memory via interleaving. Working memory is managed directly by the algorithms as needed for temporary data at various stages of the computation. All permanent data structures other than those described in this section are assumed to be in the *storage* memory.

The general strategy is as follows. When memory is deallocated it is marked as such, and a count of the number of deallocated cells is kept. After each round of computation, a garbage collector is run. If the number of deallocated cells exceeds half the total number of cells in use, then the garbage collector compacts the entire storage memory removing deallocated cells. This at most doubles the amortized cost of allocating and deallocating memory.

Given a list of allocation requests for a total of s memory cells, the allocations can be performed in $O(\min(s, s/p + \log p))$ time using p processors by a straight forward application of prefix sums. Similarly, given a list of deallocation requests for a total of s memory cells, the deallocations can be performed in $O(\min(s, s/p + \log p))$ time using p processors. The garbage collection process is also a straight forward application of prefix sums. If there are c memory cells allocated when a garbage collection is performed, then the garbage collection takes $O(c/p + \log p)$ time using p processors. It is easily seen that the total cost of garbage collections matches the total cost of marking cells as deallocated. Therefore, for the remainder of this paper we ignore the cost of performing garbage collection.

Finally, we observe that if at most n cells are allocated at any one time, then at most $O(n)$ cells of the storage memory are used. Hence it is acceptable to count the number of allocated cells to obtain a bound on the space complexity of our algorithms.

3. Dynamic Least Common Ancestors

To solve the dynamic least common ancestors problem we reduce it to the dynamic restricted range minimum problem. Our reduction is similar to that used by Berkman and Vishkin [1989] in their solution to the static lowest common ancestor problem. The following lemma is central to our reduction. Let the sequence $[x_1, \ldots, x_s]$ be the preorder tour of a tree T, where s is the number of vertices in T. (Recall that the preorder tour consists of the root of the tree, followed by the preorder tours of all subtrees.) Let $[x_i, \ldots, x_j]$ denote the sublist of elements between x_i and x_j inclusive.

LEMMA 1. *For any $i < j$, let z be the rightmost vertex of minimal depth in the sublist $[x_i, \ldots, x_j]$. If $x_i = z$, then $lca(x_i, x_j) = z$; otherwise $lca(x_i, x_j)$ is the parent of z in T.*

PROOF. We claim that z must be an ancestor of x_j. Assume not. Then let x be that ancestor of x_j with the same depth as z. By the definition of a preorder tour, x must come before x_j in the tour, and only descendants of x lie between x and x_j in the tour. It follows that x is to the right of z in X, and hence z is not the rightmost vertex of minimal depth in $[x_i, \ldots, x_j]$. It follows directly that if $x_i = z$, then $lca(x_i, x_j) = z$. Now assume $x_i \neq z$. Let y be the parent of z in T. Since z has minimum depth of any vertex in $[x_i, \ldots, x_j]$ and $depth(y) < depth(z)$, y is not in $[x_i, \ldots, x_j]$. Furthermore, y

must appear before z in preorder, thus y must appear before x_i. Finally, y must be an ancestor of x_i, since all descendants of y occur in a contiguous block in the preorder tour. Since z is not an ancestor of x_i it follows that $lca(x_i, x_j) = y$. □

REMARK 2. Note that only the relative depths of vertices are important. Depths can therefore be represented by negative numbers if necessary. We will make use of this property to allow the insertion of new roots.

With some minor additional structure this lemma reduces the problem of computing lowest common ancestors in a tree to the problems of determining the relative order of elements within a list and of finding the rightmost element with minimal depth among the elements in a sublist of a list.

3.1 The LCA Data Structure

Each tree T in the forest is represented independently. The field $T.list$ is a dynamic restricted range minimum list $[x_0, x_1, \ldots, x_s]$. The elements $x_1, \ldots, x_s$ are the vertices of T in preorder. The element x_0 is a sentinel with value $-\infty$ that is necessary to support the *add-root* operation. For $1 \leq i \leq s$, $value(x_i)$ is the depth of x_i in T. The field $T.head$ points to the first element in the list $T.list$, i.e. x_0. The field $T.root$ points to the root of the tree T, i.e. x_1. For each vertex v in the tree T, the field $v.parent$ stores the parent of the vertex v and the field $v.value$ stores $value(v)$ as defined for the dynamic restricted range minimum problem.

Efficient implementation of the *delete-leaf* operation cannot be supported directly with this structure. This difficulty is avoided by a lazy deletion scheme. When a *delete-leaf* operation occurs, the vertex in question is marked for future deletion from the range minimum list. When the total number of marked vertices becomes at least half as large as the entire data structure, then the data structure is thrown away and rebuilt from scratch. Since only leaves can be deleted, and no child will ever be added to a deleted leaf, we can see that the failure to immediately complete a deletion request will never cause a future request to be answered incorrectly.

Some additional structure is needed to support this deletion scheme. First, for each vertex v, the field $v.deleted$ stores a flag that is true if and only if the vertex has been marked for deletion. In addition, the array F stores a list of the trees in the forest. The number of trees in the forest is stored in *tree-count*, the size of the array F is stored in *tree-limit*. Initially F is an array of length 1, *tree-limit* is set to 1, and *tree-count* is set to 0. The number of vertices in the forest is stored in *forest-size*. The number of vertices in the forest that are marked for deletion is stored in *forest-deleted*. Initially both of these fields are set to 0.

Excluding the representation for the dynamic restricted range minimum lists, this structure uses $O(n)$ space. It will be shown that $O(n)$ space is sufficient for the dynamic restricted range minimum lists. Thus, the total space used is $O(n)$.

3.2 Computing LCA

Following lemma 1, an LCA operation $lca(T, x, y)$, is implemented as follows.

1. if $prec(T.list, y, x)$ is true then swap x and y;
2. $z \leftarrow rmin(T.list, x, y)$;
3. if $z = x$ then return z
   else return $z.parent$;

It will be shown in section 6 that the *prec* and *rmin* operations can be performed in constant time using a single processor without performing any write operations. It follows that the entire computation can be completed in constant time using a single processor without performing any write operations.

3.3 Initializations

A round of q initialize operations $R_1, \ldots, R_q$, where $R_i = initialize(T_i, x_i)$, is implemented as follows.

1. for $1 \leq i \leq q$ in parallel, create a vertex v_i, and set $v_i.value = -\infty$;
2. for $1 \leq i \leq q$ in parallel
   $x_i.value \leftarrow 0$;
   $x_i.parent \leftarrow$ `null`;
   $x_i.deleted \leftarrow$ false;
   $T_i.root \leftarrow x_i$;
   $T_i.head \leftarrow v_i$;
3. for $1 \leq i \leq q$ in parallel, $initialize\text{-}min(T_i.list, v_i, x_i)$;
4. increment both *forest-size* and *tree-count* by q;
5. if *tree-count* > *tree-limit* then
   tree-limit $\leftarrow 2$*tree-count*;
   allocate a new array F of size *tree-limit*;
   copy the old array contents into the new array;
6. for $1 \leq i \leq q$ in parallel, $F_{tree\text{-}count+1-i} \leftarrow T_i$;

The necessary memory for vertex and tree structures can be allocated in $O(q/p + \log p)$ time using p processors using the algorithm of section 2.6. Steps 1, 2 and 6 can be performed in $O(q/p)$ time using p processors. Since each initialize operation in step 3 initializes a list of two elements, step 3 contributes $2q$ operations toward the cost of maintaining the dynamic restricted range minimum lists. Step 4 can be performed in constant time using one processor. If the current size of F is c, then step 5 can be performed in $O(c/p + \log p)$ time using p processors.

3.4 Adding Roots

A round of q add root operations $R_1, \ldots, R_q$, where $R_i = add\text{-}root(T_i, x_i)$, is implemented as follows.

1. for $1 \leq i \leq q$ in parallel

$x_i.value \leftarrow T_i.root.value - 1$;
$x_i.parent \leftarrow$ `null`;
$x_i.deleted \leftarrow$ false;
$T_i.root.parent \leftarrow x_i$;

2. increment *forest-size* by q;
3. for $1 \leq i \leq q$ in parallel, *list-insert*$(T_i.list, x_i, T_i.head,$ right);

Note that T_i is unique for each i, since at most one root can be added to a tree in a round. Also note that new roots will have negative values assigned to them. Since the depth values are only used to determine which vertex has minimal depth, this will not cause any difficulties. The necessary memory for vertex and tree structures can be allocated in $O(q/p + \log p)$ time using p processors using the algorithm of section 2.6. Step 1 can be performed in $O(q/p)$ time using p processors. Step 2 can be performed in constant time using one processor. Step 3 contributes q insertion requests toward the cost of maintaining the dynamic restricted range minimum lists.

