Parallel Algorithms for Evaluating Sequences of Set-Manipulation Operations

MIKHAIL J. ATALLAH

Purdue University, West Lafayette, Indiana

MICHAEL T. GOODRICH

The Johns Hopkins University, Baltimore, Maryland

AND

S. RAO KOSARAJU

The Johns Hopkins University, Baltimore, Maryland

Abstract. Given an off-line sequence S of n set-manipulation operations, we investigate the parallel complexity of evaluating S (i.e., finding the response to every operation in S and returning the resulting set). We show that the problem of evaluating S is in NC for various combinations of common set-manipulation operations. Once we establish membership in NC (or, if membership in NC is obvious), we develop techniques for improving the time and/or processor complexity.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*computations on discrete structures*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Divide-and-conquer, off-line evaluation, parallel computation, parallel data structures

---

A preliminary announcement of this research appeared in *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88*. Lecture Notes in Computer Science, vol. 319. Springer-Verlag, New York, 1988, pp. 1–10.

The research of M. J. Atallah was supported by the Office of Naval Research under grants N00014-84-K-0502 and N00014-86-K-0689, the National Science Foundation under grant CCR 92-02807, and the National Library of Medicine under grant R01-LM05118. The research of M. T. Goodrich was supported by the National Science Foundation under grants CCR 93-00079 and CCR 90-03299. The research of S. R. Kosaraju was supported by the National Science Foundation under grants CCR 88-04284 and CCR 91-07293. The research of Professors Goodrich and Kosaraju was also supported by the NSF and DARPA under grant CCR 89-08092.

Authors' addresses: M. J. Atallah, Dept. of Computer Science, Purdue University, West Lafayette, IN 47907; M. T. Goodrich, Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218; S. R. Kosaraju, Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218.

1. *Introduction*

The evaluation of operation sequences is a fundamental topic in the design and analysis of algorithms. Given a sequence S of set-manipulation operations, the problem is to find the response to every operation in S and return the set one gets after evaluating S, so that the answers are the same as if the operations in S were performed in a sequential fashion. There are a host of problems that are either instances of an evaluation problem or can be solved by a reduction to an evaluation problem. For example, sorting a set $S = \{x_1, x_2, \dots, x_n\}$ can easily be reduced to the problem of evaluating the sequence $I(x_1)I(x_2)\cdots I(x_n)E\ E\ \cdots\ E$, where $I(x)$ stands for "insert x," E stands for "*ExtractMin*," and there are n E's. The answers to all the E operations immediately give a sorting of the items in S (this is, in fact, the idea in "heap sort" [Aho et al. 1974]).

The sequence evaluation problem is well-studied in the sequential setting (e.g., Aho et al. [1974], Edelsbrunner and Overmars [1985], and Gabow and Tarjan [1983]), but surprisingly little is known about its parallel complexity. Our motivation, then, comes from a desire to begin a systematic treatment of this important area from a parallel perspective. In addition, because of the foundational aspect of off-line evaluation problems, we are also interested in these problems for their possible applications. We already know of applications to such areas as processor scheduling, computational geometry, and computational graph theory, for example (we discuss some of these below).

As an example illustrating the difficulty of the parallel version of off-line evaluation problems, consider the following sequence of set-manipulation operations:

$$S = I(5)I(8)ED(5)I(7)I(9)ED(8)EE,$$

where $I(x)$ is an abbreviation for *Insert*(x) and inserts x in the set, $D(x)$ is an abbreviation for *Delete*(x) and deletes x from the set, and E stands for *ExtractMin* and simultaneously removes and returns the smallest element in the set (if the set is empty then it returns a "set empty" response). The set is initially empty, and the operations are applied to it in the same order in which they appear in S. An attempt to delete an element not in the set has no effect and returns an "element not in set" response; otherwise, it returns an "element deleted" response. The response to an $I(x)$ operation is always "element inserted" and its effect is to add x to the set (if x is already there then another copy of it is added). The problem is to compute, in parallel, the responses to all the operations in S. In the example given above, the sequence of responses is: 5 inserted, 8 inserted, 5, 5 not in set, 7 inserted, 9 inserted, 7, 8 deleted, 9, set empty. It is far from clear that the problem of evaluating such a sequence is in the complexity class *NC*, that is, that it can be evaluated in $O(\log^k n)$ time using a polynomial number of processors, for some constant k [Dymon and Cook 1980; Ruzzo 1981]. The difficulty arises from the fact that one has no a priori knowledge of the behavior of the E and the $D(x)$ operations. Some of them may not remove anything (e.g., an E applied to an empty set, or a $D(x)$ applied to a set in which there is no x), while others are successful, and determining whether or not a particular operation O_t is successful depends on knowing which operations before O_t in S are successful. We show that the evaluation of a sequence of $I(x)$, $D(x)$, and E operations is in fact in *NC* (Section 3).

We note in passing that the assumption regarding the insertion of an existing element is made without loss of generality. For example, if one wishes to define insertion so that an attempt to insert an element x already in the set is ignored, then one can easily convert a sequence S, where redundant insertions are ignored, to a sequence S', where insertions are handled as above, as follows: from S, create S' by replacing every $I(x)$ by a $D(x)I(x)$ (each such $D(x)$ can be labeled *extraneous* to distinguish it from delete operations in S). Now consider an evaluation of S'; It never attempts to insert an element that is already in the set (because of the way S' was built). Furthermore, the response in S' to an extraneous $D(x)$ tells us whether the $I(x)$ that follows it would be, in S, an attempt to insert an element already present: This is the case if and only if the response to the extraneous $D(x)$ is "element deleted," rather than "element not in set."

In general, this paper studies the following evaluation problem: One is given a sequence $S = O_1 O_2 \cdots O_n$ of operations taken from some instruction set and asked to produce the answer each O_t would give if S were evaluated sequentially in an on-line fashion. Since the answer for each operation in S is defined by a hypothetical sequential evaluation of S, we define an operation's position in S to be its *time of evaluation*, that is, O_t's time of evaluation is t. We study this problem for various instruction sets, deriving one of two types of results for each:

(i) Given a sequence S, containing various kinds of operations, we show that the problem of evaluating S is in the class *NC*.
(ii) Once membership in *NC* is established, we develop techniques for improving the time and/or processor complexity.

Our primary goal is to minimize the time complexity of evaluating S and our secondary goal is to minimize the number of processors used. The computational model we use is the CREW PRAM model, unless otherwise specified. Recall that this is the shared-memory model where the processors operate synchronously and can concurrently read any memory cell, but concurrent writes are not allowed. Some of our results are for the weaker EREW PRAM, in which no concurrent memory accesses are allowed. We outline the specific problems we address in this framework below, and give for each the time and processor bounds we achieved.

(1) *The off-line binary search tree problem.* In this problem, the operations that appear in S are *Insert*(x), *Delete*(x), and "tree-search" queries. Intuitively, a tree-search query is one that could be performed efficiently if the set were stored in a balanced binary search tree (e.g., finding the minimum, selecting the kth smallest element, range counting). We make this notion precise in Section 2, where we show how to evaluate such a sequence in $O(\log n)$ time using $O(n)$ processors. Our solution is fairly simple, and will be used as a subroutine in the (more difficult) solutions of later sections. The solution is based on the use of a parallel data structure that we call the *array-of-trees*, which has already found applications to a number of problems in parallel computational geometry [Goodrich 1991; Goodrich et al. 1990]. We know of no previous parallel algorithms for this problem; the only related work is a method

by Paul et al. [1983] for maintaining a binary tree in parallel through batch insertions and deletions (where all the insertions or all the deletions come at the same time).

(2) *The off-line competitive deletes problem.* In this problem, the operations in S come from the set $\{Insert(x), Delete(x), ExtractMin\}$. We show that this problem is in NC^2 and has an NC solution with a time-processor product of $O(n \log^2 n)$. Since there are two data-dependent ways that elements can be deleted in this problem (as discussed in the example above), showing that this problem is in NC, let alone that it has an NC solution with an efficient time-processor product, is quite interesting. (We called it the *competitive deletes* problem because the two mechanisms for deletion, the E and $D(x)$ operations, are "competing" with each other.)

(3) *The off-line mergeable heaps problem.* In this problem, the operations in S can take both set names and elements as arguments. In particular, the operations in S come from the set $\{Insert(x, A), Delete(x), Min(A), Union(A, B), Find(x)\}$, where A and B are set names. We show that any such S can be evaluated in $O(\log n)$ time using $O(n)$ processors. Our method is based on using the array-of-trees data structure in conjunction with an application of the cascade merging technique [Atallah et al. 1989; Cole 1988] to tree-contraction [Miller and Reif 1985].

(4) *The off-line priority queue problem.* In this problem, the operations that appear in S come from the set $\{Insert(x), ExtractMin\}$. We derive an algorithm that runs in $O(\log n)$ time using $O(n)$ processors, which is optimal. This improves an $O(\log^2 n)$ time, n processor solution that is implicitly present in Dekel and Sahni's work on parallel scheduling algorithms [Dekel and Sahni 1983]. Our result also improves the time complexity for solving the scheduling problem studied by Dekel and Sahni. (Subsequent to our initial announcement of this result, we have learned that Rodger has independently discovered a similar improvement to this scheduling problem [1989].)

(5) *The off-line barrier-extractmin problem.* In this problem, the operations in S come from the set $\{Insert(x), ExtractMin(y)\}$, where the $ExtractMin(y)$ operation (a generalization of $ExtractMin$) returns and simultaneously removes the smallest element greater than or equal to y. That is, it is a *barrier-extract-min* operation. We show that this evaluation problem is in NC^2 in the general case, and in NC^1 for the case when the $ExtractMin(y)$'s have non-decreasing arguments. This special case is motivated by an application to computing a maximum matching in a convex bipartite graph, which, in turn, has applications to processor scheduling [Dekel and Sahni 1984]. Our results imply that this matching problem is in NC^1, improving the previous NC^2 solution by Dekel and Sahni [1984]. We believe that the $ExtractMin(y)$ operation will be helpful in solving many other "lexicographic" problems, as well.

The details for each result are given in what follows, one per section. We conclude with some final remarks in Section 7.

In what follows, if A is a set and B a sequence of set manipulation operations, then AB denotes applying the sequence B to a set whose initial value is A (we use $\emptyset B$ to denote the case when the initial set is empty). In addition to the responses to the operations in B, an evaluation of AB also returns the set "left over" after B is evaluated. In this notation, we are interested in evaluating $\emptyset S$ for various types of S's.

2. *The Off-Line Binary Search Tree Problem*

This section gives a simple solution to a problem that is needed as a subroutine in later sections of this paper: that of evaluating a sequence of $I(x)$'s, $D(x)$'s, and "tree-search" queries. By the name "tree-search" query we mean any query that could be performed in $O(\log n)$ time if the elements in the set were stored in a balanced binary search tree where each node v of this tree could store $O(1)$ labels, each label being the value of some associative operation computed over all the elements stored in descendents of v (note that the usual search key information stored in the nodes of binary search trees satisfies this condition). Examples of such tree-search queries include finding the kth smallest element, and computing the number of elements in a certain range. For the sake of definiteness, we assume in what follows that the label *label*(v) at a node v includes the number of elements in its subtree (the method is easily seen to work for other such labels). Thus, a query Q is any query that could be done sequentially in $O(\log n)$ time if the elements of the set were available sorted at the leaves of such a balanced binary tree T is which each internal node v stores *label*(v). Thus, query Q could be "find the kth smallest element in the set".

2.1. THE ARRAY-OF-TREES DATA STRUCTURE. Let m be the number of set-modifying operations in S (the $I(x)$'s and $D(x)$'s), $m \leq n$. The subsequence S' of such set-modifying operations can easily be obtained from the input sequence S by a parallel prefix computation,[1] and we assume that this has already been done.

Our method is based on the idea of storing all of the m relevant "snapshots" of a sequential data structure that evaluates the sequence on-line with $O(\log m)$ time per operation. However, storing m copies of this sequential data structure would be prohibitively expensive, so we "compress" the representation of these logical m data structures into a single data structure that is suitable for building and processing in $O(\log m)$ time using $O(m)$ processors. The method for constructing this representation makes use of the cascading divide-and-conquer technique [Atallah et al. 1989].

Let A_t denote the set of items that are present at "time" t, that is, the set that would be formed by performing all the operations of S' up to and including the operation in position t of S', assuming that the initial set is $\emptyset$. The array-of-trees data structure allows one processor to perform a query Q in any such A_t in $O(\log m)$ time. In fact, this structure can be viewed as an array of m trees where the tth tree stores the elements of A_t (hence, the name "array-of-trees"). In this section we show that this structure can be built in $O(\log m)$ time and $O(m \log m)$ space using $O(m)$ processors in the CREW PRAM model.

Because of the absence of *ExtractMin* operations, we can, without loss of generality, assume that all the $D(x)$ operations are non-redundant (i.e., there

[1]Recall that in a parallel prefix computation one has a sequence $(a_1, a_2, \ldots, a_n)$ and one wishes to compute all partial sums $s_k = \sum_{i=1}^{k} a_i$, which can be done in $O(\log n)$ time using $O(n/\log n)$ processors [Kruskal et al. 1985; Ladner and Fischer 1980].

are no attempts at deleting an element not in the set). The redundant $D(x)$ operations can be removed as a pre-processing step by sorting the pairs (x, t), where x is an argument to an *Insert*(x) or *Delete*(x) operation and t is the position of that operation in S'. Such a preprocessing step can be implemented in $O(\log m)$ time using $O(m)$ processors [Cole 1988]. Also, without loss of generality, we assume that every element x that is the argument of an *Insert*(x) is unique. If not, then we can think of the element as a pair (x, t), where t is the position of the insert operation in S'. We can then use some uniform way of determining which copy of an element x is removed by a *Delete*(x) operation, for example, that it removes the most recently inserted copy.

Let O_t denote the tth operation in S', and let x_t denote the argument of O_t. Recall that we think of O_t as the operation that is to be performed at *time t* (in a hypothetical sequential execution of S'). Let X be a list of all the distinct x_t values, in sorted order. The "skeleton" of the array-of-trees in a complete binary tree T with $|X| \leq m$ leaves, such that the elements of X are associated with the leaves of T in left-to-right order. By an abuse of notation, we use the same symbol to denote both a leaf in T and the value that is associated with that leaf. In each leaf node x of T, we store $S'(x)$, the subsequence of S' consisting of all operations that have x as their argument. Note that X, T, and all the $S'(x)$'s can be constructed in $O(\log m)$ time with $O(m)$ processors by using parallel sorting [Cole 1988].

The *array-of-trees* (*AOT*) structure consists of m trees that share nodes, the tth tree depicting the (hypothetical) sequential binary tree just after operation O_t is applied to it. A node of the (skeleton tree) T is called a *supernode* and contains a number of *mininodes*, that are nodes of the m individual trees it is supposed to represent. The root supernode of the array of trees contains a list of m mininodes such that the tth one is the root of the tth tree. If one starts at the root of the tth tree, one can traverse the tth tree by following left and right child pointers stored at each of the mininodes of the AOT. Because of mininode-sharing (the details of which are given later), there are only $O(m \log m)$ mininodes, organized as $\log m$ levels such that the ith level contains m mininodes grouped into $\leq 2^i$ nodes (the root is at level 0). The supernodes at a certain level need not contain the same number of mininodes, but their total at that level is m mininodes. Each mininode consists of a 4-tuple (t, l, r, X), whose significance is as follows. The first component of a mininode's 4-tuple, t, indicates that this mininode's subtree describes the corresponding subtree of A_t, the (hypothetical) sequential binary tree just after operation O_t is applied to it. The second (respectively, third) component of a mininode's 4-tuple, l (respectively, r) is a pointer to the mininode that is its left (respectively, right) child. The fourth component, X, is the *label* of that mininode in A_t (in this case, the number of leaves in its subtree in A_t). The above was an "overview," and we now give a precise description of the AOT. We do so in a "bottom up" fashion, starting from the m leaf supernodes (i.e., at level $\log|T|$).

For each leaf node x of T, we construct a leaf supernode $B(x)$ of the AOT that consists of the list (also called $B(x)$) obtained from $S'(x)$ by replacing each O_t in $S'(x)$ with a record $(t, \textbf{nil}, \textbf{nil}, 0)$ if $O_t = D(x)$ or with $(t, \textbf{nil}, \textbf{nil}, 1)$ if $O_t = I(x)$. We also add the "dummy" mininode $(0, \textbf{nil}, \textbf{nil}, 0)$ to the beginning of the list $B(x)$. Thus, $B(x)$ represents the history of all sets defined by restricting one's attention to the operations in $S'(x)$. That is, if we let

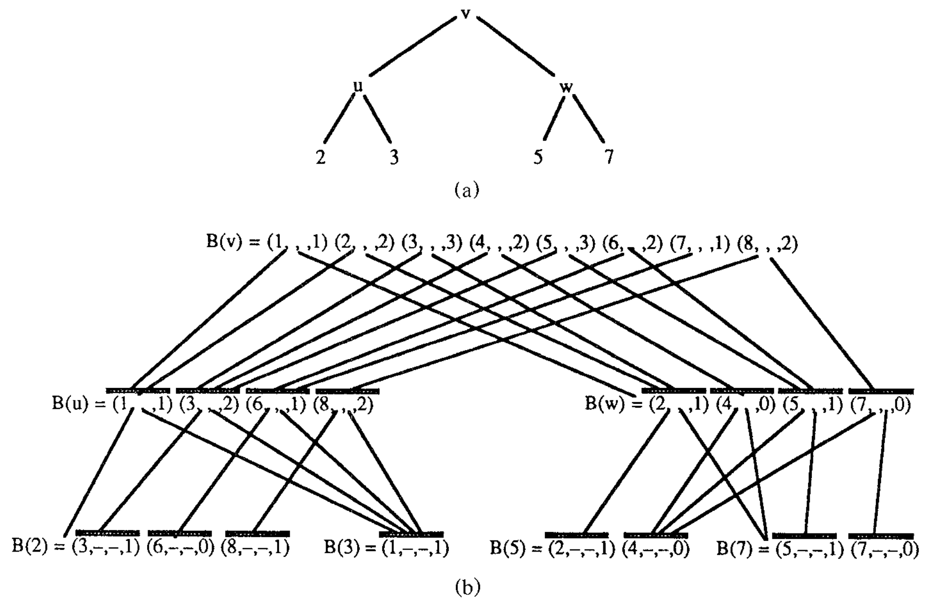
v
u
w
2
3
5
7
(a)
B(v) = (1, , ,1) (2, , ,2) (3, , ,3) (4, , ,2) (5, , ,3) (6, , ,2) (7, , ,1) (8, , ,2)
B(u) = (1, , ,1) (3, , ,2) (6, , ,1) (8, , ,2)
B(w) = (2, , ,1) (4, , ,0) (5, , ,1) (7, , ,0)
B(2) = (3,–,–,1) (6,–,–,0) (8,–,–,1)
B(3) = (1,–,–,1)
B(5) = (2,–,–,1) (4,–,–,0)
B(7) = (5,–,–,1) (7,–,–,0)
(b)

FIG. 1. *AOT* for $S' = I(3)I(5)I(2)D(5)I(7)D(2)D(7)I(2)$. (a) T. (b) *AOT*.

$(t_0, t_1, \ldots, t_{|B(x)|})$ denote the list of t-values in $B(x)$, then each mininode $(t_i, \mathbf{nil}, \mathbf{nil}, *)$ in $B(x)$ can be alternatively thought of as representing the root of a (trivial) binary tree storing the "projections" of the sets $A_{t_i}, A_{t_i+1}, \ldots, A_{t_{i+1}-1}$ on element x (the projection of a set on an element is that element if the set contains it, empty otherwise). It is because the projections of $A_{t_i}, A_{t_i+1}, \ldots, A_{t_{i+1}-1}$ on element x are identical that we achieve a savings in space: we only store one copy of this projection, namely, the mininode $(t_i, \mathbf{nil}, \mathbf{nil}, *)$.

Now, for each internal node v of T, we construct an internal supernode $B(v)$ of the *AOT* by merging $B(u)$ and $B(w)$ as sorted lists by t-values (removing the duplicate for $t_0 = 0$), where u and w are the children of v in T. Each element of $B(v)$ is a mininode $(t, l, r, L_{v,t})$ where t is the first coordinate of a mininode in $B(u) \cup B(w)$ (as determined by the merge), and l (respectively, r) is a pointer to the mininode in $B(u)$ (respectively, $B(w)$) whose first coordinate t_l (respectively, t_r) is the largest such coordinate less than or equal to t. $L_{v,t}$ is the label of v in A_t, in this case simply the sum of L_{u,t_l} and L_{w,t_r}. By a simple induction, it is easy to see that, if $(t_1, t_2, \ldots, t_{|B(v)|})$ denotes the list of t-values in $B(v)$, then each mininode (t_i, l, r, L_{v,t_i}) in $B(v)$ represents the root of a binary tree representing the (common) subset of the sets $A_{t_i}, A_{t_i+1}, \ldots, A_{t_{i+1}-1}$, as it relates to the elements which are descendents of v. Thus, each mininode $(t, l, r, *)$ of $B(root(T))$ will represent the root of a binary tree storing the entire list A_t. (See Figure 1 for an example, where we avoided showing the dummy mininode at the beginning of each supernode.)

Since the list of times for the mininodes in $B(v)$ is exactly the merged union of the lists of mininodes stored in v's two supernode children, we can apply the cascading divide-and-conquer technique [Atallah et al. 1989; Cole 1988] to construct the AOT data structure in $O(\log m)$ time and $O(m \log m)$ space using $O(m)$ processors in the CREW PRAM model. The method also produces the labels (such as the $L_{v,t}$ values) for each mininode in $B(v)$ within these same bounds.

2.2. USING THE ARRAY-OF-TREES FOR THE OFF-LINE BINARY SEARCH TREE PROBLEM. Once we have constructed the AOT data structure for S', computing the responses to all the tree-search queries of S is quite simple. From the parallel prefix computation that obtained S' from S, we know for each query operation Q in S the nearest set-modifying operation before it in S, say it is O_t in S' (i.e., the tth operation in S'). This tells us which "tree" we must search in order to process query Q. We therefore assign a single processor to each such query operation Q, and that processor then performs the query operation in the appropriate tree A_t just as it would in the sequential algorithm.

Let us make this more concrete with an example. Suppose we want to evaluate a sequence of *Insert*(x), *Delete*(x), and *Select*(k) operations, where *Select*(k) reports the kth smallest element in the set at the time. In this case, one constructs the AOT so that each internal node stores the number of descendent leaves of that node (in addition to the t, l, and r fields). One then can answer a particular *Select*(k) operations at, say, time t by searching in the "tree" for time t using the obvious searching strategy. This takes $O(\log m)$ time for each operation. Thus, the entire sequence can be evaluated in $O(\log m)$ time using $O(n)$ processors. See Goodrich et al. [1990] for applications of the AOT data structure to some important computational geometry problems.

Incidentally, the evaluation of a sequence of $I(x)$, $D(x)$, and *Query* operations can be performed in $O(\log n)$ time using only $O(n)$ space if all the queries are themselves the values of associative operations, for example, *Min*, *Sum*, etc. [Atallah et al. 1989]. (See Atallah et al. [1989] for applications of such sequences to several computational geometry problems.)

THEOREM 2.2.1. *Given a sequence S of n Insert(x), Delete(x), and tree-query operations, one can evaluate $\emptyset S$ in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.*

In the next section, we address the (considerably harder) case when there are two different kinds of operations that delete elements, namely, *Delete*(x) and *ExtractMin*.

3. *The Off-Line Competitive Deletes Problem*

In this section, we show that the problem solving $\emptyset S$, where the operations in S come from the set $\{I(x), D(x), E\}$, is in NC^2 (using a quadratic number of processors). (Recall that E is a shorthand for *ExtraxtMin*.) We also show how to refine our approach to achieve $O(\log^2 n \log\log n)$ time using only $O(n/\log\log n)$ processors. The difficulty in designing an NC algorithm for this problem comes from the fact that a $D(x)$ operation can cause a subsequent E operation to return the "set empty response," and an E operation can cause a subsequent $D(x)$ operation to return the "element not in set" response. This complicates the parallel evaluation of $\emptyset S$, since the competing $D(x)$ and E operations may be far apart in S.

Suppose we are given such a sequence S. Our method for evaluating $\emptyset S$ is as follows. Let S_1 (respectively, S_2) be the sequence consisting of the first (respectively, last) $n/2$ operations in S. Recursively, solve $\emptyset S_1$ and $\emptyset S_2$ in parallel. The recursive call for $\emptyset S_1$ returns (i) the correct responses for the operations in it (i.e., the same as in $\emptyset S$), and (ii) the set just after $\emptyset S_1$ terminates (let L_1 denote this set). The recursive call for $\emptyset S_2$ returns responses and a final set that may differ from the correct ones, because we applied S_2 to $\emptyset$ rather than to L_1. The main problem that we now face is how to incorporate the effect of L_1 into the solution returned by the recursive call for $\emptyset S_2$. We show how to compute $L_1 S_2$ in the following subsection. The crucial insight is contained in Lemma 3.3.

3.1 AN NC^2 SOLUTION. Our method for incorporating the effect of L_1 on S_2 involves a number of restructurings of S_2: roughly speaking, we remove some operations from S_2 and permute the remaining ones so that the restructured list has a special "suffix property" relative to the effects of L_1. We show that a solution to the restructured problem can be easily converted to a solution to the original problem.

Notation. If R is a subsequence of S, then $S - R$ denotes the sequence obtained by removing every operation in R from S.

Throughout this section, our algorithms adopt the convention that sets are actually multisets (i.e., multiple copies of an element are allowed), so that whenever we say "element x" we are actually referring to a particular copy of x. As mentioned in the introduction, it is straightforward to modify our results for the case when it is forbidden to have multiple copies of an element.

By convention, a $D(x)$ executed when there are many copies of x in the set removes the copy that was inserted latest. Similarly, an E executed when there are many copies of the smallest element in the set removes the copy that was inserted latest. These conventions cause no loss of generality, because they do not change any response. However, they do simplify our correctness proofs.