3.5 Adding Leaves

A round of q add leaf operations $R_1, \ldots, R_q$, where $R_i =$ *add-leaf*(T_i, x_i, y_i), is implemented as follows.

1. for $1 \leq i \leq q$ in parallel

$y_i.value \leftarrow x_i.value + 1$;
$y_i.parent \leftarrow x_i$;
$y_i.deleted \leftarrow$ false;

2. increment *forest-size* by q;
3. for $1 \leq i \leq q$ in parallel, *list-insert*$(T_i.list, x_i, y_i,$ right);

The necessary memory for vertex and tree structures can be allocated in $O(q/p + \log p)$ time using p processors using the algorithm of section 2.6. Step 1 can be performed in $O(q/p)$ time using p processors. Step 2 can be performed in constant time using one processor. Step 3 contributes q insertion requests toward the cost of maintaining the dynamic restricted range minimum lists.

3.6 Deleting Leaves

The deletion of leaves is implemented by marking deleted vertices as such, and rebuilding the entire data structure whenever the number of marked vertices becomes greater than half the total number of vertices. Since only leaves can be deleted, any vertex that has been marked as deleted will never be the answer to any lowest common ancestor query. Thus the failure of the algorithm to remove these vertices from the structure will not cause a future error.

A round of q delete operations $R_1, \ldots, R_q$, where $R_i =$ *delete-leaf*(T, x), is implemented as follows.

1. for $1 \leq i \leq q$ in parallel, $v_i.deleted \leftarrow$ true;
2. increment *forest-deleted* by q;

3. if 2*forest-deleted* $>$ *forest-size* then
   perform collect operations on all the trees in F placing the contents of the trees into arrays;
   remove deleted items from the arrays;
   rebuild the data structure using the initialize operations;
   deallocate memory that is no longer in use;

Step 1 can be performed in $O(q/p)$ time using p processors. Step 2 can be performed in constant time using one processor. If c' is the total number of vertices in the forest that had to be collected (including vertices marked for deletion), then step 3 contributes collect operations on a total of c' elements toward the total cost, plus initialization operations on at most c' elements. In addition, the compaction of the collected elements takes $O(c'/p + \log p)$ time using p processors, and the deallocation of memory takes a further $O(c'/p + \log p)$ time using p processors.

3.7 Complexity

The solution to the dynamic restricted range minimum problem presented in the remainder of this paper, together with the above discussion implies the following.

THEOREM 1. *For a forest containing at most n vertices at any one time, any sequence of m cooperative dynamic lowest common ancestor operations, presented over r rounds, can be processed on a CREW PRAM in $O(m/p + r \log p + \min(m, r \log n))$ time and $O(n)$ space using p processors. In addition, lca queries can be processed in constant time using a single processor without performing any writes.*

PROOF. The stated bound on space has already been established, as has the stated bound on the time taken by *lca* queries. It remains to establish the time bound on cooperative operations.

Let m_i denote the number of cooperative operations requested in round i. Note that $\sum_{i=1}^{r} m_i = m$. Let c_i denote the number of elements of F that must be copied due to a reallocation of F in round i. If F is not reallocated in round i then $c_i = 0$. Since F always at least doubles in size, and the final size of F is at most $2n$, it follows that $\sum_{i=1}^{r} c_i \in O(n)$. Let c'_i denote the number of vertices in the forest that must be collected and then initialized in round i because the number of deleted elements exceeded half the size of the data structure. If no collections were performed in round i, then $c'_i = 0$. Since a round of collections only occurs when the number of deleted elements is more than half the number of elements in the data structure, it follows that $\sum_{i=1}^{r} c'_i \in O(m)$.

From these facts, and the running times derived in the subsections above, we can see that a total of $O(m)$ operations are induced over $O(r)$ rounds in the dynamic restricted range minimum data structure. Furthermore, summing the times for these operations we obtain a total of $O(m/p + r \log p + \min(m, r \log n))$ time on p processors. □

4. A Simple Dynamic Restricted Range Minimum Algorithm

This section presents a simple, but not very efficient, algorithm for the dynamic restricted range minimum problem. The data structure presented in this section will be referred to as the *basic structure*. This algorithm relies on a solution to a restricted version of the dynamic lowest common ancestor problem, which we solve by taking advantage of some structural properties of a binary tree.

4.1 Preliminaries

For any list X, let $|X|$ denote the number of elements in the list. For any vertex v in a binary tree, let B_v denote the subtree rooted at v, let $|B_v|$ denote the number of leaves in B_v, let $\ell(v)$ denote the leftmost leaf and let $r(v)$ denote the rightmost leaf of B_v. The depth of a vertex v is denoted $\mathit{depth}(v)$, where the root of a tree has depth 0. Let $\mathit{lca}(x, y)$ denote the lowest common ancestor of any two vertices x and y.

By analogy with half open intervals on a number line we define half open intervals on a list. The notation $[x, \ldots, y)$ denotes the list of elements between x and y excluding y. Similarly $(x, \ldots, y]$ denotes the list of elements between x and y excluding x. In both cases, if $x = y$ the list is empty.

The operators $\vee$, $\wedge$, $\oplus$ and $\neg$ denote the bitwise boolean or, bitwise boolean and, bitwise boolean exclusive or, and bitwise complement operations, respectively.

Finally, throughout this section, we assume that n and $\hat{n}$ are known quantities, where n the maximum number of elements that will be in the collection at any one time and $\hat{n}$ is the maximum number of elements that will be in any one list in the collection at any one time. The final algorithm given in section 6 will remove this assumption.

4.2 The Basic Data Structure

The basic structure for representing a list $[x_1, \ldots, x_s]$ is a binary tree of depth $\mathrm{O}(\log s)$ whose leaves in left to right order are $x_1, \ldots, x_s$. For each list X there is a single field $X.\mathit{root}$ that points to the root of the binary tree representing the list. In addition:

(1) Each vertex v has a field $v.\mathit{min}$ that points to to the minimum leaf in B_v, a field $v.\mathit{depth}$ that stores its depth, and fields $v.\mathit{parent}$, $v.\mathit{left}$ and $v.\mathit{right}$ that respectively point to its parent, left child and right child, if any.

(2) Each leaf v holds an array $A_v[1, \ldots, \mathit{depth}(v)]$. Let v_i denote the ancestor of v at depth i. For all $1 \leq i \leq \mathit{depth}(v)$, if v_i is a right child, then $A_v[i]$ points to the minimum element in the list $[\ell(v_i), \ldots, v)$, otherwise $A_v[i]$ points to the minimum element in the list $(v, \ldots, r(v_i)]$. In either case, if the list in question is empty, then $A_v[i] = 0$.

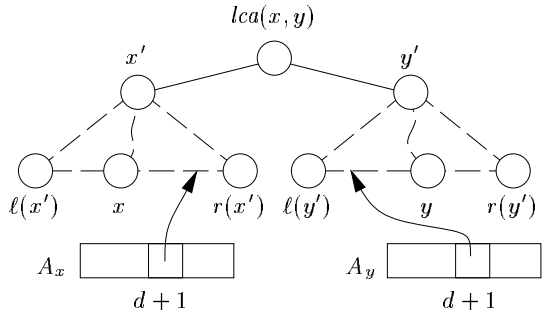

Fig. 1: Computing *rmin*.

(3) Each vertex v is labelled by a pair of integers $\langle \pi_v, \mu_v \rangle$. The root of a tree is labelled $\langle 0, 1 \rangle$. If vertex v is labelled $\langle \pi_v, \mu_v \rangle$, then its left child is labelled $\langle \pi_v, 2\mu_v \rangle$, and its right child is labelled $\langle \pi_v \vee \mu_v, 2\mu_v \rangle$. Note that if v is at depth d, then $\mu_v = 2^d$. Also note that the d least significant bits of π_v, read from least to most significant bit, describe the path from the root to v, with a 0 bit indicating a left branch and a 1 bit indicating a right branch.

In order to simplify memory allocation issues, every array A_v is allocated $O(\log \hat{n})$ entries. This means that the binary tree representing a list of s elements takes $O(s \log \hat{n})$ space. A collection of lists with a total of at most n elements takes $O(n \log \hat{n})$ space. Furthermore, for each vertex v, the integers used in the field $\langle \pi_v, \mu_v \rangle$ fit into an $O(\log n)$ bit word.

4.3 Computing rmin and prec

For any two leaves x, y, where x is to the left of y, the answer to the query $rmin(X, x, y)$ can be computed as follows. Let $\gamma = \pi_x \oplus \pi_y$. Let d be the position of the least significant 1 bit in the binary representation of γ. Observe that this is the position where the paths to x and y diverge. Thus, d is the depth of $lca(x, y)$. It is easily shown that if $\lambda = \gamma \wedge \neg(\gamma - 1)$, then $\lambda = 2^d$. Let x' be the ancestor of x at depth $d+1$ and y' be the ancestor of y at depth $d+1$ (see Fig. 1). By definition $A_x[d+1]$ contains the minimum element in $(x, \ldots, r(x'))$ and $A_y[d+1]$ contains the minimum element in $[\ell(y'), \ldots, y)$. It follows directly that the answer to the query $rmin(X, x, y)$ is $\min(x, A_x[d+1], A_y[d+1], y)$.