Let L_2 be the set resulting from $\emptyset S_2$ (i.e., the set after $\emptyset S_2$ terminates). Consider an $I(x)$ for which x is not removed by any E in $\emptyset S_2$, that is, it either ends up in L_2 or gets removed by a $D(x)$ (in the latter case we say that the $D(x)$ *corresponds* to $I(x)$). Let S' be the sequence obtained from S_2 by removing every such $I(x)$ and its corresponding $D(x)$ (if any). In other words the only $I(x)$ operations in S' are those whose x was removed by an E in $\emptyset S_2$, and the only $D(x)$ operations in S' are those whose response in $\emptyset S_2$ was "x not in set." It is easy to see that the response to any operation in S' is the same in $\emptyset S'$ as in $\emptyset S_2$. However, the following also holds:

LEMMA 3.1.1. *The responses to the operations in S' are the same in $L_1 S'$ as in $L_1 S_2$. The set resulting from $L_1 S_2$ equals L_2 plus the set resulting from $L_1 S'$.*

PROOF. The lemma would immediately follow if we can prove that, for any $I(x)$ that is in $S_2 - S'$, the following properties (i) and (ii) hold:

(i) if x ends up in L_2 after $\emptyset S_2$, then it also ends up in the set resulting from $L_1 S_2$.
(ii) if x is, in $\emptyset S_2$, removed by a $D(x)$, then it is removed by the same $D(x)$ in $L_1 S_2$.

Properties (i) and (ii) together would imply that the operations in $S_2 - S'$ have, in $L_1 S_2$, no effect on any operation in S' and can therefore be ignored, their only effect being the addition of L_2 to the resulting set (as returned by

$\emptyset S_2$). We prove (i) and (ii) by contradiction: let $I(x)$ be the rightmost insertion in $S_2 - S'$ that violates (i) or (ii).

Case 1. $I(x)$ violates property (i), that is, x ends up in L_2 after $\emptyset S_2$ but is removed by some operation O in L_1S_2. If $O = D(x)$, then, since x ends up in L_2 after $\emptyset S_2$, O does not remove x in S_2 and hence must have removed another copy of x (call it x', $x' = x$). By our convention that the latest copy is removed by a deletion, $I(x')$ must have occurred after $I(x)$. Since $I(x')$ violates property (ii), this contradicts our choice of $I(x)$ as the rightmost violation of (i) or (ii). If $O = E$, then, since O did not remove x in $\emptyset S_2$, the response to O in S_2 must have been better (either smaller than x, or equal to it but inserted later). But it is a contradiction for x, the response of O in L_1S_2, to be worse than the response of O in $\emptyset S_2$ (because having L_1 rather than $\emptyset$ as the initial set can only make the response of any E better).

Case 2. $I(x)$ violates property (ii), that is, x is removed by $D(x)$ in $\emptyset S_2$, but is not removed by the same $D(x)$ in L_1S_2. Suppose x is removed in L_1S_2 by O. If O is a deletion, then, since x is removed by $D(x)$ rather than by O in $\emptyset S_2$, O must have removed in $\emptyset S_2$ another copy of x (call it x', $x' = x$). By our convention that the latest copy is removed by a deletion, $I(x')$ must have occurred after $I(x)$. Since $I(x')$ violates property (ii), this contradicts our choice of $I(x)$ as the rightmost violation of (i) or (ii). If $O = E$, then, since O did not remove x in $\emptyset S_2$, the response to O in $\emptyset S_2$ must have been better (either smaller than x, or equal to it but inserted later). But this is a contradiction for x, the response of O in L_1S_2, to be worst than the response of O in $\emptyset S_2$. □

Lemma 3.1.1 has reduced the problem of solving L_1S_2 to that of solving L_1S'. The next lemma will further reduce the problem to one in which a crucial *suffix property* holds, as is later established in Lemma 3.1.3.

LEMMA 3.1.2. *Let $\hat{S}$ be obtained from S' by moving every $I(x)$ to just before the E whose response it was in $\emptyset S_2$ (such an E must exist by definition of S'). Then the responses to the operations in S' are the same in $L_1\hat{S}$ as in L_1S_2. The set resulting from L_1S_2 equals L_2 plus the set resulting from $L_1\hat{S}$.*

PROOF. Because of Lemma 3.1.1, it suffices to prove that the responses to the operations in S' are the same in $L_1\hat{S}$ as in L_1S' and that the set resulting from $L_1\hat{S}$ is the same as the set resulting from L_1S'. Therefore, it suffices to show that for no $I(x) \in S'$ can x be removed, in L_1S', any earlier than by the E (call it E_1) that removed x in $\emptyset S_2$ (this would establish that moving that $I(x)$ to just before E_1 does not change anything). Suppose, to the contrary, that such an x is removed in L_1S' by some operation O that occurs before E_1. That operation O cannot be a $D(x)$ because otherwise that same $D(x)$ (and not E_1) would have removed x in $\emptyset S_2$ (since that $D(x)$ is in S', it had an "x not in set" response in $\emptyset S_2$). Therefore, O is an E (say, E_2). Now, the response of E_2 in $\emptyset S_2$ must have been some y that is better than x (because x ended up being removed by E_1). This means that x, the response to E_2 in L_1S', is worse than its response in $\emptyset S_2$. Since the response to E_2 in $\emptyset S_2$ is the same as its response in $\emptyset S'$, it follows that the response to E_2 in L_1S' is worse than its response in $\emptyset S'$. It is a contradiction for the response to an E to be worse in L_1S' than it is in $\emptyset S'$. □

Since we already know the responses to $\varnothing S_2$ (they were returned by one of the two parallel recursive calls), a simple parallel prefix computation easily identifies the set S' (and hence $S_2 - S'$ and $\hat{S}$), in $O(\log n)$ time and with $O(n/\log n)$ processors. The responses in L_1S_2 to the operations in $S_2 - S'$ are now trivially known: the response to an $I(x)$ is "x inserted" by the definition of $I(x)$, and the response to a $D(x)$ is "x deleted" by the definition of S'. The main problem we face is obtaining the responses in L_1S_2 to the operations in S', and obtaining the final set resulting from L_1S_2. Lemma 3.1.2 has reduced this problem to that of solving $L_1\hat{S}$, so we now focus on obtaining the responses and final set for $L_1\hat{S}$. The rest of this subsection shows that they can be obtained in $O(\log n)$ time and with $O(n^2)$ processors, thus implying for the overall problem an $O(\log^2 n)$ time and $O(n^2)$ processor bounds.

Let $\hat{S} = O_1O_2 \cdots O_m$, $m \le n/2$. For every j, $1 \le j \le m$, let $S(j)$ be the sequence of operations obtained from $O_1 \cdots O_j$ by removing the E's from it. Note that $S(j)$ contains only two kinds of operations: (i) $I(x)$ for which x was a response to an E in $\varnothing S_2$, and (ii) $D(x)$ whose response was "x not in set" in $\varnothing S_2$. Let $L(j)$ denote the set resulting from $L_1S(j)$. Let $L(0)$ denote L_1. Recall that, by convention, element x is better than element y if and only if either (i) $x < y$, or (ii) $x = y$ and x was inserted later than y.

LEMMA 3.1.3 (THE SUFFIX PROPERTY LEMMA). *For every j such that O_j is a $D(x)$ or an E, $1 \le j \le m$, there is an integer $f(j)$, $0 \le f(j) \le |L(j)|$, such that the set resulting from $L_1O_1 \cdots O_j$ consists of the $f(j)$ worst (i.e., largest) elements in $L(j)$.*

PROOF. It suffices to prove that the $D(x)$'s and E's (in $L_1O_1 \cdots O_j$) remove the b best elements in $L(j)$, for some integer b (this would establish the lemma, with $f(j) = |L(j)| - b$). The proof is by contradiction: Suppose to the contrary that some O_i, $i < j$, removes an element x of $L(j)$ and that some element y of $L(j)$, where y is better than x, is not removed by any operation (in $L_1O_1 \cdots O_j$). We distinguish two cases.

Case 1. O_i is an E (call it E_1).
Since y is better than x, y could not have been present when E_1 removed x, and therefore y was inserted by an $I(y)$ that comes after E_1 and before O_j. Such an $I(y)$ is (by definition) in $\hat{S}$, and therefore (by the definition of $\hat{S}$) it is immediately followed in $\hat{S}$ by an E (call it E_2) that is after E_1 and not after O_j (possibly $E_2 = O_j$, since $I(y) = O_{j-1}$ is possible). By hypothesis, y is not removed in $L_1O_1 \cdots O_j$ and hence E_2 must have removed a z that is better than y. Since z is better than x, z could not have been present when E_1 removed x and therefore z was inserted by an $I(z)$ that comes in between E_1 and E_2. Such an $I(z)$ is (by definition) in $\hat{S}$, and therefore (by the definition of $\hat{S}$) $I(z)$ is immediately followed in $\hat{S}$ by an E (call it E_3) that is in between E_1 and E_2 ($E_3 \ne E_2$ because it is $I(y)$ and not $I(z)$ that occurs just before E_2 in $\hat{S}$). Now, repeat the argument with E_3 playing the role of E_2, as follows. E_3 did not remove z in $L_1O_1 \cdots O_j$ and hence must have removed a w that is better than z. This w could not have been present when E_1 removed x and hence it must have been inserted by an $I(w)$ that comes in between E_1 and E_3, and is followed by an E_4 that is in between E_1 and E_3. Repeat the argument with E_4 playing the role of E_3, resulting in an E_5 that is in between E_1 and E_4, etc.

Eventually, after (say) q iterations of this argument, a contradiction is reached (when there is no E in between E_1 and E_q). Thus, O_i cannot be an E.

Case 2. O_i is a $D(x)$.

Then clearly $y < x$, since if $y = x$, then O_i would have removed y rather than x. In $L_1 S(j)$, x ended up in $L(j)$ and hence was not removed by O_i, and therefore O_i removed another, better (i.e., later) copy x_1 $(x_1 = x)$. The fact that O_i removes x rather than x_1 in $L_1 O_1 \cdots O_j$ means that x_1 was removed earlier by some operation O_t, $t < i$. If O_t is an E, then a contradiction is obtained as in Case 1 (with O_t and x_1 playing the roles of E_1 and x, respectively). So suppose O_t is a $D(x)$. In $L_1 S(j)$, x_1 was removed by O_i rather than by O_t, and therefore O_t removed another, better copy x_2 $(x_2 = x)$. The next paragraph iterates the argument of this paragraph one more time.

That O_t removes x_1 rather than x_2 is $L_1 O_1 \cdots O_j$ means that x_2 was removed earlier by some operation O_u, $u < t$. If O_u is an E, then a contradiction is obtained as in Case 1 (with O_u and x_2 playing the roles of E_1 and x, respectively). So suppose O_u is a $D(x)$. In $L_1 S(j)$, x_2 was removed by O_t rather than by O_u, and therefore O_u removed another, better copy x_3 $(x_3 = x)$. Iterating the argument eventually leads to a contradiction (when after q iterations we get to x_q, the earliest copy of x). Thus, O_i cannot be a $D(x)$ either. This completes the proof of the lemma. □

Thus, if O_j is an E or a $D(x)$, then the set resulting from $L_1 O_1 \cdots O_j$ is a suffix of $L(j)$. The size of this suffix is $f(j)$. It is not hard to come up with examples showing that the suffix property does not hold for an O_j which is an $I(x)$; by convention, if O_j is an $I(x)$, then $f(j)$ is undefined. The "suffix" property is the main reason why we can solve the problem in *NC*. We have yet to show how to exploit this property, however. For now, we note that, if we knew all the $f(j)$ values, then we would be essentially done (we omit the trivial details of the proof that knowing the $f(j)$'s implies knowing the responses to $L_1 \hat{S}$).

We now turn our attention to showing that the $f(j)$'s can, in fact, be computed in $O(\log n)$ time with $O(n^2)$ processors. Using the array-of-trees technique described in the previous section, we can compute an implicit representation of each of $L(0), L(1), \ldots, L(m)$, stored in a binary tree. Once we have such a description of the $L(i)$'s, a single processor can determine in $O(\log n)$ time whether a certain x is in $L(i)$ or not. Now, for each $O_i = D(x)$, we check whether x is in $L(i-1)$: If not, then such a $D(x)$ has no effect and can therefore be ignored; we henceforth assume that all such $D(x)$'s have been purged from $\hat{S}$ and the $L(i)$'s recomputed accordingly (i.e., from now on for each $D(x)$ in $\hat{S}$, we know that $x \in L(i)$).

Let $L(i, k)$, $1 \le i \le m$, $1 \le k \le |L(i)|$, denote the set consisting of the worst k elements of $L(i)$. Note that $L(0, |L_1|) = L_1$. We say that an O_i is *relevant* if it is an E or a $D(x)$ (i.e., not an $I(x)$). Let O_i be relevant, and let $O_{s(i)}$ be the next relevant operation in $\hat{S}$; in fact we have either (i) $s(i) = i + 1$ (if O_{i+1} is not an $I(x)$), or (ii) $s(i) = i + 2$ (if O_{i+1} is an $I(x)$), because every $I(x)$ is followed by an E in $\hat{S}$. If $L(i, k) O_{i+1} \cdots O_{s(i)}$ results in $L(s(i), p)$, then we say that $L(s(i), p)$ is the *successor* of $L(i, k)$. The lemma below shows that if $L(s(i), p)$ is the successor of $L(i, k)$, then $p \in \{k, k-1\}$. An $L(i, k)$ has no successor if O_i is the last relevant operation in $\hat{S}$ (i.e., if $i = m$); otherwise, it has exactly one successor.

Definition 3.1.4. For each $O_{t+1} = D(x)$, let n_t be the number of elements in $L(i)$ that are $\geq x$.

All the n_i's can easily be computed in $O(\log n)$ time, since we have the $L(i)$'s.

In the lemma below, the reader should keep in mind that, by the definition of $\hat{S}$, every $I(x)$ in it is immediately followed by an E, and every E is immediately preceded by an $I(x)$.

LEMMA 3.1.5. *For a relevant O_t, the successor of $L(i,k)$ is obtained as follows*:

Case 1. *O_{t+1} is a $D(x)$ operation. The successor of $L(i,k)$ is $L(i+1,k)$ if x is not in $L(i,k)$ (i.e., if $k < n_t$), and is $L(i+1,k-1)$, otherwise.*

Case 2. *O_{t+1} is an $I(x)$ operation and O_{t+2} is an E operation. The successor of $L(i,k)$ is $L(i+2,k)$.*

PROOF. Let us consider each case in turn.

Case 1. O_{t+1} is a $D(x)$ operation. In this case, if x is in $L(i,k)$, then there is one less element in the set resulting from $L(i,k)O_{t+1}$ than in $L(i,k)$. If x is not in $L(i,k)$, then this $D(x)$ operation has no effect.

Case 2. $O_{t+1}O_{t+2} = I(x)E$. There are two subcases, depending on whether x is in $L(i,k)$ or not.

Case 2a. x is in $L(i,k)$. In this case, the set resulting from $L(i,k)O_{t+1}$ has one more element than $L(i,k)$ and consists of the last $k+1$ elements in $L(i+1)$. But the next operation is an E, which will delete one of these elements—namely the best one in $L(i+1,k+1)$. Thus, the combined affect of $I(x)$ and E is that the set resulting from $L(i,k)O_{t+1}O_{t+2}$ is $L(i+2,k)$. Therefore, it is correct to say that the successor of $L(i,k)$ is $L(i+2,k)$.

Case 2b. x is not in $L(i,k)$. In this case, the set resulting from $L(i,k)O_{t+1}$ has one more element than $L(i,k)$ but does not consist of the worst $k+1$ elements in $L(i+1)$; it consists of $L(i+1,k)$ plus the element $x \in L(i)$, which is less than all the elements in $L(i+1,k)$. But the next operation is an E, and, since x is the best element in the set resulting from $L(i,k)O_{t+1}$, it will delete x. Thus, in this case, the combined effect of $I(x)$ and E is that the set resulting from $L(i,k)O_{t+1}O_{t+2}$ is $L(i+2,k)$. Therefore, it is correct to say that the successor of $L(i,k)$ is $L(i+2,k)$. □

The successor function for $L(i,k)$'s defines a forest $\mathscr{F}$ whose $O(n^2)$ nodes are the $L(i,k)$'s for which $O_t \neq I(x)$, and such that the edge emanating out of $L(i,k)$ goes to its successor node (Figure 2 shows such a forest $\mathscr{F}$). Note that the only nodes with no predecessors, that is, the source nodes, are the $L(0,k)$'s, and the only ones with no successors, that is, the sink nodes, are the $L(m,k)$'s. The problem of computing the $f(i)$'s then becomes a path finding problem in $\mathscr{F}$, where we wish to compute the path of successors in $\mathscr{F}$ from $L(0,|L_1|)$ to the appropriate $L(m,k)$. This path is drawn in heavy lines in Figure 2. Marking this path can easily be done in $O(\log n)$ time using $O(n^2)$

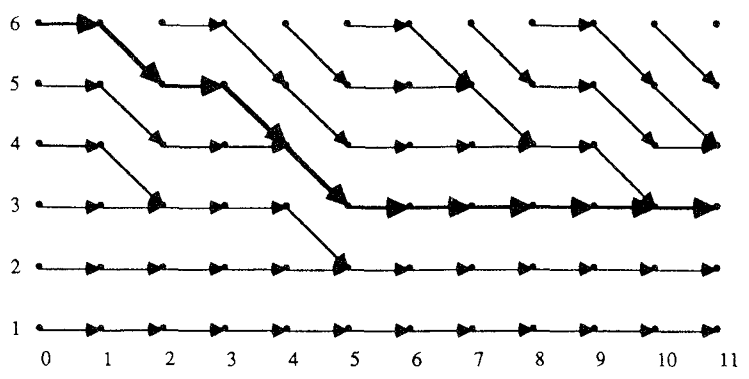

FIG. 2. An example of a successor forest $\mathscr{F}$.

processors, by a simple pointer-doubling scheme. Thus, we get the following lemma:

LEMMA 3.1.6. *Given a sequence S of n $I(x)$, $D(x)$, and E operations, one can evaluate $\varnothing S$ in $O(\log^2 n)$ time using $O(n^2)$ processors in the CREW PRAM model.*

In the next subsection, we show how to use the relationships established in the above discussion to reduce the total work to $O(n \log^2 n)$ while only increasing the time by a $\log\log n$ factor.

3.2. STREAM-LINING THE CONSTRUCTION. The previous subsection essentially reduces the problem to the following path problem. We are given a *grid* G whose columns are numbered $\{0,\dots,h\}$ and whose rows are numbered $\{1,\dots,l\}$, and a *threshold* integer value n_i for every column i of the grid. A node at row r and column c is numbered (c, r) rather than (r, c), in keeping with the notation of the previous subsection (where $L(i)$ was thought of as representing "column i" and $L(i,k)$ as representing "the k-suffix of column i"). There is one edge leaving each node (i,k) if $k < h$: that edge goes to node $(i+1,k)$ if $k < n_i$, to node $(i+1, k-1)$ if $k \geq n_i$. No edge leaves any node of the form (h,k) (i.e., a node in the last column). We want to mark, for each column, the node in it reachable from node $(0,l)$.

Note. The correspondence with the notation of the previous subsection is as follows:

Here h is the number of relevant operations of $\hat{S}$, and $l = |L_1|$. Also, in the forest $\mathscr{F}$ of the previous subsection, for some columns i we had a "successor" edge from $L(i,k)$ to $L(s(i),k)$ for all k (i.e., irrespective of any n_i value); this situation is modeled here by considering n_i to be ∞ for each such column i.

The first thing to observe is that, if we start at any (i,k) in grid G and take s steps, we end up at an $(i+s, k')$ where $k - s \leq k' \leq k$ (this follows from the fact that when moving along an edge we either stay at the same row or move down one row). Let $\lambda_{i,s}(k)$ denote $k - k'$; that is, starting at (i,k) and taking s steps brings us down by $\lambda_{i,s}(k)$ rows, where $0 \leq \lambda_{i,s}(k) \leq s$. Suppose that, for a given i and s, we partition the nodes at column i into equivalence classes as per their $\lambda_{i,s}(k)$ values: nodes (i,k_1) and (i,k_2) are in the same class if and only if $\lambda_{i,s}(k_1) = \lambda_{i,s}(k_2)$. Let $\Gamma_{i,s}$ denote this partition of column i into equivalence classes. In $\Gamma_{i,s}$, *equivalence class* α is the element of $\Gamma_{i,s}$ consisting of the row indices k for which $\lambda_{i,s}(k) = \alpha$. In Figure 2, $\Gamma_{0,2}$ consists of two equivalence classes: class 0 consisting of $\{1, 2, 3\}$, and class 1 consisting of $\{4, 5, 6\}$.

LEMMA 3.2.1. *$\Gamma_{i,s}$ contains $\leq s + 1$ equivalence classes. Each equivalence class is a contiguous interval of row indices. Furthermore, for any two equivalence classes α and β where $\alpha < \beta$, the row indices of equivalence class α are smaller than those of equivalence class β.*

PROOF. A straightforward induction on s. □

Thus, if in a given $\Gamma_{i,s}$ partition we let the highest (respectively, lowest) row number of equivalence class α be u_α (respectively, l_α), then equivalence α consists of the nodes $(i, l_\alpha), (i, l_\alpha + 1), \ldots, (i, u_\alpha - 1), (i, u_\alpha)$. Thus, we do not need to explicitly store equivalence class α: we can just remember the beginning and end of its interval of row indices (we call these the *endpoint row indices* of that class). In Figure 2, the endpoints of equivalence class 1 of $\Gamma_{0,2}$ are 4 and 6. Hence, $O(s)$ space suffices to describe $\Gamma_{i,s}$. Of course, the trade-off of such an implicit representation of $\Gamma_{i,s}$ is that, for a particular k, in order to compute $\lambda_{i,s}(k)$, we now need to locate k in one of the $O(s)$ intervals of $\Gamma_{i,s}$.

A by-product of the above representation is that, given $\Gamma_{i,s}$ and $\Gamma_{i+s,s}$, one can obtain $\Gamma_{i,2s}$ in $O(\log\log s)$ time and $O(s)$ work in the CREW-PRAM model. This is done by using parallel merging to implement the following:

(1) Create a sorted sequence σ consisting of the elements $k - \alpha$ where k is an endpoint row index of class α in $\Gamma_{i,s}$ (i.e., σ contains $k - \alpha$ for all such pairs k, α). Note that $|\sigma| = O(s)$. Also note that σ may contain more than one copy of an element, since the sum $k - \alpha$ might be achieved for more than one pair k, α: in that case, we "remember" where a copy came from by attaching to each such $k - \alpha$ a reminder that this entry was caused by row endpoint k of equivalence class α.

(2) Locate the relative positions of the elements of (i.e., "cross-rank") the following two sequences: (i) σ, and (ii) the sequence σ' of the endpoint row indices of $\Gamma_{i+s,s}$. This "cross-ranking" is done by merging σ and σ'.

(3) For each $k - \alpha$ in σ, if $k - \alpha$ is equal to an entry k' that is in equivalence class β of $\Gamma_{i+s,s}$ (not necessarily as a row endpoint), then we mark k as being a row endpoint of equivalence class $\alpha - \beta$ in $\Gamma_{i,2s}$.

   Note: More than one such k might have $k - \alpha = k'$ for the same k' value, but these k's become the row endpoints of different equivalence classes of $\Gamma_{i,2s}$, since each of them is in a different equivalence class of $\Gamma_{i,s}$.

(4) For each element k' of σ' that does *not* coincide with any $k - \alpha$ of σ, locate the equivalence class (say, α) of $\Gamma_{i,s}$ that contains the point (i, k) such that $k - \alpha = k'$ (note that this $k - \alpha$ is *not* in σ', since k is not a row endpoint of $\Gamma_{i,s}$). Mark k as being a row endpoint of equivalence class $\alpha - \beta$ of $\Gamma_{i,2s}$, where β is the equivalence class of $\Gamma_{i+s,s}$ that contains k'.

Note. It is not hard to see that the point k is unique, since the only way there can be two such k's is if they are both row endpoints in $\Gamma_{i,s}$.

The above has shown how to obtain $\Gamma_{i,2s}$ from $\Gamma_{i,s}$ and $\Gamma_{i+s,s}$. Now, for each row endpoint k in $\Gamma_{i,2s}$, let $cut_{i,2s}(k)$ be the row index at which the path from (i,k) intersects column $i+s$ (the "middle" column). That is, node $(i+s, cut_{i,2s}(k))$ is reachable from node (i,k). The computation of the $cut_{i,2s}(k)$'s can easily be incorporated into the above "combining" procedure for obtaining $\Gamma_{i,2s}$ from $\Gamma_{i,s}$ and $\Gamma_{i+s,s}$: in both steps (3) and (4), simply set $cut_{i,2s}(k)$ equal to $k' = k - \alpha$.

We are now ready to describe the procedure for marking the nodes reachable from node $(0,l)$. Build a complete binary tree T on top of the column indices, where each node v of T has associated with it an interval $I(v)$ of column indices: If v is a leaf, then $I(v)$ is the column index associated with it, and if v is an internal node then $I(v)$ is the union of the two intervals associated with its two children. Thus, if v is at height j then $|I(v)| = 2^j$. Let $first(v)$ be the smallest column index in $I(v)$. The computation consists of two stages, which we describe next.

The first stage builds, in a "bottom-up" fashion, $\Gamma_{first(v),|I(v)|}$ for each node v in T. While doing so, it also computes the $cut_{first(v),|I(v)|}(k)$'s for that node v. This is done in $O(\log n \log\log n)$ time and $O(n \log n)$ work by using the above-mentioned combining procedure once at each node v (here $n = h + l$).

The second stage uses the results of the first stage to *mark*, in each column, the node that is reachable from node $(0,l)$. We explain how to do it in $O(\log n)$ time and $O(n)$ processors. The procedure is recursive, and starts at the root. When called at a node v of T, its input also consists of (i) $|I(v)|$ processors, and (ii) a grid node $(first(v), \zeta)$ (ζ need not be a row endpoint of $\Gamma_{first(v),|I(v)|}$). The output is to cause, for each column c in the column interval $I(v)$, the marking of the node of c that is reachable from node $(first(v), \zeta)$ (this marking is permanent in the sense that it does not get undone when the recursive procedure returns). The procedure does this marking as follows:

—Mark grid node $(first(v), \zeta)$. If v is a leaf of T, return. Otherwise, proceed with the following steps.
—Use the $|I(v)|$ processors to locate, in constant time, which equivalence class of $\Gamma_{first(v),|I(v)|}$ contains row index ζ (say it is class γ). Then, in constant time, mark grid node $(first(v) + |I(v)|, \zeta + \gamma)$.
—Recursively call the procedure for the left child u of v in T and grid node $(first(u), \zeta)$, giving it $|I(u)| = |I(v)|/2$ processors.
—Recursively call the procedure for the right child w of v in T and grid node $(first(w), cut_{first(v),|I(v)|}(\zeta))$, giving it $|I(w)| = |I(v)|/2$ processors.