To compute the answer to the query $rmin(X, x, y)$ for any two elements x, y in a list X, we need to be able to determine if x precedes y in the list (i.e. check if $prec(X, x, y)$ is true). Assume without loss of generality that $x \neq y$. Compute λ as described above. The leaf x precedes y in the list X if and only if $\pi_x \wedge \lambda = 0$ (i.e. bit d of π_x is 0). To see this, note that the path from the root to x takes a left branch at $lca(x, y)$ exactly when bit d of π_x is 0.

Using the above algorithms both *rmin* and *prec* can be computed in constant time by one processor without performing any writes. Furthermore, because no writes are performed for either operation, there is no difficulty in performing multiple queries in parallel on a CREW PRAM.

4.4 Rebalancing and Initializations

As elements are inserted into or deleted from a list X, the tree representing the list must continue to have depth $O(\log |X|)$. Following Nievergelt and Reingold [1973], we use a brute force rebalancing scheme that replaces any potentially unbalanced subtree with a perfectly balanced subtree. For each vertex v, store the size of B_v in $v.size$. If v is a leaf then define the *root balance* $\rho(v)$ to be $1/2$, otherwise define $\rho(v) = v.left.size/v.size$. For a given constant $\alpha \in (0, 1/2)$, a subtree B_v is balanced if and only if $\alpha \leq \rho(v) \leq 1-\alpha$. (This criteria can be replaced by any of the criteria defined for weight balanced trees. See, for example, Andersson 1989.)

To rebalance a subtree B_v, we replace it with a balanced subtree and recompute the data structure on those vertices. Let $s = |B_v|$. Since the depth of a tree increases by at most one during a round of insertions, and all subtrees were balanced before the insertions, it follows that, even before rebalancing, the subtree B_v will have depth $O(\log s)$. Therefore, the vertices of B_v can be collected into an array in $O(\log s)$ time using s processors by recursively descending the tree. The vertices can be formed into a tree of depth $\lceil \log s \rceil$ in constant time using s processors [Moitra and Iyengar 1986]. Let w be the root of the resulting tree. For any vertex z, the field $\langle \pi_z, \mu_z \rangle$ can be computed from z's parent, if any, in constant time. Thus, these fields can be computed for all vertices z in B_w in a total of $O(\log s)$ time using s processors by descending the tree. Similarly, for any vertex z the fields $z.min$ and $z.size$ can be computed from the children of z, if any, in constant time. Thus, these fields can be computed for all vertices z in B_w in a total of $O(\log s)$ time using s processors by ascending the tree. It remains to describe how to compute A_x for each leaf x in B_w. The following procedure computes A_x using one processor in $O(depth(x))$ time.

$R' \leftarrow 0$, $L' \leftarrow 0$, $v \leftarrow x$
for $i \leftarrow depth(x)$ down to 1 do
  loop invariant: $L' = \min[\ell(v), \ldots, x)$ and $R' = \min(x, \ldots, r(v)]$.
  if v is a right child then $A_x[i] \leftarrow L'$; $L' \leftarrow \min(v.parent.left.min, L')$.
  if v is a left child then $A_x[i] \leftarrow R'$; $R' \leftarrow \min(R', v.parent.right.min)$.
  $v \leftarrow v.parent$

We show that the loop correctly initializes $A_x[1, \ldots, depth(x)]$. Let x_i denote x's ancestor at depth i. It is clear that in round i, $v = x_i$. Furthermore, if the loop invariant holds it is clear that $A_x[i]$ is computed correctly for each $1 \leq i \leq depth(x)$. The loop invariant holds trivially for $i = depth(x)$. Assume the loop invariant holds for i, we show it holds for $i - 1$. We

consider the case where v is a right child. The case where v is a left child is symmetrical. If v is a right child, then $r(v) = r(v.parent)$, so

$$(x, \ldots, r(v.parent)] = (x, \ldots, r(v)] = R',$$

and thus R' is correct for iteration $i - 1$. Furthermore,

$$[\ell(v.parent), \ldots, x) = [\ell(v.parent.left), \ldots, r(v.parent.left)] \cup [\ell(v), \ldots, x),$$

so

$$\min[\ell(v.parent), \ldots, x) = \min(v.parent.left.min, L'),$$

and thus L' is correct for iteration $i - 1$.

Since all trees have depth $O(\log \hat{n})$, it follows that the total time to rebalance a subtree B_v is $O(\log \hat{n})$ time using $s = |B_v|$ processors.

This procedure can also be used to perform *initialize-min* operations. Let $R_1, \ldots, R_{\hat{q}}$ be the requested initializations, where the request $R_i =$ *initialize-min*$(X_i, x_{i,1}, \ldots, x_{i,s_i})$ specifies that the list X_i should be initialized to contain $x_{i,1}, \ldots, x_{i,s_i}$. Let q be the total number of elements in the initialization requests. By simulating q processors and assigning s_i processors to the request R_i, the $\hat{q}$ data structures can be initialized in a total of $O(q \log \hat{n}/p + \log p)$ time on p processors, where $O(q \log \hat{n}/p + \log p)$ time is used to allocate the memory for the data structures using the algorithm of section 2.6, and $O(q/p + \log p)$ time is used to assign processors to their tasks.

4.5 Collect Operations

Let $R_1, \ldots, R_{\hat{q}}$ be the requested collect operations, where the request $R_i =$ *collect*(X_i, C_i) specifies that the elements in the list X_i should be copied into the array C_i in left to right order. Compute the prefix sums of $X_i.root.size$, for $1 \leq i \leq \hat{q}$. Using this information, $X_i.root.size$ virtual processors can be assigned to each list X_i. These processors then recursively descend the tree to assign a single processor to each leaf, i.e. a block of $v.size$ contiguous processors assigned to a vertex v splits itself into two sub-blocks of size $v.left.size$ and $v.right.size$ and assigns the sub-blocks to the left and right children of v, if any. When a single processor is assigned to each leaf, each processor copies its leaf into the appropriate array C_i at the position indicated by the processor's identifier.

Let q be the total number of elements collected. Since all trees have depth $O(\log \hat{n})$, the complete algorithm takes $O((q/p + 1) \log \hat{n} + \log p)$ time using p processors.

4.6 Insertions

Let $R_1, \ldots, R_q$ be the requested insertion operations, where the request $R_i =$ *list-insert*$(X_i, x_i, y_i, side_i)$ specifies that the new leaf x_i is to be inserted to the left or right of y_i in X_i depending on whether $side_i =$ "left" or

$side_i$ = "right". The insertions are dealt with in two stages. In stage one, the new leaves are inserted into the specified trees and the data structure is extended to these new leaves. In stage two, subtrees are rebalanced if necessary.

The insertion algorithm begins by allocating a record for each new leaf x_i, and allocating a corresponding record z_i for the new internal node that will be the parent of x_i and y_i. Using the algorithm of section 2.6, this can be accomplished in $O(q \log \hat{n}/p + \log p)$ time using p processors. Next, for all requests R_i, we replace the leaf y_i with a three node subtree rooted at z_i having x_i and y_i as leaves, where the order of the leaves is determined by the value of $side_i$. This can be performed in constant time using q processors. Having constructed these replacement trees, we compute the fields for all their vertices using the same method as in the rebalancing procedure. This takes a further $O(\log \hat{n})$ time using q processors.

The information stored in the vertices of the replacement subtrees is correct by construction. We must show that the rest of the data structure is still correct. Consider any vertex v not in any replacement subtree. Since the path from v to the root is unchanged, $\langle \pi_v, \mu_v \rangle$ remains correct trivially. Before the insertions, $v.min$ was correct. Any new leaf x_i inserted into B_v must have a value greater than its corresponding insertion point y_i, which was in the subtree B_v before the insertions. Hence $value(x_i) > v.min$. It follows that $v.min$ is correct for all vertices. Now suppose v is a leaf. Before the insertions $A_v[i]$ was correct for all $1 \leq i \leq depth(v)$. Assume $A_v[i]$ is now incorrect for some i, and let v_i denote the ancestor of v at depth i. We deal with the case where v_i is a right child. The case where v_i is a left child is symmetrical. If v_i is a right child, then it must be the case that there is a new minimal element in the list $[\ell(v_i), \ldots, v)$. By assumption, the insertion point of this element is not v, so the insertion point must also be in the list $[\ell(v_i), \ldots, v)$. However, this is impossible, since any new leaf must have value greater than its insertion point. Hence $\min[\ell(v_i), \ldots, v)$ must be unchanged and $A_v[i]$ is correct.

It remains to update subtree sizes and carry out rebalancing. For each z_i, it is necessary to recompute $v.size$ at every ancestor v of z_i. This is done by having one processor walk up the tree from each z_i and compute $v.size$ at each ancestor of z_i from the sizes of its children. Some care must be taken in scheduling to avoid write conflicts. One way to avoid conflicts is to alternate moves up from left and right children and have processors taking a path up from a right child halt if they discover that another processor has just travelled up from the left child. If all processors leave marks on the vertices they examine, then a processor that is examining a vertex v can easily discover if another process is currently at v's sibling. A similar process is used to find, for each z_i, the highest unbalanced subtree on the path from z_i to the root of the tree z_i occurs in. Let W be the set of unbalanced subtrees found in this search. Since the subtrees in W are disjoint they can be rebalanced in parallel. Updating sizes and collecting W can be done in $O(\log \hat{n})$ time using q processors. Let c be the total number of vertices in

the subtrees to be rebalanced. The rebalancing step can be performed in a further $O(\log \hat{n})$ time using c processors. By the same argument as used above, the data structure remains correct after the rebalancing.

The complete algorithm takes a total of $O(((q + c)/p + 1) \log \hat{n} + \log p)$ time using p processors.

4.7 Deletions

Let $R_1, \ldots, R_q$ be the requested delete operations, where the request $R_i =$ *list-delete*(X_i, x_i) specifies that the element x_i is to be deleted from the list X_i. As is the case with insertions, deletions proceed in two stages. In stage one, the vertices are deleted from the tree and the fields in some vertices are recomputed to maintain correctness. In stage two, subtrees are rebalanced if necessary.

Deleting a single leaf is done by removing both the leaf and its parent and updating its siblings parent pointer. This cannot be done in the presence of parallel deletions, since the sibling may also be deleted. This difficulty can be avoided by alternating deletions between left and right children. Note that as many as $O(\log \hat{n})$ rounds of alteration may be necessary, since after deleting all left children, some right children may become left children. During the deletion process, subtrees rooted at siblings of deleted vertices are moved up in the tree. Because of this, the fields in the vertices of these subtrees must be updated. Note that the same subtree may be moved more than once.

We must show how to find the subtrees that must be updated. Let z_i be the sibling of x_i at the time that x_i is deleted. The roots of the subtrees that must be updated are contained in the list $Z = z_1, \ldots, z_q$. However, this list may contain some extra vertices. There are three possible problems.

(1) Some vertices in the list Z may have been deleted after they were recorded. All such vertices must be removed from Z. If processors mark the vertices they delete, then the deleted vertices in Z can easily be identified in constant time using q processors.

(2) Some vertices may occur more than once in the list Z. This happens when a vertex is moved by more than one delete operation. All but one occurrence of each vertex must be removed from Z. If each processor i marks the sibling of x_i at the time x_i is deleted with its identifier i, then for each v in Z there is exactly one $z_i = v$ such that v is marked with i. Thus, the extra copies of each vertex can be identified in constant time using q processors.

(3) Some vertices may be descendants of other vertices in the list. All such vertices must be removed from Z. These vertices can be found in $O(\log \hat{n})$ time using q processors by marking each vertex in the list and ascending the tree looking for marked vertices.

Once the vertices that must be thrown away have been identified, they can be removed from Z by array compaction in $O(q/p + \log p)$ time using p

processors. The rebalancing procedure is used to correct the fields in the subtrees rooted at the remaining vertices in Z. Since the trees were originally balanced, and since each such subtree was a sibling of a deleted vertex, the total number of vertices in these subtrees is $O(q)$. Thus, this rebalancing takes $O(\log \hat{n})$ time using q processors.

A similar argument to that used for insertions shows that the data structure is correct after the first stage of deletions. Updating subtree sizes and performing any necessary rebalancing is done exactly as for insertions. If c is the number of vertices in subtrees that are rebalanced, then this takes $O(\log \hat{n})$ time using c processors.

Using the algorithm of section 2.6, memory that is no longer in use can be deallocated at the end of the deletion algorithm in $O(q \log \hat{n}/p + \log p)$ time using p processors.

The complete algorithm takes a total of $O(((q + c)/p + 1) \log \hat{n} + \log p)$ time using p processors.

4.8 Complexity

Let the maximum number of elements contained in the entire collection of lists at any one time be denoted by n, and let the maximum number of elements contained in any one list at any one time be denoted by $\hat{n}$.

THEOREM 2. *Given n and $\hat{n}$, any sequence of m cooperative dynamic restricted range minimum operations presented over r rounds can be processed on a CREW PRAM in $O(m \log^2 \hat{n}/p + r \log p + r \log \hat{n})$ time and $O(n \log \hat{n})$ space using p processors. In addition, rmin and prec queries can be processed in constant time using a single processor without performing any writes.*

PROOF. The stated bound on space has already been established, as have the stated bounds on the time taken by *rmin* and *prec* queries. It remains to establish the time bound on cooperative operations.

Let m_i denote the number of cooperative operations requested in round i. Note that $\sum_{i=1}^{r} m_i = m$. Let c_i be the number of vertices in subtrees rebalanced in round i. By the balance criteria, a subtree rooted at v is rebalanced only if one of its subtrees has become a constant fraction bigger than the other. Let the number of insertions and deletions since the last rebalancing be denoted respectively by I and D, and let S be the size of B_v at the last rebalancing. We claim that $v.size \in O(I + D)$. Assume without loss of generality that the left subtree has become bigger than the right. If v is unbalanced and heavier on the left than the right, then $v.left.size/v.size > 1 - \alpha$. Now, $v.left.size \leq \lceil S/2 \rceil + I$ and $v.size = S + I - D$, so we have $(\lceil S/2 \rceil + I)/(S + I - D) > 1 - \alpha$. Simplifying we obtain $S < (I + D + 1)/(\frac{1}{2} - \alpha)$. The claim follows. Now, since each insertion or deletion occurs in at most $O(\log \hat{n})$ subtrees, it follows that $\sum_{i=1}^{r} c_i \in O(m \log \hat{n})$.

From the complexity results derived in the previous subsections we have that round i takes $O(((m_i + c_i)/p + 1) \log \hat{n} + \log p)$ time using p processors.

Summing the time for r rounds on a p processor CREW PRAM, we obtain a total time of $O(m \log^2 \hat{n}/p + r \log \hat{n} + r \log p)$. □

4.9 The Grouping Problem

In the work optimal construction given in section 6, it will be necessary to group together requests for operations on the basic structure that are for the same list. To be precise, we must solve the following problem.

The Grouping Problem: Given an array L of pairs, such that all the second components are distinct, construct an array L' that is a permutation of the original array L such that all pairs with the same value in the first component occur contiguously.

In our application the pairs of the array are of the form $\langle X_i, x_i \rangle$, where X_i is the name of a list represented using the basic structure, and x_i is an element of the list X_i. Note that each x_i is a unique leaf in the tree representing X_i.

We solve the grouping problem in two stages. In the first stage we assign a processor to each x_i and ascend the tree collecting together the leaves owned by other processors that are ascending the same tree. Whenever two processors meet at a vertex they combine their collections and only one processor proceeds to the root. At the end of the first stage we have a collection of linked lists, one for each unique list X that occurs in L. In the second stage we use list ranking on this set of linked lists to place the leaves into the array L'.

The first stage of this construction finishes in $O(\log \hat{n})$ time using s processors, where s is the number of pairs in the array L. The second stage takes $O(s/p + \log p)$ time using p processors. The total is $O(s \log \hat{n}/p + \log p)$ time using p processors.

5. An Optimal Algorithm for Small Lists

The algorithm presented in section 4 performs large amounts of work both in rebalancing, and in computing the A_v arrays for new vertices v. This section presents an algorithm that avoids this work for a collection of lists, where each list has length at most b, where $(2b + 2)\lceil \log(b + 1) \rceil \in \log n + O(1)$, and where n is the maximum number of elements in the entire collection at any one time. The general approach is to precompute all possible answers for all possible lists of this size. The size b is chosen so that the cost of the precomputation is subsumed in the cost of the overall computation. Note that, for the purposes of this algorithm, the value of n must be known in advance. The data structure presented in this section will be referred to as the *micro structure*. The next section will combine this algorithm with the previous algorithm to obtain an optimal algorithm for the dynamic restricted range minimum problem. In addition, the requirement that n be known will be removed.