Correctness of the above second stage follows from the definitions. Its complexity bounds are clearly $O(\log n)$ time and $O(n)$ processors.

This completes the proof that the desired path can be marked in $O(\log n \log\log n)$ time and $O(n \log n)$ work, thus implying an $O(\log^2 n \log\log n)$ time and $O(n \log^2 n)$ work solution for the Competitive Deletes problem.

4. *The Off-Line Mergeable Heaps Problem*

The methods of the previous sections only apply when the set-manipulation operations all are for the same set. In this section, we study sequences of operations that can take set names as arguments in addition to specific elements. In particular, we address the problem of evaluating a sequence of operations from the set $\{Insert(x, A), Delete(x), Min(A), Union(A, B), Find(x)\}$. We begin by describing the semantics associated with each operation. Initially, we assume that every set named in the sequence S exists and is empty. Since one of the possible operations in S is $Find(x)$, we also assume that the elements are distinct.

(1) $Insert(x, A)$. Insert x into the set A.
(2) $Delete(x)$. Delete an element x from whichever set it currently belongs to.
(3) $Union(A, B)$. Union the elements of A and B into the set B, destroying A (i.e., no operations after a $Union(A, B)$ can have A as an argument).
(4) $Find(x)$. Determine the name of the set to which x currently belongs.
(5) $Min(A)$. Return the value of the minimum element currently in A. Here, "minimum" can be replaced by any associative operation.

The *element argument* (respectively, *set argument*) of an operation like $Insert(x, A)$ is x (respectively, A). Without loss of generality, one may assume that none of the operations in S are inconsistent (e.g., a $Delete(x)$ issued when x is not in any set), since these can all be eliminated by a simple pre-processing step in which one sorts all the elements referenced in S.

Suppose we are given a sequence $S = O_1O_2 \cdots O_n$ of operations from the above collection. In this section we show how to evaluate $\emptyset S$ in $O(\log n)$ time using $O(n)$ processors. We begin by creating a *union tree* U from S, where the nodes of U are labeled with the set names used in S and there is an edge from a node v, whose label is A, to a node w, whose label is B, if and only if there is an operation $O_t = Union(A, B)$ in S. For the time being, let us assume that U is a proper binary tree (i.e., all internal nodes have exactly two children). We will show later how to relax this condition. For each internal node v whose label is A, the *extinction time* of v (denoted t_v), is the time of evaluation of the operation $Union(A, B)$, that is, $O_{t_v} = Union(A, B)$ (note that A is the first argument). The tree U can easily be created in $O(\log n)$ time using $O(n)$ processors, by sorting [Cole 1988].

Intuitively, our method is to construct a subsequence $I(v)$ of S for each node v in U, which consists of all the operations in S whose element argument (say, x) was originally inserted in the set (say, A) labeling v (i.e., the earliest reference to x in S is an $Insert(x, A)$). We then "percolate" the $I(v)$'s up and down the tree U to construct for each v in U a list (which we will denote by M'_v) of all (t, m) pairs such that O_t involves the set name labeling v (call it A), and m is the minimum value that would be stored in A at that time t (i.e., after a hypothetical sequential evaluation of $\emptyset O_1 \cdots O_t$). We call this the *minimum-history vector* for v. We store the M'_v lists sorted by t values. Given these M'_v lists it is trivial to then print out a solution to $\emptyset S$. Specifically, the solution to an operation $O_t = Find(x)$ is simply the set name labeling the node v such that the list M'_v contains a pair of the form $(t, *)$, and a solution to an $O_t = Min(A)$ is the m value of the pair (t, m) in the M'_v list for the node v that A labels.

We give below an overview of our method for constructing these M'_v lists.

High Level Description

Step 1. In this step, we convert the union tree U into a binary tree T that has $O(n)$ nodes and $O(\log n)$ height (U does not necessarily have $O(\log n)$

height). For each v in U let $I(v)$ denote the subsequence of S consisting of all *Insert*, *Delete*, and *Find* operations O_t such that the element argument of O_t was originally inserted in the set name labeling v. Let T_v be a complete binary tree built "on top" of $I(v)$, where each leaf of T_v is associated with an operation in $I(v)$. We perform a tree-contraction procedure on U, in which we iteratively combine pairs of nodes in U, until U has been reduced to a single node z. Each time we combine two nodes v and w into a node u we combine T_v and T_w into a tree T_u by creating a root for T_u and making the roots of T_v and T_w its children. We let T denote the final tree T_z. We implement this using the tree-contraction scheme of Abrahamson et al. [1989] and Kosaraju and Delcher [1988], which build on the "rake-and-compress" paradigm of Miller and Reif [1985]. This scheme implies that the resulting T has $O(n)$ nodes and $O(\log n)$ height.

Step 2. In this step, we perform a cascade merging procedure in T, similar to that used for the array-of-trees construction, computing for each node $\hat{v}$ the list of all elements stored in descendents of $\hat{v}$ sorted by their execution times. (We use the "hat" accent to distinguish the nodes in T from their corresponding nodes in U.) In addition, for each element in each such list we store the min of the elements present at the execution time associated with that element (as we did in the array-of-trees). For each $\hat{v} \in T$, we let $M_{\hat{v}}$ denote the list of (t, m) pairs, where t is an execution time and m is the minimum for that time over all elements stored in $I(\hat{u})$ lists in the descendents of $\hat{v}$, including $\hat{v}$. We also compute for each node v the maximum of all the extinction times of nodes that were contracted to form v. (Recall that, if v is labeled by set name A, then its extinction time is the time t such that $O_t = Union(A, B)$.)

Step 3. In this step, we perform a reversal of the tree-contraction step (Step 1), in which we iteratively reconstruct the union tree U from T in the reverse order in which T was obtained from U (by "uncontracting" nodes, etc.). As we perform the reversed tree-contraction we maintain a list, M'_v, of (t, m) pairs with each node v in the "current" tree U_i (i.e., the ith tree in the contraction, $i = O(\log n)$). As mentioned above, we define the M'_v lists so that when the procedure completes and we have reconstructed the tree U, M'_v will contain a "history" of all the minimum values stored in the set that labels v.

End of High-Level Description

Before presenting the details for implementing each of the above steps, let us give some notational conventions.

Notation. Given a sorted list A of records, and two values k and l taken from the universe of keys for records in A (with $k \leq l$), we let $A|_{[k,l]}$ denote the sublist of A consisting of all records whose key value falls in the interval $[k, l]$. Given two lists of records A and B whose keys come from the same universe, we let $A \cup B$ denote the merge of sorted A and B.

4.1. STEP 1: CONTRACTING THE UNION TREE. Recall that, for each v in U, $I(v)$ denotes the subsequence of S consisting of all operations O_t such that O_t has an element argument which was initially inserted in the set labeling v. Also recall that T_v is a binary tree built "on top" of $I(v)$. We perform a tree-contraction procedure on U, in which we iteratively combine pairs of nodes in U, until U has been reduced to a single node. We store a pointer in each v to the root of its associated T_v tree, denoted $\hat{v}$. Each time we combine two nodes u and w into a new node v we combine T_u and T_w into a tree T_v by making $\hat{u}$ and $\hat{w}$ be the children of $\hat{v}$.

As mentioned earlier, we implement this step using the tree-contraction scheme of Abrahamson et al. [1989] and Kosaraju and Delcher [1988], which is built upon the rake-and-compress paradigm of Miller and Reif [1985]. We let U_0 denote the initial tree U and iteratively contract U_0, producing U_1, U_2, and so on, until we reach a U_s that is a single node ($s = O(\log n)$). Specifically, we assign an index variable $i := 0$ and perform the following steps:

(1) Number the leaves of U_i from left to right 1, 2, 3, etc.
(2) Combine each odd-numbered leaf v of U_i with its parent z, provided v is a left child. This is commonly called *raking* v [Miller and Reif 1985]. We also combine T_v and T_z into a single tree, as mentioned above. We don't deallocate the space used for the nodes v and z, however. Instead, we store the records for v and z with the nodes $\hat{v}$ and $\hat{z}$, which were previously the roots of T_v and T_z, respectively, and "splice" v and z out of U_i by changing the pointers that point to them. (We shall use these records to help the contraction-reversal step (Step 3).) Let U_{i+1} denote the resulting tree, and assign $i := i + 1$.
(3) For each node v of U_i that had one of its children raked, combine v with its remaining child w (if there is one). This is commonly called a *compress* operation [Miller and Reif 1985]. We also combine T_v and T_w as in the previous step. Let U_{i+1} denote the resulting tree, and assign $i := i + 1$.
(4) Repeat the previous two steps for odd-numbered leaves that are right children.
(5) If the tree U_i resulting from the above four steps has more than one node, then repeat the previous four steps for U_i.

It should be clear that, given a processor assigned to each leaf, each iteration of the above procedure can be implemented in $O(1)$ time. In addition, since each iteration eliminates half of the leaf nodes, there are at most $O(\log n)$ iterations. This implies that the tree $T = T_z$ resulting from the last execution of Steps 2–3 has $O(\log n)$ height and $O(n)$ nodes. (In fact, it follows from Abrahamson et al. [1989] and Kosaraju and Delcher [1988], that the entire procedure can be implemented in $O(\log n)$ time using only $O(n/\log n)$ processors.)

4.2. STEP 2: CASCADING IN THE TREE T. In this step, we perform a cascade merging procedure on T, computing for each node v in T the list of all elements stored in descendents of v sorted by their execution times. In addition, for each element in each list, we store the min of the elements present at the execution time of that element (as in the array-of-trees section). For each $\hat{v}$ in T, we let $M_{\hat{v}}$ denote the list of (t, m) pairs, where t is an execution time and m is the minimum for that time. We also compute for each node $\hat{v}$ in T the maximum of all the extinction times of nodes in U associated with descendents of $\hat{v}$ (including itself).

Let v be a node in some U_i, and let *Nodes*(v) be the set of nodes of U that were combined to form v. Let us generalize the definition of $I(v)$ to nodes in U_i so that $I(v)$ denotes the subsequence of S consisting of all the operations O_t

such that O_t has an element argument which was initially inserted in the set labeling one of the nodes in $Nodes(v)$. Since $\hat{v}$ is both the root of T_v and a node in T, it stores a list $M_{\hat{v}}$, which can be viewed as the history of minimums for $I(v)$ as if all the operations in $I(v)$ were for the same set. In addition, $M_{\hat{v}} = M_{\hat{a}} \cup M_{\hat{b}}$, where $\hat{a}$ and $\hat{b}$ are the children of $\hat{v}$. So, just as with the array-of-trees data structure, we can compute each (t, m) pair in each $M_{\hat{v}}$ by applying the cascading divide-and-conquer scheme [Atallah et al. 1989, Cole 1988] to achieve a running time that is $O(\log n)$ using $O(n)$ processors.

In the next step, we take advantage of the properties of T and its $M_{\hat{v}}$ lists to complete the evaluation of $\varnothing S$.

4.3. Step 3: Reversing the Tree-Contraction to Reconstruct U. In this step, we perform a reversal of the tree-contraction step (Step 1). Let v be a node in some U_i. We let $Ops(v)$ denote the subsequence of S consisting of all the operations O_t such that O_t has an element argument which was initially inserted in the set labeling a node in $Nodes(w)$ for some descendent w of v in U_i (including v itself). Note that the operations in $Ops(v)$ are all the operations that could possibly affect v. We let $Up(v)$ denote the minimum-history vector for the operations in $Ops(v)$ as if they all applied to the same set, restricted to the range $[t_v, +\infty]$, where t_v denotes the maximum extinction time of nodes in $Nodes(v)$. (This minimum history vector corresponds to information that must be passed up from v to nodes higher in U_i.)

For each node v with parent z in the current tree U_i, we maintain a list M'_v, which is defined as follows (recall that $A|_{[k,l]}$ denotes the sublist of a sorted list A consisting of all records whose key value falls in the interval $[k, l]$):

(1) If v has no children, then $M'_v = M_{\hat{v}}|_{[0, t_z]}$.
(2) If v has one child, u, then $M'_v = M_{\hat{v}}|_{[0, t_z]} \cup Up(u)|_{[t_u, t_z]}$.
(3) If v has two children, u and w, then $M'_v = M_{\hat{v}}|_{[0, t_z]} \cup Up(u)|_{[t_u, t_z]} \cup Up(w)|_{[t_w, t_z]}$.

The m value for each (t, m) in M'_v is determined in the obvious way: Namely, by taking the minimum of the m values of the (t', m) pairs in the sets unioned to define M'_v, where t' is the immediate predecessor of t.

As mentioned above, our method is based on the observation that if $U_i = U$, then, for each v in U, the list M'_v will contain a history of all the minimum values stored in the set that labels v. We iteratively reverse the tree-contraction step, converting U_{i+1} back to U_i, while maintaining M'_v lists for each v in the current tree. In the next lemma, we establish an important relationship between the M and M' lists, which we exploit for quickly reconstructing U in parallel.