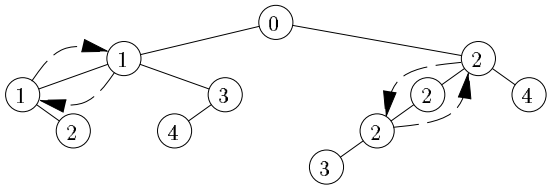

Fig. 2: An augmented Cartesian Tree

5.1 Representing Lists by Cartesian Trees

Gabow *et al.* [1984] observed that the problem of finding range minima in a fixed list can be reduced to finding lowest common ancestors in a Cartesian tree. The Cartesian tree for the list of elements $[x_1, \ldots, x_s]$ is the binary tree with $x_1, \ldots, x_s$ as the vertices defined as follows. The (rightmost) minimal element in X, call it x_m, is the root of the tree. The left subtree is the Cartesian tree for $[x_1, \ldots, x_{m-1}]$ and the right subtree is the Cartesian tree for $[x_{m+1}, \ldots, x_s]$. It follows from this definition that $rmin(x, y)$ in the list is the same element as $lca(x, y)$ in the Cartesian tree. This structure is the basis of the algorithm presented in this section.

We call a vertex v in the Cartesian tree a *head vertex* either if its value is not the same as its parent's or if it is the root of the tree. We call a vertex v in the Cartesian tree a *tail vertex* either if its value is not the same as its left child's, or if it has no left child. Note that each head vertex is joined to a tail vertex by a chain of left children, such that all vertices in the chain have the same value. We say that a head and tail vertex joined by such a chain *correspond*. Also note that the parent of a vertex v in the Cartesian tree has a value of at most $value(v)$.

The *rank* of a vertex in a Cartesian tree is the number of vertices that occur to its left in the tree.

5.2 A Data Structure for Small Lists

Each list in the collection is represented independently. A list $[x_1, \ldots, x_s]$ is represented by the corresponding Cartesian tree as described above, together with a compact one word representation of the Cartesian tree. The compact representation is used to answer queries by doing table lookups.

We elaborate upon the structure for a list X. Each vertex in the Cartesian tree has pointers to its parent and its left and right children, if any. To support fast insertions the Cartesian tree is augmented so that each head vertex v has a pointer $v.tail$ pointing to the corresponding tail vertex, and each tail vertex has a pointer $v.head$ pointing to the corresponding head vertex (See Fig. 2). The field $X.micro$ stores a word that is divided into $2b$ fields of size $\lceil \log_2(b+1) \rceil$ bits each. Following Gabow and Tarjan [1985] this field is used to store a compact representation of the Cartesian tree of

the list X. First, each vertex v is assigned a unique index $v.index$ from the set $\{1, \ldots, s\}$. For each $1 \leq i \leq b$, the field $2i-1$ stores the index of the left child of the vertex with index i, and the field $2i$ stores the index of the right child of the vertex with index i. If the vertex with index i has no left or right child, then the corresponding field is set to 0. To allow access to a vertex given its index the array $X.map$ stores a pointer to the vertex with index i in $X.map[i]$. As the list grows, the array $X.map$ will need to be periodically reallocated. Toward this end, the field $X.array\text{-}size$ stores the current size of $X.map$, and the field $X.size$ will store the current number of elements in the list X. Whenever $X.size > X.array\text{-}size$ or $2X.size < X.array\text{-}size$, $X.array\text{-}size$ is set to $2X.size$ and the array $X.map$ reallocated so that its length is $X.array\text{-}size$. Note that as vertices are deleted from a list it is important that the remaining vertices have indices in the range $[1, X.size]$, otherwise it will not be possible to reallocate $X.map$ when it shrinks.

Finally, there are two precomputed lookup tables: LCA and $RANK$, where $LCA[T, i, j]$ stores the least common ancestor of the vertices with indices i and j in a tree represented by the word T, and $RANK[T, i]$ stores the rank of the vertex with index i in a tree represented by the word T.

Due to the assumed bounds on b, each field in this structure can be represented in a single $O(\log n)$ bit word. In addition the lookup tables LCA and $RANK$ take a total of $O(n)$ words of space. Finally, the representation for a collection of lists having at most n elements at any one time takes a further $O(n)$ space, giving a total space consumption of $O(n)$.

5.3 Computing the Lookup Tables

At the start of the computation the lookup tables LCA and $RANK$ must be computed. Both tables can be constructed in $O(b^2)$ time using n/b^2 processors, giving $O(\min(n, n/p + b^2))$ time using p processors.

Each processor is assigned one of the possible $2b\lceil \log(b+1) \rceil$ bit values. Observe that there are at most n/b^2 such values. The construction using a single processor proceeds as follows. Let T be the $2b\lceil \log(b+1) \rceil$ bits assigned to the processor. First the processor checks that T represents a Cartesian tree, i.e. that the representation contains no cycles, and each vertex has only one incoming edge. This can be done in $O(b)$ time. Next the processor builds a list of the leaves as they appear from left to right in the Cartesian tree T. The values $RANK[T, 1], \ldots, RANK[T, b]$ can be computed from this list in $O(b)$ time. Finally the values $LCA[T, i, j]$, for all $1 \leq i, j \leq b$, can be computed in a further $O(b^2)$ time.

5.4 Computing rmin and prec

The answer to a query $rmin(X, x, y)$ can be computed by looking up the value $LCA[X.micro, x.index, y.index]$ and returning the vertex with this index. Likewise, the answer to a query $prec(X, x, y)$ can be computed by comparing $RANK[X.micro, x.index]$ and $RANK[X.micro, y.index]$. Both op-

erations take only constant time using a single processor. Furthermore, because no writes are performed for either operation, there is no difficulty in performing multiple queries in parallel on a CREW PRAM.

5.5 Collect Operations

A collect operation, *collect*(X, C), on a list X with s elements can be done in $O(s)$ time using one processor by looking up $RANK[X.micro, i]$ for each index i in $[1, X.size]$, and assigning the vertex with index i to the indicated position in the array C.

A round of $\hat{q}$ collect operations, $R_1, \ldots, R_{\hat{q}}$, where $R_i = collect(X_i, C_i)$, can be treated as $\hat{q}$ independent tasks of maximum size b. If there are a total of q elements to be collected, then using the load balancing procedure discussed in section 2.5, these operations can be performed in $O(\min(q, q/p+\log p+b))$ time using p processors.

5.6 Initializations

The construction of a Cartesian tree for a list $[x_1, \ldots, x_s]$ can be accomplished with the following algorithm of Gabow *et al.* [1984]. Start with the empty Cartesian tree. To derive the Cartesian tree on the first i elements from the Cartesian tree on the first $i-1$ elements, ascend the path from the rightmost vertex to the root until x_k, the first node smaller than x_i, is found; the right subtree of x_k is made into the left subtree of x_i and x_i is made into the right child of x_k. The total time to construct the Cartesian tree is $O(s)$, since a vertex leaves the rightmost path immediately after it is traversed. Using this algorithm an initialization request *initialize-min*$(X, x_1, \ldots, x_s)$ can be processed in $O(s)$ time using one processor.

Consider a round of $\hat{q}$ initialize operations, $R_1, \ldots, R_{\hat{q}}$, initializing a total of q elements, where $R_i = initialize\text{-}min(X_i, x_{i,1}, \ldots, x_{i,s_i})$. A total of $O(q)$ space is required for the data structures that are being initialized. This memory can be allocated in $O(\min(q, q/p + \log p))$ time using p processors (see subsection 2.6). The application of the above initialization procedure can be treated as $\hat{q}$ independent tasks of maximum size b. Using the load balancing procedure discussed in section 2.5 these operations can be performed in $O(\min(q, q/p + \log p + b))$ time using p processors. In total the round takes $O(\min(q, q/p + \log p + b))$ time using p processors.

5.7 The Grouping Problem

In order to efficiently deal with insertions and deletions it will be necessary to group together requests for operations on the micro structure that are for the same list. This operation will also be needed in the optimal algorithm given in section 6. We reiterate the problem that must be solved.

The Grouping Problem: Given an array L of pairs, such that all the second components are distinct, construct an array L' that is a permutation of the

original array L such that all pairs with the same value in the first component occur contiguously.

In our application the pairs of the array are of the form $\langle X_i, x_i \rangle$, where X_i is the name of a list represented using the micro structure, and x_i is an element of the list X_i.

If $s < b$, then there is no reason to parallelize the solution, and we simply perform a bucket sort on one processor in $O(s)$ time. Otherwise, we must make use of some parallelism.

We take advantage of the fact that within the list X_i, the element x_i has a unique index in $\{1, \ldots, X.size\}$. The construction begins by sorting L by these indices. Reif [1985] gives an algorithm that sorts s elements drawn from a set of size b in $O(b)$ time on an EREW PRAM using s/b processors. Let L_j denote the contiguous sub-array of elements with index j. The ends of the sub-arrays $L_1, \ldots, L_{X.size}$ can be determined from the sorted list in constant time using s processors. Observe that for all j, all pairs in L_j are unique, i.e. no list X occurs in more than one entry of L_j. Thus we can quickly construct linked lists of pairs that refer to the same list by successively assigning processors to all the entries in L_j for each $j = 1, \ldots, X.size$. Furthermore, the length of each linked list and the rank of each element are determined. Using s processors this can be done in $O(s/p + b)$ time. Prefix sums can be used to determine the allocation of space for each linked list in $O(s/p + \log p)$ time using p processors. The elements can then be packed into an array in a further $O(s/p)$ time using p processors. The total time to solve the grouping problem with this technique is $O(s/p + \log p + b)$ time using p processors.

Combining the single processor algorithm with the parallel algorithm, we get a final complexity of $O(\min(s, s/p + \log p + b))$ time to perform grouping using p processors.

5.8 Insertions

Consider the insertion of a new element x next to an existing element y. If x is to be inserted as an endpoint of the list, it is simply made a child of y. Otherwise, suppose that x is inserted inserted between y and z, where z is a neighbor of y. We consider the case where z is a right neighbor of y; the case for the left neighbor is symmetrical. Observe that if $value(y) > value(z)$, then y has no right child in the Cartesian tree and otherwise z has no left child. There are two sub-cases to be dealt with. If $value(x) > value(z)$ then x becomes either a right child of y or a left child of z, whichever has greater value. Otherwise, $value(y) < value(x) \leq value(z)$, which implies that y is an ancestor of z. The new vertex x must be inserted on the path from y to z as the left child of the deepest vertex v with value no less than the value of x. The old left child of v becomes the right child of x. Because the conditions on insertions require that $value(x)$ is either within k of $value(y)$ or within k of $value(z)$, the vertex v can be found in $O(k)$ time by following a path from either y or z (whichever has the value closest to $value(x)$). Note that long

runs of vertices with the same weight must be skipped using the *head* and *tail* pointers. Provided that the array $X.map$ is large enough, a group of s insertions on a list X can be processed using this algorithm in $O(s)$ time using one processor.

Consider a round of q insertion operations, $R_1, \ldots, R_q$, where the operation $R_i = \textit{list-insert}(X_i, x_i, y_i, \textit{side}_i)$. First, the grouping algorithm of subsection 5.7 is used to group together requests for the same list. This takes $O(\min(q, q/p + \log p + b))$ time using p processors. After the grouping operation, the size of each unique X in $X_1, \ldots, X_q$ can updated in a further $O(q/p)$ time using p processors. The memory for the records to store the q new elements can be allocated in $O(\min(q, q/p + \log p))$ time using p processors. Next, all arrays $X_i.map$ that have become too small are reallocated. Let c be the total number of elements in the reallocated arrays after reallocation. The necessary memory allocation, deallocation and copying from the old arrays to the new arrays can be performed in $O(\min(c, c/p + \log p))$ time using p processors. The main work of the insertions, performed by the algorithm given above, can be treated as at most q independent tasks of maximum size b and of total size q. Using the load balancing procedure discussed in section 2.5 these tasks can be performed in $O(\min(q, q/p + \log p + b))$ time using p processors. In total the round takes $O(\min(c+q, (c+q)/p + \log p + b))$ time using p processors.

5.9 Deletions

Any element to be deleted is a local maximum in the list, hence it is necessarily a leaf in the Cartesian tree, and so can be removed in constant time. The only difficulty is that once a vertex v is deleted the index $v.index$ does not have an associated vertex, and this may introduce a hole in the set of indices. This problem can be avoided by first swapping indices with the vertex that has the largest index. This will also require some adjustments in $B.micro$, but these can easily be performed in constant time. Using this procedure a sequence of s deletions in a list can be performed in $O(s)$ time using one processor.

Consider a round of q delete operations, $R_1, \ldots, R_q$, where the operation $R_i = \textit{list-delete}(X_i, x_i)$. First, the grouping algorithm of subsection 5.7 is used to group together requests for the same list. This takes $O(\min(q, q/p + \log p + b))$ time using p processors. The main work of deletion, performed by the algorithm given above, can be treated as at most q independent tasks of maximum size b and of total size q, with one task for each group found by the grouping algorithm. Using the load balancing procedure discussed in section 2.5 these tasks can be performed in $O(\min(q, q/p + \log p + b))$ time using p processors. After the deletions have been processed, the size of each unique X in $X_1, \ldots, X_q$ can updated in a further $O(q/p)$ time using p processors. Finally, all arrays $X.map$ that have become too large are reallocated. Let c be the total number of elements in the reallocated arrays before reallocation. The necessary memory allo-

cation, deallocation and copying from the old arrays to the new arrays can be performed in $O(\min(c, c/p + \log p))$ time using p processors. In total the round takes $O(\min(c+q, (c+q)/p + \log p + b))$ time using p processors.

5.10 Complexity

Let the maximum number of elements contained in the entire collection of lists at any one time be denoted by n, and let the maximum number of elements contained in any one list at any one time be denoted by b.

THEOREM 3. *Given n and b, where $(2b+2)\lceil\log(b+1)\rceil \in \log n + O(1)$, any sequence of m cooperative dynamic restricted range minimum operations, presented over r rounds, can be processed on a CREW PRAM in $O(m/p + \min(m, r\log p + rb + b^2))$ time and $O(n)$ space on using p processors. In addition, rmin and prec queries can be processed in constant time using a single processor without performing any writes.*

PROOF. The stated bound on space has already been established, as have the stated bounds on the time taken by *rmin* and *prec* queries. It remains to establish the time bound on cooperative operations.

Let m_i denote the number of cooperative operations requested in round i. Note that $\sum_{i=1}^{r} m_i = m$. Let c_i denote the total number of memory cells that are either allocated or deallocated during round i as part of the reallocation of an array $X.map$ for some list X. An array $X.map$ is reallocated in round i if the portion in use has changed by a factor of 2 since its last reallocation. Using this fact is is easily established that $\sum_{i=1}^{r} c_i \in O(m)$. Now, the running time for round i using p processors is $O(\min(c_i + m_i, (c_i + m_i)/p + \log p + b))$. Summing the time for each round over the r rounds gives a total cost of $O(m/p + \min(m, r\log p + rb))$ time using p processor. Including the $O(\min(n, n/p + b^2))$ time using p processors to initialize the lookup tables, we obtain a total of $O(m/p + \min(m, r\log p + rb + b^2))$ time using p processors. □

REMARK 3. It will be necessary to apply this theorem for $b = c(\log\log n)^2$, for some positive constant c. Observe that

$$\begin{aligned}(2b+2)\lceil\log(b+1)\rceil &\leq 2(b+1)(\log(b+1)+1)\\ &\leq 2(2b)(2b)\\ &\leq 8b^2.\end{aligned}$$

Hence, it is sufficient to show there is a constant c, such that for $n > 0$, $8c(\log\log n)^4 \leq \log n$. Since $(\log\log n)^4 \in o(\log n)$ this is clearly the case.

6. An Optimal Algorithm

Examining the algorithm of section 4 shows that much of the work performed arises from always rebalancing an entire subtree, despite the fact that the

subtrees near the leaves are already balanced. This cost can be circumvented by reducing the dynamic restricted range minimum problem to a smaller version of itself. Combining this approach with the algorithm for small lists presented in section 5 leads to a work optimal algorithm.

6.1 Preliminaries

Assume there exist two algorithms, A_1 and A_2 that solve the dynamic restricted range minimum problem, and furthermore that we can solve the grouping problem on their respective data structures. These algorithms will be used to construct a new algorithm parameterized by a function $f(n)$. The complexity of the construction will be expressed in terms of its running time excluding computations performed by A_1 and A_2, plus the number of operations performed by A_1 and A_2. The construction described in the following subsections also assumes that the value of n, the maximum number of elements ever in the collection of lists at any one time, is known in advance. The final subsection shows that this assumption can be removed.

6.2 The Data Structure

Each list X in the collection is represented by a collection of sublists of X. Say X contains $[x_1, \ldots, x_s]$. Then it is partitioned into the sublists $G_1, \ldots, G_{\lceil s/f(n) \rceil}$ each of size $\mathrm{O}(f(n))$. For each sublist G_i three pointers are maintained: $G_i.head$, $G_i.tail$ and $G_i.min$, that point to the head, the tail, and the minimum element of the sublist G_i respectively. As well, the field $G_i.size$ records the number of elements in the sublist. The field $X.first$ points to G_1. Each element x_i also has a pointer $x_i.sublist$ that points to the sublist it is contained in. The sublists are maintained using algorithm A_2. If $s \geq f(n)$ a second list named $X.reduced$ containing $[b_1, \ldots, b_{\lceil s/f(n) \rceil}]$ is maintained by algorithm A_1, where $b_i = G_i.min$. Note that each b_i occurs in two lists, G_i and $X.reduced$. Also, the list $X.reduced$ is doubly linked so that for each b_i, $b_i.next$ and $b_i.prior$ point to the next and prior elements in the list.