Lemma 4.3.1. *Suppose a and b are the two nodes of U_i that were combined to form some v in U_{i+1}. Without loss of generality, let b be the child of a (so $t_b < t_a$). Suppose further that z is the parent of a in U_i (if z does not exist, then take $t_z = +\infty$). (See Figure 3.) Then we have the following relationships for M'_a and M'_b:*

Case 1. *The node v has no children in U_{i+1}. Then $M'_a = M_{\hat{a}}|_{[0, t_z]} \cup M_{\hat{b}}|_{[t_b, t_z]}$, and $M'_b = M_{\hat{b}}|_{[0, t_a]}$.*

Case 2. *The node v has a child, w, in U_{i+1}.*

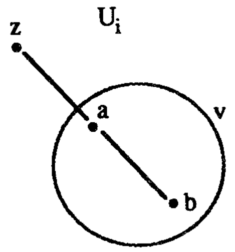

FIG. 3. Illustrating Lemma 4.3.1.

Case 2.a. *a and b were combined by a rake operation. Then* $M'_a = M_{\hat{a}}|_{[0,t_z]} \cup M_{\hat{b}}|_{[t_b,t_z]} \cup M'_w|_{[t_w,t_z]}$, *and* $M'_b = M_{\hat{b}}|_{[0,t_a]}$.

Case 2.b. *a and b were combined by a compress operation. Then* $M'_a = M_{\hat{a}}|_{[0,t_a]} \cup M_{\hat{b}}|_{[t_b,t_z]} \cup M'_w|_{[t_b,t_z]}$, *and* $M'_b = M_{\hat{b}}|_{[0,t_a]} \cup M'_w|_{[t_w,t_a]}$. *In addition, we can assign* $M'_w := M'_w|_{[0,t_b]}$ *in* U_i, *since b is the parent of w in* U_i *(the old* M'_w *extended to* $t_v = t_a$*).*

The m value for each (t, m) *in* M'_a *(respectively,* M'_b*) is determined by taking the minimum of the m values of the* (t', m) *pairs in the sets unioned to define* M'_v, *where* t' *is the immediate predecessor of t.*

PROOF. The proof is by induction on the iteration number i of the reversed contraction procedure (note that i decreases as the algorithm progresses). Initially, U_{i+1} is a single node. Thus, Case 1 applies. The lemma follows from the fact, then, that b is a leaf and a has no other children. Suppose, then, that the lemma holds for the nodes in U_{i+1}. Consider U_i. If a node v is a leaf in U_{i+1}, then Case 1 applies, and is clearly correct. So suppose v has a child w in U_{i+1}.

Case 1. a and b were combined by a rake operation. In this case, a has children b and w in U_i. The lemma follows in this case, since b is a leaf, and, by induction, M'_w restricted to $[t_w, t_z]$ must be the same as $Up(w)$ restricted to $[t_w, t_z]$.

Case 2. a and b were combined by a compress operation. In this case, in U_i, w is the only child of b, which is, in turn, the only child of a. The formula for M'_a follows, by induction, from the fact that $Up(b)|_{[t_b,t_z]} = M_{\hat{b}}|_{[t_b,t_z]} \cup M'_w|_{[t_b,t_z]}$, since $Up(b) = M_{\hat{b}}|_{[t_b,\infty]} \cup Up(w)|_{[t_b,\infty]}$. The formulas for M'_b and M'_w follow immediately from induction.

This completes the proof. □

Thus, we have a method for constructing U_i with all its M'_v lists, given U_{i+1} and its M'_v lists. We have yet to describe how we implement each step of the reversed contraction routine in $O(1)$ time using $O(n)$ processors, however.

Initially, we assign two processors, which we call a *processor pair*, to each element in M'_z, where z is the single node to which U was contracted. As we reverse each iteration of the tree-contraction step (Step 1) we maintain the M'_v lists as mentioned above and two important ranking invariants: (i) that M'_v is ranked in $M_{\hat{v}}$, for each v in U_i, where $\hat{v}$ is the root of T_v, and (ii) that M'_v is ranked into M'_w, for each v in U_i, where w is a child of v. (Recall that a list A is *ranked* in a list B if we know the rank of the predecessor in B of each element a in A [Cole 1988].) We can easily maintain these ranking invariants as the procedure progresses, since, for each invariant of the form "A is ranked in B" that we wish to maintain, we have $B \subseteq A$. In addition to these two ranking invariants, we assume that $M_{\hat{v}}$ is ranked in $M_{\hat{a}}$ and $M_{\hat{b}}$, where $\hat{a}$ and $\hat{b}$ are the children of $\hat{v}$, the root of T_v, since this comes for free from the cascading procedure (recall that $M_{\hat{v}} = M_{\hat{a}} \cup M_{\hat{b}}$).

Let us, then, describe how to implement each of the uncontract steps. Let a and b be the two nodes of U_i that were combined to form v in U_{i+1}, with b being the child of a. Let us consider the possible cases:

Case 1. v is a leaf in U_{i+1}. In this case, we can construct M'_a in $O(1)$ time, since (i) M'_v is ranked in $M_{\hat{v}}$, and (ii) $M_{\hat{v}}$ is ranked in $M_{\hat{a}}$ and $M_{\hat{b}}$. In addition, there is an element in M'_v for each element in $M_{\hat{v}}$ (hence, each element of $M_{\hat{a}}|_{[0,t_z]}$ and $M_{\hat{b}}|_{[t_b,t_z]}$). This implies that we can use the processors associated with the elements of M'_v to construct M'_a, and assign these processors to M'_a. Some of these processors may be needed in M'_b, however. In particular, the elements in $M_{\hat{b}}|_{[t_b,t_a]}$ are needed for both M'_a and M'_b. In this case, we split the processor pairs for these elements, assigning a single processor to the copy of each element from $M_{\hat{b}}|_{[t_b,t_a]}$ in M'_a and a processor to each element from $M_{\hat{b}}|_{[t_b,t_a]}$ in M'_b. We will show later that once a processor pair has been split for an element t; we will never again attempt such a split again for t (in any list). This does not give us all the processors needed for M'_b, but, fortunately, for M'_b, there is an element in M'_v for each element in $M_{\hat{b}}|_{[0,t_b]}$, and none of these elements are needed to form M'_a. Thus, we can reassign the processors assigned to these elements to their counterparts in M'_b.

Case 2.a. a and b were combined by a rake operation, and v has a child, w, in U_{i+1}. We can construct M'_a in $O(1)$ time in this case, since (i) M'_v is ranked in M'_w, (ii) M'_v is ranked in $M_{\hat{v}}$, and (iii) $M_{\hat{v}}$ is ranked in $M_{\hat{a}}$ and $M_{\hat{b}}$. In addition, there is an element in M'_v for each element in $M_{\hat{v}}$ (hence, each element of $M_{\hat{a}}|_{[0,t_z]}$ and $M_{\hat{b}}|_{[t_b,t_z]}$) and for each element in $M'_w|_{[t_w,t_z]}$. This implies that we can use the processors associated with the elements of M'_v to construct M'_a, and in turn assign these processors to M'_a. As in the previous case, we may need some of these processors for M'_b, however. As before, for M'_b, there is an element in M'_v for each element in $M_{\hat{b}}|_{[0,t_b]}$, and none of these elements are needed to form M'_a, but the elements in $M_{\hat{b}}|_{[t_b,t_a]}$ are needed for both M'_a and M'_b. So, as before, we split the processor pairs for these elements, assigning a processor to the copy of the element from $M_{\hat{b}}|_{[t_b,t_a]}$ in M'_a and a processor to the copy in M'_b.

Case 2.b. a and b were combined by a compress operation, and v has a child, w, in U_{i+1}. Let z be the parent of a in U_i. We can construct M'_a in $O(1)$ time, by essentially the same method as in the previous case. A similar method constructs M'_b in $O(1)$ time. The processor assignments are more involved, however. As before, there is an element in M'_v for each element in M'_a. Thus, if we were only interested in constructing M'_a, the processor assignment would be trivial. Recall, however, that $M'_b = M_{\hat{b}}|_{[0,t_a]} \cup M'_w|_{[t_w,t_a]}$, and, since b is a child of a, $M'_b|_{[t_b,t_a]}$ must be a subset of M'_a. In this case, we do not resolve the overlap by processor-pair splitting alone. We only split processor pairs for the elements in $M_{\hat{b}}|_{[t_b,t_a]}$ (giving a processor to each copy in M'_b and M'_a of each element from $M_{\hat{b}}|_{[t_b,t_a]}$). We do not need to split processor pairs for the elements from $M'_w|_{[t_w,t_a]}$. Instead, we can locate a sufficient number of processors assigned to elements in lists of U_{i+1} such that these processors are no longer needed in the corresponding lists of U_i. Specifically, the elements in $M'_w|_{[t_w, t_b]}$ were in M'_v of U_{i+1}, but these elements are not in M'_a. Thus, we can reallocate the processors for the copies of these elements in M'_v (of U_{i+1}) to their copies in M'_b (of U_i). In addition, the elements in $M'_w|_{[t_b, t_a]}$ (of U_{i+1}) need no longer be stored in M'_w (of U_i), since w's parent in U_i is b. Thus, we can reallocate the processors for the copies of these elements in M'_w (of U_{i+1}) to their copies in M'_b (of U_i). This completes the description of the method for implementing each round of the reversed tree-contraction in $O(1)$ time using $O(n)$ processors. The following lemma completes the proof of correctness of this implementation:

LEMMA 4.3.2. *At no point in the computation will we ever try to perform a processor-pair split for an element that is assigned only one processor.*

PROOF. Any time we split a processor pair for an element t, we do so only if t is in an interval $[t_b, t_a]$ where b is the child of a in the tree U_i. Since t_b is the extinction time for b, all the extinction times for nodes (in *Nodes*(b)) that were combined to form b must necessarily be less than t_b. Thus, all the future processor-pair splits done for nodes in *Nodes*(b) must involve elements that are not in the interval $[t_b, t_a]$. So, the only possible illegal processor-pair splits must come from nodes in *Nodes*(a). But we will have performed processor splits only for the elements of $M_{\hat{b}}|_{[t_b, t_a]}$ (which are also in M'_a). These elements are not in $M_{\hat{a}}$, however. Thus, these elements are not in $M_{\hat{c}}$ for any node $\hat{c}$ in *Nodes*(a). This completes the proof. □

Thus, we have the following lemma:

LEMMA 4.3.3. *Suppose one is given a sequence S of Insert(x, A), Delete(x), Union(A, B), Find(x), and Min(A) operations. If the tree determined by the Union(A, B) operations in S is a proper binary tree, then one can evaluate$\emptyset S$ in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.*

In the next subsection, we show how to extend this lemma to arbitrary union trees.

4.4. ALLOWING FOR NONBINARY UNION TREES. The tree, U, determined by the *Union*(A, B) operations in S does not have to be a proper binary tree for us to be able to evaluate S in $O(\log n)$ time using $O(n)$ processors. In this subsection, we show how to transform U into a proper binary tree U', such that applying the above procedure on U' can easily be converted into a solution for U. The method for converting U into U' consists of two steps. The first step adds a "dummy" child to each node with only one child, and the second step adds dummy descendents to a node v if v has more than two children, so as to "fan in" the sets coming from the children of v.

Step 1. Let v be a node in U that has only one child, w. Let $O_t =$ *Union*(A, B) be the union operation in S that determines the edge from w to v, that is, A is the set name labeling w and B is the set name labeling v. We add an operation *Union*(Z, B) just before O_t in S, where Z is a set not referenced by any operation in S. Let S' denote the resulting sequence.

Comment. It is easy to see that Step 1 forces v, the node labeled by B, to have two children in the union tree determined by S'. Moreover, since Z is not referenced by any other operation in S, the response to an operation O in S is the same as its response in S'.

Step 2. Let U be the union tree determined by the operations of S'; so each node in U has at least two children. Let v be a node in U that has children $w_1, w_2, \ldots, w_k$ such that $k \geq 3$. Order these children of v so that $t_{w_i} < t_{w_{i+1}}$ for $i \in \{1, 2, \ldots, k-1\}$. We modify U by building a complete binary tree B_v whose leaves are $w_1, w_2, \ldots, w_k$ and whose root is v. For each internal node u in B_v, we make the extinction time for u, denoted t_u, be the maximum of the extinction times of u's descendents in B_v. Let U' denote the resulting union tree. Clearly, U' is a proper binary tree.

Comment. U' clearly has $O(|U|)$ nodes. The only difference between U' and the union tree of this algorithm is that for any child–parent pair (b, a) in U' we have $t_b \leq t_a$, instead of $t_b < t_a$. This does not change the correctness of Lemma 4.1, however. Thus, we can implement the algorithm of Lemma 4.3 on U' so as to still run in $O(\log n)$ time using $O(n)$ processors. So we have only to convert the solution to U' to a solution for U.

For any node v in the (nonbinary) union tree determined by S, if v has at most two children, then, by arguments given above, the list M'_v for the corresponding node v in U' is the same as M'_v would be in the union tree determined by S. So, let v be a node that has children $w_1, w_2, \ldots, w_k$ in U such that $k \geq 3$. We show how to construct the M'_v list for v in U, given the M'_u list for each node u in B_v of U'.