Using this construction, the total space required to store a collection of lists containing a total of at most n elements at any one time is $\mathrm{O}(n)$ space plus the space required by A_1 to represent a collection of lists containing at most a total of $n/f(n)$ elements, with no list larger than $n/f(n)$ elements, plus the space required by A_2 to represent a collection of list containing at most a total of n elements, with no list larger than $2f(n)$ elements.

6.3 Computing rmin and prec

A request $prec(X, x, y)$ can be satisfied as follows. If $x.sublist = y.sublist$, then it can be computed in the sublist $x.sublist$ using algorithm A_2. Otherwise, $prec(x, y)$ is the same as $prec(X.reduced, x.sublist.min, y.sublist.min)$ computed using algorithm A_1.

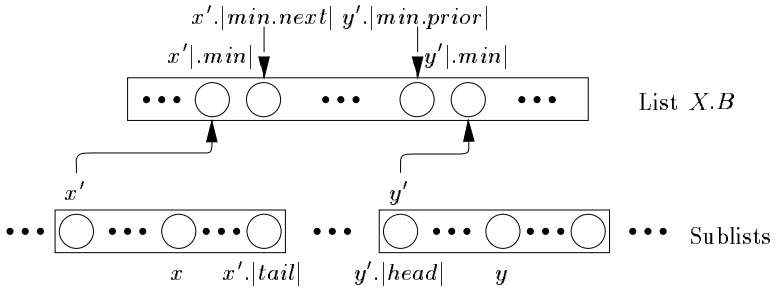

Fig. 3: Computing $rmin(x, y)$ in the reduction structure.

A request $rmin(X, x, y)$ can be satisfied as follows. Without loss of generality, suppose that x occurs to the left of y in the list. For notational simplicity let x' denote $x.sublist$ and let y' denote $y.sublist$. If x and y are in the same sublist, i.e. $x' = y'$, then the request can be satisfied directly using algorithm A_2 on sublist x'. Otherwise, there are three possibilities (see Fig. 3). The minimum could be in the list x' between x and $x'.tail$, it could be in the list y' between y and $y'.head$, and it could be the minimum element in the list $X.reduced$ between $x'.min.next$ and $y'.min.prior$. The first two of these can be found using algorithm A_2, the last by using algorithm A_1.

Using these algorithms an *rmin* or *prec* query can be processed using one processor in constant time plus the sum of the time required to process a constant number of queries with both algorithms A_1 and A_2. Provided algorithms A_1 and A_2 can perform queries without performing writes, the combined construction can perform queries without performing any writes.

6.4 Collections

A round of $\hat{q}$ requests, $R_1, \ldots, R_{\hat{q}}$, for *collect* operations on a total of q elements, where $R_i = collect(X_i, C_i)$, is performed in two stages. In the first stage collect operations are performed on lists with less than $f(n)$ elements using algorithm A_2 alone. In the second stage collect operations are performed on longer lists.

Let $X_{\sigma(1)}, \ldots, X_{\sigma(c)}$ be those X_i with less than $f(n)$ elements, i.e. those X_i with an empty list $X_i.reduced$. This set can be found in $O(\hat{q}/p + \log p)$ time using p processors. The corresponding collect operations $collect(X_{\sigma(i)}, C_{\sigma(i)})$, $1 \leq i \leq c$, are performed using algorithm A_2.

Let $X_{\tau(1)}, \ldots, X_{\tau(c')}$ be those X_i with at least $f(n)$ elements, i.e. those X_i with a non empty list $X_i.reduced$. This set can be found in $O(\hat{q}/p + \log p)$ time using p processors. For each $1 \leq i \leq c'$, a collect operation is performed on $X_{\tau(i)}.reduced$ using algorithm A_1. Let $s_{\tau(i)}$ denote the number of elements collected from list $X_{\tau(i)}.reduced$, and let $S = \sum_{1 \leq i \leq c'} s_{\tau(i)}$. A prefix sums

operation on the sizes of the $S < q$ sublists $X_{\tau(i)}.G_j$, $1 \leq i \leq c'$ and $1 \leq j \leq s_{\tau(i)}$, collected in the first step can be used to allocate space in each array $C_{\tau(i)}$ for the elements in the sublists. This requires $O(q/p + \log p)$ time using p processors. Finally, using algorithm A_2 collect operations can be performed on each list $X_{\tau(i)}.G_j$, placing the results into the appropriate place in the array $C_{\tau(i)}$.

The total running time of the collect procedure is $O(q/p+\log p)$ time using p processors, plus the time to perform *collect* operations on at most $q/f(n)$ elements using algorithm A_1 and on exactly q elements using algorithm A_2.

6.5 Initializations

A round of $\hat{q}$ requests, $R_1, \ldots, R_{\hat{q}}$, for *initialize-min* operations on a total of q elements, where $R_i =$ *initialize-min*$(X_i, x_{i,1}, \ldots, x_{i,s_i})$, is performed as follows. The necessary memory is allocated using the algorithm of subsection 2.6. This takes $O(q/p + \log p)$ time using p processors. Then, for all requests R_i, $1 \leq i \leq \hat{q}$, divide the input lists into sublists of size $f(n)$ in $O(q/p + \log p)$ time on p processors and initialize each sublist using algorithm A_2, next construct the contents of the lists $X_1.$*reduced*$, \ldots, X_{\hat{q}}.$*reduced*, if any, in a further constant time using q processors, and finally initialize the lists $X_1.$*reduced*$, \ldots, X_{\hat{q}}.$*reduced* using algorithm A_1. Note that some $X_i.$*reduced* may be empty, and therefore are not initialized.

The total running time of the initialization procedure is $O(q/p + \log p)$ time using p processors, plus the time to perform *initialize-min* operations on $q/f(n)$ elements using algorithm A_1 and q elements using algorithm A_2.

6.6 Insertions

We consider the problem of processing a set of q *list-insert* requests. The necessary memory is allocated using the algorithm of subsection 2.6. This takes $O(q/p + \log p)$ time using p processors. Next we must group together insertions that are in the same sublist. This requires that we solve an instance of size q of the grouping problem on the data structure of algorithm A_2. Once the grouping has been done, the insertions in each sublist are carried out using algorithm A_2. Next any sublists that have become larger than $f(n)$ elements must be identified; this can be done in $O(q/p + \log p)$ time using p processors. Let q' be the total size of these sublists. Each of these sublists must be split in half. Toward this end, each sublist is mapped into an array, then divided into two sublists of size at most $f(n)$ each, and then the initialization procedure of algorithm A_2 must be run on each new sublist. The memory necessary to store these new sublists can be allocated in $O(q/p + \log p)$ time using p processors. Mapping the sublists into arrays can be done by collect operations on a total of q' elements using algorithm A_2. The sublists can be split in half in $O(q'/p)$ time using p processors. For each sublist that is split, one of the new sublists will have a global minimum equal to the global minimum of the sublist being replaced. This sublist is

already represented in the appropriate list *X.reduced*, unless *X.reduced* is empty, in which case it must also be inserted. The other new sublist must have its global minimum inserted into the appropriate list *X.reduced*. Both cases are dealt with using algorithm A_1.

The total time required for the q insertions is $O(q/p+q'/p+\log p)$ plus the time for q insertions, q' collections, and q' initializations of elements using algorithm A_2, at most q insertions using A_1, at most q initializations of one element lists using A_1, and the time to solve a grouping problem of size q on the data structure for algorithm A_2.

6.7 Deletions

We consider the problem of processing a set of q *list-delete* requests. First we must group together deletions that are in the same sublist. This requires that we solve an instance of size q of the grouping problem on the data structure of algorithm A_2. Once the grouping has been done, the q elements to be deleted are removed from their respective sublists in parallel using algorithm A_2. The sublists that become empty due to these deletions are identified and collected into an array in a further $O(q/p + \log p)$ time using p processors, and then for all such sublists G in parallel, *G.min* is removed from the appropriate list *X.reduced* using algorithm A_1. Note that *G.min* is always the last element to be deleted from a sublist G, so there is no danger that the list *X.reduced* will become incorrect. The memory that is freed by the deletions can be deallocated using the algorithm of subsection 2.6. This takes $O(q/p + \log p)$ time using p processors.

The total running time for the q deletions is $O(q/p + \log p)$ time using p processors, plus the time to perform q deletions using algorithm A_2, at most q deletions using algorithm A_1, and the time to solve a grouping problem of size q on the data structure for algorithm A_2.

6.8 Complexity

Let T_1' be the time to process a single independent operation on a single processor using algorithm A_1. Let $T_1(\hat{n}, m, r, p)$ be the time to process a collection of m cooperative operations presented over r rounds using algorithm A_1 on a p processor CREW PRAM, where $\hat{n}$ is the maximum number of elements in any one list in the data structure. Let $S_1(n, \hat{n})$ be the space required by algorithm A_1 to store a collection of lists with a maximum n total elements, and at most $\hat{n}$ elements in any one list. Define T_2', T_2, and S_2 similarly with respect to algorithm A_2. Let $G_2(\hat{n}, m, r, p)$ be the time to process r instances of the grouping problem for the data structure for A_2 using p processors, where m is the total of the sizes of the r instances, and $\hat{n}$ is the maximum number of elements stored in any one list in the data structure.

Let the maximum number of elements ever contained in a collection of lists at any one time be denoted by n.

THEOREM 4. *Given n and a function $f(n) \in o(n)$, any sequence of m cooperative dynamic restricted range minimum operations, presented over r rounds, can be processed on a CREW PRAM in*

$$\begin{aligned} O(m/p &+ r\log p + T_1(n/f(n), m/f(n), \min(r, m/f(n)), p) \\ &+ T_2(f(n), m, r, p) + G_2(f(n), m, r, p)) \end{aligned}$$

time and

$$O(S_1(n/f(n), n/f(n)) + S_2(n, f(n)) + n)$$

space using a p processors. In addition, rmin and prec queries can be processed in $O(T_1' + T_2')$ time using a single processor without performing any writes.

PROOF. The stated space complexity has already been established, as has the stated complexity for processing *rmin* and *prec* queries. It remains to prove the time complexity for the cooperative operations.

Let m_i denote the number of cooperative operations requested in round i. By definition $\sum_{i=1}^{r} m_i = n$. Define m_i' to be the number of elements that are in sublists that are split in round i. Since a list that is splitting must have at least doubled in size since the last split, the total number of elements involved in splits can be counted by charging 2 to each element inserted since the last split. This implies that $\sum_{i=1}^{r} m_i' \leq 2m$. Excluding the cost of running algorithms A_1 and A_2, processing a round of cooperative operations requires $\mathrm{O}((m_i+m_i')/p+\log p)$ time using p processors. Summing these times we obtain a time of $\mathrm{O}(m/p + r\log p)$ time. The running time required for operations by algorithm A_2 is easily seen to be $T_2(f(n), m, r, p)$. Similarly, the complexity of the grouping operations is easily seen to be $G_2(f(n), m, r, p)$. The number of elements inserted into lists maintained by algorithm A_1 is $\mathrm{O}(n/f(n))$ by construction, and hence no one list has more than $\mathrm{O}(n/f(n))$ elements in it. The number of operations performed on lists maintained by algorithm A_1 is bounded by $\mathrm{O}(m/f(n))$. This is immediate for the *collect* and *initialize* operations. For *list-insert* operations it follows from the fact that an insertion is performed using algorithm A_2 only if a sublist has reached size $2f(n)$. It follows from this that there are at most $\mathrm{O}(m/f(n))$ *list-delete* operations performed using algorithm A_2. Finally, the number of rounds that algorithm A_1 is applied in is at most $\mathrm{O}(m/f(n))$. This implies that the running time for algorithm A_1 is $T_1(n/f(n), m/f(n), \min(r, m/f(n)), p)$. The stated time complexity follows. □

THEOREM 5. *Given n, any sequence of m cooperative dynamic restricted range minimum operations, presented over r rounds, can be processed on a CREW PRAM in $O(m/p + r\log p + \min(m, r\log n))$ time and $O(n)$ space using p processors. In addition, rmin and prec queries can be processed in constant time using a single processor without performing any writes.*

PROOF. Setting both A_1 and A_2 to the algorithm of section 4, the grouping algorithm to that of subsection 4.9, and $f(n)$ to $\log^2 n$, an application of theorem 4 gives a CREW PRAM algorithm that runs in $O(m(\log\log n)^2/p + r(\log\log n)^2 + r\log p + \min(m, r\log n))$ time and $O(n\log\log n)$ space using p processors. Setting A_1 to the algorithm thus obtained, A_2 to the algorithm of section 5, the grouping algorithm to that of subsection 5.7, and $f(n)$ to $c(\log\log n)^2$ for some constant $c > 0$, an application of theorem 4 gives a CREW PRAM algorithm that runs in $O(m/p + r\log p + \min(m, r\log n))$ time and $O(n)$ space using p processors. □

We conclude the section by showing that advance knowledge of n is not necessary.

THEOREM 6. *Any sequence of m cooperative dynamic restricted range minimum operations, presented over r rounds, on a collection of lists containing at most n elements can be processed on a CREW PRAM in $O(m/p + r\log p + \min(m, r\log n))$ time and $O(n)$ space using p processors. In addition, rmin and prec queries can be processed in constant time using a single processor without performing any writes.*

PROOF. If n is not known in advance a sequence of operations can be processed by initially assuming an upper bound on n of 2, and doubling the upper bound each time the real value of n meets it. To process a sequence of operations the algorithm of theorem 5 is run using the upper bound as if it were the real value of n. Each time the upper bound is changed the entire data structure is rebuilt. The necessary information to allow rebuilding of the data structure can be recorded in a manner analogous to that used in the algorithm of section 3. This at most doubles the overall time complexity. □

7. Concluding Remarks

In the dynamic restricted range minimum problem, the insertion of a vertex requires only that it be greater than one of its neighbors, but the deletion of a vertex requires that the vertex be a global maximum in the list. It would be interesting to allow deletions under the same restrictions as insertions.

Similarly, in the dynamic lowest common ancestor problem, new roots can be inserted, but not deleted. It would be interesting to allow more general deletion operations.

Finally, all of the algorithms presented in this paper only perform well in an amortized sense. It remains open whether or not the amortization can be eliminated.

References

AHO, ALFRED V., HOPCROFT, JOHN E., AND ULLMAN, JEFFREY D. 1976. On Computing Least Common Ancestors in Trees. *SIAM J. Comput. 5*, 115–132.

Andersson, Arne. 1989. Improving Partial Rebuilding by Using Simple Balance Criteria. In *Proceedings of the Workshop on Algorithms and Data Structures*. Springer Verlag, 393–402.

Berkman, Omer and Vishkin, Uzi. 1989. Recursive *-Tree Parallel Data-Structure. In *Proceedings of the 30th Annual IEEE Symposium on the Foundations of Computer Science*, 196–202.

Cole, Richard and Vishkin, Uzi. 1988. Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM J. Comput. 17*, 1 (Feb.), 128–142.

Gabow, Harold N. 1990. Data Structures for Weighted Matching and Nearest Common Ancestors with Linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 434–443.

Gabow, Harold N., Bentley, Jon Louis, and Tarjan, Robert E. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing*, 135–143.

Gabow, Harold N. and Tarjan, Robert Endre. 1985. A Linear–Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. System Sci. 30*, 209–221.

Harel, Dov and Tarjan, Robert Endre. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput. 13*, 2, 338–355.

Higham, Lisa and Schenk, Eric. 1992. The Parallel Asynchronous Recursion Model. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, 310–316.

JáJá, J. and Simon, J. 1982. Parallel algorithms in graph theory: Planarity testing. *SIAM J. Comput. 11*, (May), 314–328.

Karp, Richard M. and Ramachandran, Vijaya. 1990. A Survey of Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science*, van Leeuwen, J., Editor. Volume A. Elsevier Science Publishers, Amsterdam, The Netherlands, and The MIT Press, Cambridge, Massachusetts, U.S.A., chapter 17.

Ladner, R. E. and Fischer, M. J. 1980. Parallel prefix computation. *J. Assoc. Comput. Mach. 27*, 831–838.

Maier, D. 1979. An efficient method for storing ancestor information in trees. *SIAM J. Comput. 8*, (Nov.), 559–618.

Moitra, A. and Iyengar, S. S. 1986. Derivation of a Parallel Algorithm for Balanced Binary Trees. *IEEE Trans. Software Engrg. SE-12*, 3 (Mar.), 442–449.

Nievergelt, J. and Reingold, E. M. 1973. Binary Search Trees of Bounded Balance. *SIAM J. Comput. 2*, 33–43.

Reif, J. H. 1985. An Optimal Parallel Algorithm for Integer Sorting. In *Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science*, 335–344.

Schenk, Eric. 1992. The Parallel Asynchronous Recursion Model. Master's thesis, Department of Computer Science, University of Calgary, Canada.

Schenk, Eric. 1994. Parallel Dynamic Lowest Common Ancestors. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*. Springer Verlag, 302–313.