Let (t, m) be a pair in some M'_u list for an internal node u of B_v (u may be v). Since (t, m) is in an M'_u list for an internal node of B_v, there must be a pair (t, m^*) in M'_v in U (i.e., with the same first coordinate). Thus, we have only to determine the minimum value, m^*, associated with this pair. Let π be the path from u to v in B_v. Since the leaves of B_v are listed left-to-right by increasing extinction times, any leaf w_i that is the descendent of a node on the left fringe of π must have $t_{w_i} < t$. In addition, any leaf w_i that is the descendent of a node on the right fringe of π must have $t_{w_i} > t$. (Recall that a node is on the *left fringe* (respectively, *right fringe*) of a path π if it is not on π but is the left child (respectively, right child) of a node on π.) If $m < m^*$, then m^* must belong to a pair (t', m^*) in some M'_z list, where z is on the left fringe of π and t' is the immediate predecessor of t in M'_z. This is because t has no immediate predecessors in any of M'_z list if z is on π or the right fringe of π. Thus, to determine the value of m^*, we have only to assign a processor to the pair (t, m) and have that processor locate the immediate predecessor of t in each M'_z list such that z is on the left fringe of π. If we were to implement the query for this processor by performing a binary search in each M'_z list such that z is on the left fringe of π, then the running time of our algorithm would grow to be $O(\log^2 n)$. Thus, we must be more clever in how we implement this query.

To perform the query for a pair (t, m) in M'_u, it certainly is sufficient for the processor for (t, m) to locate in each M'_z the pair (t', m') such that t' is the immediate predecessor of t, where z is a node on the walk ω in B_v that starts from u, and traverses up B_v, visiting each node on π and each node on the left fringe of π. Such a traversal is defined as a multilocation of t in ω [Atallah et al. 1989]. Atallah et al. [1989] show that one can perform such a multilocation of t in ω in $O(\log N + |\omega|)$ time, where $|\omega|$ is the number of nodes in ω, given a preprocessing step that takes $O(\log N)$ time using $O(N/\log N)$ processors, where N is the total size of the graph being searched, including all the lists it contains. In our case, N is $O(n)$, since there can be at most two pairs in M'_t lists of U with the same t value (i.e., in the M'_v list for a node v and in the M'_z list for its parent, z). In addition, $|\omega|$ is $O(\log n)$. Thus, we can determine the value of m^* for each (t, m) pair such that (t, m) is in some M'_u list for a node u in U' in $O(\log n)$ time using $O(n)$ processors. This gives us the correct M'_v list for each node v in U; hence, gives us the following theorem:

THEOREM 4.4.1. *Given a sequence S of Insert(x, A), Delete(x), Union(A, B), Find(x), and Min(A) operations, one can evaluate $\varnothing S$ in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.*

In the next section, we address the off-line priority queue problem.

5. *The Off-Line Priority Queue Problem*

In this section, we show that one can evaluate S in $O(\log n)$ time using $O(n)$ processors when the operations in S are $I(x)$ and E, i.e., an off-line priority queue problem. This is optimal, because one can easily reduce sorting to this problem. Our algorithm generalizes an algorithm by Dekel and Sahni for processor scheduling [Dekel and Sahni 1984] that can be applied to this problem, which ran in $O(\log^2 n)$ time using $O(n)$ processors. The main contribution of our algorithm is the development and application of generalized cascade merging to the off-line priority queue problem.

Let $S = O_1O_2 \cdots O_n$ be a sequence of *Insert*(x) and *ExtractMin* operations. We wish to evaluate $\varnothing S$. As mentioned earlier, Dekel and Sahni [1984] study a related processor scheduling problem, namely, that of finding a schedule for n jobs, specified by release times and deadlines, so as to minimize the maximum lateness. Their solution amounts to a reduction of this scheduling problem to the $\{I(x), E\}$ evaluation problem, which is essentially the sequential method used by Horn [1974]. If the sequence S does not contain any redundant E's, then the method used by Dekel and Sahni can be applied directly to solve the $\{I(x), E\}$ evaluation problem, resulting in a solution running in $O(\log^2 n)$ time using $O(n)$ processors. If there can be redundant E's, then one must precede their algorithm by a parallel prefix computation to eliminate the redundant E's.

The main idea of the Dekel–Sahni algorithm is to build a complete binary tree "on top" of the operations in S and then perform two "passes" over this tree—the first flowing up the tree and the second flowing down the tree. Our method uses a similar approach, except that each pass is implemented by a generalized cascade merging procedure. We perform this procedure in two directed acyclic graphs (dag's), rather than using a tree. The dag we use for the first pass is derived from a recursive merging procedure similar to that used in the first pass of the algorithm by Dekel and Sahni. Since some nodes in this dag have out-degree 2 (i.e., two "parents"), one of the important aspects of our implementation is showing how to perform cascade merging in this dag using only $O(n)$ processors. This is also true for the dag we use to implement our second phase, for it too contains nodes that have out-degree 2. This second dag is derived from a "merge-and-purge" procedure that is quite different from the second phase of the Dekel–Sahni algorithm (in fact, it is not clear that one can efficiently implement their second phase with a cascade merging procedure). We give the details of our algorithm below.

We begin by constructing a complete binary tree T "on top" of S so that each leaf of T is associated with a single operation O_t (listed from left to right). For each node v let $e(v)$ denote the number of *ExtractMin* operations stored in the descendent leaves of v. One can compute $e(v)$ for each v in T in $O(\log n)$ time using $O(n/\log n)$ processors by a simple bottom-up summation computation in T. For every leaf of T corresponding to an E operation, we replace that leaf with a node v with two leaf-node children such that its left child corresponds to an $I(\infty)$ operation and its right child corresponds to an E. This allows us to assume that each E has a response. That is, the ∞'s are added so that the response to an E is ∞ if and only if its response should be "set empty" in $\emptyset S$.

For each v in T, let $S(v)$ denote the substring of S that corresponds to the descendents of v. For each v in T, we will compute two sets $A(v)$ and $L(v)$: $A(v)$ will be the sorted list of answers to all the E's in $\emptyset S(v)$ (recall that this denotes performing $S(v)$ with the set of elements initialized to $\emptyset$), and $L(v)$ will be the sorted list of elements left in the set after we perform $\emptyset S(v)$. For any list B and integer m, we let $\mathit{Prefix}_m(B)$ denote the list consisting of the first m elements in B (if $|B| < m$, then $\mathit{Prefix}_m(B) = B$). Similarly, we let $\mathit{Suffix}_m(B)$ denote the list consisting of the last m elements in B (if $|B| < m$, then $\mathit{Suffix}_m(B) = B$).

LEMMA 5.1. *Let S be a sequence of I and E operations, and let L be a sorted list of elements. If A is the sorted list of answers from $\emptyset S$, then $\mathit{Prefix}_{|A|}(L \cup A)$ is the list of answers from LS.*

PROOF. The proof follows from arguments given in Dekel and Sahni [1983]. □

This immediately implies the following corollary:

COROLLARY 5.2. *Let v be a node in T with left child x and right child y. Then we have the following relationships*:

$$A(v) = A(x) \cup \mathit{Prefix}_{e(y)}(L(x) \cup A(y)),$$
$$L(v) = L(y) \cup \mathit{Suffix}_{n_x - e(x)}(L(x) \cup A(y)).$$

In words, this states that the answers in $A(v)$ that are for *extractMin* operations that are stored in descendents of y come from the first $e(y)$ elements of $L(x) \cup A(y)$. We shall use this lemma to construct $A(v)$ and $L(v)$ for every v in T. We begin by constructing a dag G from T by expanding each node v into T into five nodes: $[Av]$, $[Lv]$, $[Sv]$, $[Axv]$, and $[Lyv]$, where x and y are the left and right children of v, respectively. For each such node v of T, the following are edges in G: $([Ax],[Axv])$, $([Lx],[Sv])$, $([Ay],[Sv])$, $([Ly],[Lyv])$, $([Axv],[Av])$, $([Sv],[Av])$, $([Sv],[Lv])$, and $([Lyv],[Lv])$. (See Figure 4.) Before we explain the role of each of the five nodes of G that correspond to a node $v \in T$, we observe that G consists of a number of layers equal to twice the height of T (hence, G has $O(\log n)$ layers). This is because the definition of the edges of G is such that, if v is on level l in T, then the nodes $[Av]$ and $[Lv]$ are on level $2l - 1$ in G and the nodes $[Sv]$, $[Axv]$, and

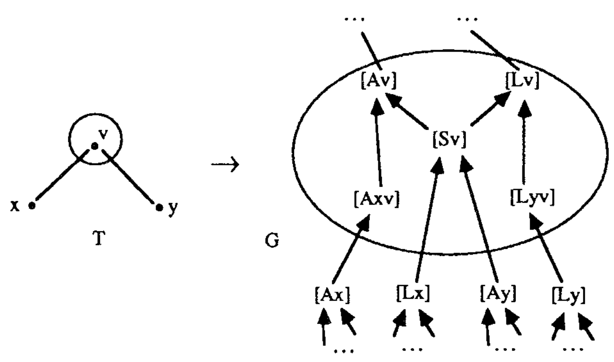

FIG. 4. The upward cascade merging procedure.

$[Lyv]$ are on level $2l$. We now discuss the roles played by each of the five nodes of G corresponding to a $v \in T$. We will construct a single sorted list for each node in G by a cascade merging procedure [Atallah et al. 1989]. We generalize the method of Atallah et al. [1989] and Cole [1988], however, in that the input to a node v in G will not necessarily be strictly a sorted merge of the lists at the in-nodes of v. The set we will build at $[Av]$ is $A(v)$ and the set we will build for $[Lv]$ is $L(v)$. Intuitively, $[Sv]$ is a "splitter" node, as its output will be split between $[Av]$ and $[Lv]$. The nodes $[Axv]$ and $[Lyv]$ are added so as to synchronize the flow from level to level. We perform a cascade merging computation in G that proceeds in stages, where, for each stage t, each node $[\alpha]$ in G will store a list $U_t([\alpha])$. Initially, $U_t([\alpha])$ is empty for all but the nodes that correspond to leaves of T. Specifically, if v is a leaf of T, then (i) $U_0([Av]) = \{\infty\}$, and (ii) $U_0([Lv])$ equals $\{x\}$ is $O_v = I(x)$, $\{\infty\}$ if $O_v = E$. We say that a node v of G becomes *full* in stage t if $U_t(v)$ will equal $U_{t'}(v)$ for all $t' > t$. Intuitively, v is full when $U_t(v)$ contains all the elements it was intended to have. In our procedure, which we describe below, we can easily test if a node becomes full in stage t as soon as it happens (because we know the final size of the sorted list we are building at each such node).

Let $Samp_{v,t}(U_t(v))$ denote the *sample* of $U_t(v)$ at node v, defined as follows: if v was not full at the end of stage $t-1$, then $Samp_{v,t}(U_t(v))$ consists of every 4th element from $U_t(v)$; if v just became full at the end of stage $t-1$, then $Samp_{v,t}(U_t(v))$ consists of every other element from $U_t(v)$; and if v was full at the end of stage $t-2$, then $Samp_{v,t}(U_t(v)) = U_t(v)$.

The five nodes for v have the following merge equations:

$$\begin{aligned} U_{t+1}([Av]) &= Samp_{[Axv],t}(U_t([Axv])) \\ &\quad \cup Samp_{[Sv],t}\big(Prefix_{e(y)}(U_t([Sv]))\big), \end{aligned} \tag{5.1}$$

$$\begin{aligned} U_{t+1}([Lv]) &= Samp_{[Sv],t}\big(Suffix_{n_v-e(x)}(U_t([Sv]))\big) \\ &\quad \cup Samp_{[Lyv],t}(U_t([Lyv])), \end{aligned} \tag{5.2}$$

$$U_{t+1}([Axv]) = Samp_{[Ax],t}(U_t([Ax])) \tag{5.3}$$

$$U_{t+1}([Sv]) = Samp_{[Lx],t}(U_t([Lx])) \cup Samp_{[Ay],t}(U_t([Ay])), \tag{5.4}$$

$$U_{t+1}([Lyv]) = Samp_{[Ly],t}(U_t([Ly])). \tag{5.5}$$

Note that, if both children of a node are full in stage t, then that node is full in stage $t + 3$.

Comparing the above five equations to the two equations of Corollary 5.2, we have that in the stage, t, when $[Av]$ and $[Lv]$ become full, then $U_t([Av]) = A(v)$ and $U_t([Lv]) = L(v)$. Since G has twice as many levels as T, if we can perform our cascade merging procedure in G so that each stage can be implemented in $O(1)$ time, then we will have an $O(\log n)$ time algorithm.

In Atallah et al. [1989] and Cole [1988], it was shown that in a cascade merging procedure as above, but without *Prefix* and *Suffix* functions, one can maintain a *rank* label for each element e of $U_{t-1}(v)$ that gave the rank of e's predecessor in $U_t(v)$, as well as similar labels from $U_t(v)$ to the samples at v's in-nodes (i.e., its "children") in stage $t - 1$ (which were merged to form $U_t(v)$). Moreover, Atallah et al. [1989] and Cole [1988] show that these labels can be used to perform the merge at node v for stage $t + 1$ in $O(1)$ time using $O(|U_{t+1}(v)|)$ processors in the CREW PRAM model, provided the sample that came from each of v's in-nodes in stage $t - 1$ is a "good approximation" of the sample coming from that node in stage t. In particular, if e and f are elements of the sample that came from v in stage $t - 1$ such that there are k elements of this sample in the interval $[e, f)$, then there must be at most $c(k + 1)$ elements in $[e, f)$ from the sample coming from v in stage t, for some constant c (in the scheme of Atallah et al. [1989] and Cole [1988], $c = 2$). This is called the *c-cover property*.

The only difference between our merge equations and those of Atallah et al. [1989] and Cole [1988] is that in Eq. (5.1) we use the *Prefix* function and in Eq. (5.2) we use the Suffix function. Thus, had we not added the *Suffix* and *Prefix* functions, we would have satisfied the c-cover property. These functions do not upset the crucial c-cover property, however, as we see from the following observation:

Observation 5.3. Let e and f be two elements of $Samp_{[Sv],t-1}(Prefix_{e(y)}(U_{t-1}([Sv])))$ with $e < f$. If there are at most d elements of $Samp_{[Sv],t}(U_t([Sv]))$ is the interval $[e, f)$, then there are at most d elements of $Samp_{[Sv],t}(Prefix_{e(y)}(U_t([Sv])))$ in the interval $[e, f)$.

A similar observation can be made for equations involving the *Suffix* function. Thus, if a cascade merging procedure without *Prefix* and *Suffix* functions has the c-cover property, taking prefixes or suffixes before taking samples will not upset this. Note, however, that this might not be the case if we were to take prefixes or suffixes *after* taking samples. Therefore, we can implement each stage in $O(1)$ time, provided we have enough processors assigned to each active node.

To show that our method can be implemented in $O(\log n)$ time with only $O(n)$ processors, we must show that we can perform the processor allocation with only an $O(1)$-time overhead per stage. Our method is to "send" processors along with elements. Specifically, if we send m_t elements from a node w to a node v in stage t (as a part of the merge for node v), then we send $m_t - m_{t-1}$ processors to accompany them, where m_{t-1} is the number of elements we sent in the previous stage. Thus, each nonfull node v receives new processors for all the "extra" elements it receives in stage t and sends a fourth of the processor assignments it had in the previous stage. By a simple inductive argument, it is easy to see that this maintains $n_t - m_{t-1}$ processors assigned to such a v, where n_t is the size of the list stored at v at the end of stage t. For if a node v becomes full in stage $t-1$, then it sends $n_t/4 - m_{t-1}$ processors in stage t, $n_t/2 - n_t/4$ in stage $t+1$, and $n_t - n_t/2$ in stage $t+2$. Since $n_t - m_{t-1}$ is $O(n_t)$, this scheme is sufficient to solve the processor assignment for our method.

When the cascade merging procedure in G terminates, each $v \in T$ can just "read" from G its $A(v)$ and $L(v)$ lists. This does not yet give us the response to each specific *ExtractMin* in S, however. It only gives us the total set of answers. To determine the answer that is the response to each *ExtractMin*, we perform one more cascade merging procedure, this one derived from proceeding down the tree T, as follows:

Let $L'(v)$ denote the set of elements that is left over after performing the operations in $\varnothing S$ up to, but not including, the operations in $S(v)$. In other words, $L'(v)$ is the set of elements that are actually left over just before performing the operations in $S(v)$. The following lemma gives us the main idea for performing the downward sweep.

LEMMA 5.4. *Let v be a node in T with left child x and right child y. Suppose we have $L'(v)$ at v and $A(x)$ and $L(x)$ at x. Then*

$$L'(x) = L'(v),$$
$$L'(y) = \mathit{Suffix}_{m_x - e(x)}(L'(x) \cup A(x)) \cup L(x),$$

where $m_x = |L'(x) \cup A(x)|$.

PROOF. The proof that $L'(x) = L'(v)$ follows from the definition of $L'(x)$ and $L'(v)$. The proof that $L'(y) = \mathit{Suffix}_{m_x - e(x)}(L'(x) \cup A(x)) \cup L(x)$ follows from Lemma 5.1, with $S(v)$ playing the role of S in the Lemma and $L'(x)$ playing the role of L. □

We can use these definitions to define a top-down computation to construct all the possible true "left-over" sets. The response of an E operation at leaf-node v is simply the first element in the left-over set $L'(w)$ for v's parent w. This approach is not enough to give us an efficient algorithm, however. As it is expressed now, it would be impossible to construct the necessary left-over sets in $O(\log n)$ time using $O(n)$ processors. This is because for each level of the tree we would essentially be doubling the amount of space we need to represent all the left-over sets. We can get around this problem, however, by noting that for any node v we need only send its children as many left-over elements as the number of E's that are descendents of that child. That is, if x and y are the left and right children of v, respectively, then we need only send the first $e(x)$ elements of $L'(v)$ to x and only the first $e(y)$ elements of $L'(x)$ to y.

The details of the construction are as follows:

We obtain a dag G from T, as follows: Let v be a node in T with left child x and right child y. Corresponding to each such $v \in T$ are the following six nodes of G: $[L'v]$, $[L'x]$, $[Lx]$, $[Ax]$, $[\mathit{Sufx}]$, and $[L'y]$. (See Figure 5.) The idea is to define U_t lists so that, when it becomes full $U_t([L'v]) = L'(v)$, $U_t([Ax]) = A(x)$, and $U_t([Lx]) = L(x)$. For each such node v of T, the following are edges in G: $([L'v],[L'x])$, $([L'v],[\mathit{Sufx}])$, $([Ax],[\mathit{Sufx}])$, $([Lx],[L'y])$, and $([\mathit{Sufx}],[L'y])$. In addition, there is a complete binary tree that feeds into $[Ax]$

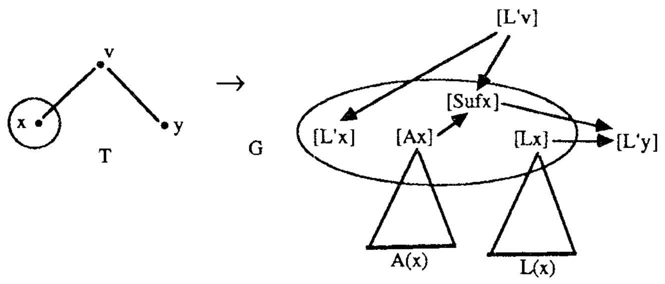

FIG. 5. The downward cascade merging procedure.

(respectively, $[Lx]$) and contains all the elements of $A(x)$ (respectively, $L(x)$) in its leaves. The flow equations in each of these two $[Ax]$ and $[Lx]$ trees are just as in the sorting algorithm of Cole [1988]. Initially, there is a complete binary tree feeding into $[L'root]$; it has n leaves, each containing $\{\infty\}$. The flow equations for the other nodes of G are as follows:

$$U_{t+1}([L'x]) = Samp_{[L'v],t}\big(Prefix_{e(x)}(U_t([L'_v]))\big)$$

$$U_{t+1}([Sufx]) = Samp_{[Ax],t}(U_t([Ax])) \cup Samp_{[L'v],t}\big(Prefix_{e(y)}(U_t([L'_v]))\big)$$

$$U_{t+1}([L'y]) = Samp_{[Lx],t}(U_t([Lx])) \cup Samp_{[Sufx],t}\big(Suffix_{e(y)}(U_t([Sufx]))\big).$$

The reader should note that these flow equations satisfy the constraints determined by Lemma 5.4. Also recall that the *Samp* functions are synchronized so that a node becomes full three stages after both of its children become full.

It is not hard to show that the graph G that results from this construction contains $O(n)$ nodes and has $O(\log n)$ height. As with the first pass, the *Prefix* and *Suffix* functions do not upset the c-cover property. Moreover, even though each node $[L'v]$ has out-degree 2, the number of elements that we send from $[L'v]$, when $[L'v]$ is full, does not exceed the total number of elements stored in $U_t([L'v])$. Thus, the cascading flow problem can be solved for G in $O(\log n)$ time using $O(n)$ processors. This, in turn, gives us a solution to the sequence evaluation problem that runs in these bounds, because for each leaf node v associated with an *ExtractMin* operation, we can simply examine the $L'(w)$ list for v's parent w to determine the response for this *ExtractMin*. Thus, we have the following theorem:

THEOREM 5.5. *Given a sequence S of Insert(x) and ExtractMin operations, one can evaluate $\varnothing S$ in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model, which is optimal.*

In the next section, we study a generalization to the *ExtractMin* operation that can be used to parallelize certain types of "lexicographic" sequential algorithms.

6. *The Off-Line Barrier-ExtractMin Problem*

Let the operation *ExtractMin*(y) ($E(y)$ for short) return and simultaneously remove from the set the smallest element $\geq y$ (if there are many copies of it then, by convention, the one inserted latest gets removed). This section concerns itself with the case where the operations appearing in S are $I(x)$ and $E(y)$. Before we give our method for evaluating $\emptyset S$, let us give an application of this sequence-evaluation problem to an important matching problem, so as to motivate our study of the $E(y)$ operation.

6.1. APPLICATION: MAXIMUM MATCHING IN A CONVEX BIPARTITE GRAPH. One additional problem that can be formulated as an off-line sequence of set manipulation operations is that of computing a maximum matching in a convex bipartite graph. An $O(\log^2 n)$-time algorithm for solving this problem on an EREW PRAM model was given by Dekel and Sahni [1984]. In this section, we show how to formulate this problem as the evaluation of a sequence of $I(x)$ and $E(y)$ operations. This reduction can be implemented in $O(\log n)$ time.

First, recall that a convex bipartite graph is such that its vertex set can be written as $A \cup B$ where $A = \{a_1, \dots, a_p\}$ and $B = \{b_1, \dots, b_q\}$, where (i) every edge has one endpoint in A and the other endpoint in B, and (ii) of (a_i, b_j) and (a_i, b_{j+k}) are edges, then so is (a_i, b_{j+s}) for every $1 \leq s < k$. Let $l_i(r_i)$ be the smallest (largest) j such that (a_i, b_j) is an edge. Glover's algorithm [1967] for finding a maximum matching in such a graph works as follows: Consider the vertices of B one by one, starting at b_1. When b_j is considered, match it against a remaining a_k that is adjacent to it and whose r_k is smallest, and then delete a_k from the graph. Glover's algorithm can be formulated as a sequence of $I(x)$ and $E(y)$ operations, as follows:

Without loss of generality, we assume that the a_i's are renamed so that $r_1 \leq \cdots \leq r_p$. Let L_j (R_j) denote the set that contains every a_i whose l_i (r_i) equals b_j. Then Glover's algorithm is equivalent to the problem of evaluating the sequence S created by considering the vertices of B one by one, starting at b_1 with $S = \emptyset$ and $\beta = -\infty$. When b_j is considered, we append to the end of S an $I(a_i)$ for every $a_i \in L_j$, followed by an $E(\beta)$. Then (before moving to b_{j+1}), we set β equal to the max of its old value and the largest element in R_j. If, in S, the response to the jth $E(y)$ is a_i, then the edge (a_i, b_j) is in the maximum matching. It is easy to prove that this procedure results in exactly the same matching as Glover's algorithm. We can construct the list of a_i's by sorting [Cole 1988] and then construct all the corresponding β values by a parallel prefix computation [Kruskal et al. 1985; Ladner and Fischer 1980]. Thus, we have the following:

THEOREM 6.1.1. *The maximum matching problem for convex bipartite graphs can be reduced to the problem of evaluating $\emptyset S$ in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model, where S contains $I(x)$ and $E(y)$ operations, and where the arguments to the $E(y)$ operations are nondecreasing.*

In the next subsection, we show that the problem of evaluating a sequence of $I(x)$ and $E(y)$ operations is in the class NC^2. In the subsequent subsection, using a completely different technique, we show that if the arguments to the $E(y)$ operations are nondecreasing then the evaluation problem is in NC^1. Thus, as a simple corollary, we get that the maximum matching problem for convex bipartite graphs is in NC^1.

6.2. THE GENERAL OFF-LINE BARRIER-EXTRACTMIN PROBLEM. In this subsection, we show how to evaluate a sequence S of *Insert*(x) and *ExtractMin*(y)

operations in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors in the CREW PRAM model. For expository reasons, we first concern ourselves with proving membership in NC by giving a rather inefficient algorithm that runs in $O(\log^2 n)$ time with $O(n^5)$ processors. The next lemma reduces the problem to that of determining which $E(y)$'s have an empty response.

LEMMA 6.2.1. *Let S be a sequence of n $I(x)$ and $E(y)$ operations. Let O be any one of the $E(y)$ operations in S, and let $TEST(S,O)$ be any algorithm that solves the problem of determining whether O has an empty response in $\varnothing S$. Let $T(n)$ and $P(n)$ be the time and processor complexities of $TEST(S,O)$. Then determining the actual responses to all the $E(y)$ operations in $\varnothing S$ can be done in time $O(T(n) + \log n)$ with $O(n^2P(n))$ processors.*

PROOF. To every operation O that is an $E(y)$, assign $P(n)$ processors that perform $TEST(S,O)$ to determine whether it has a nonempty response in $\varnothing S$. If $TEST(S,O)$ determines that the response to O in $\varnothing S$ is empty, then that is the correct response for O. However, if $TEST(S,O)$ determines that O has a nonempty response in $\varnothing S$, then O gets assigned $nP(n)$ processors whose task it will be to determine the actual response of O. We now show how these $nP(n)$ processors can find the (nonempty) response of such an O in time $O(T(n) + \log n)$. We need only consider the prefix of S that ends with O, i.e., if $S = O_1O_2 \cdots O_n$ and $O = O_j = E(y)$ then we need only look at $\varnothing S_j$ where S_j is $O_1O_2 \cdots O_j$. Let $(x_1, x_2, \dots, x_q)$ be the elements inserted in S_j that are $\geq x$, sorted from worst to best (and hence $x_1 \geq x_2 \geq \cdots \geq x_q \geq y$). In other words, if there are, in S_j, q insertions of elements $\geq y$, then the sequence $(x_1, x_2, \dots, x_q)$ is the sorted version of

$$\{x : I(x) \in S_j \text{ and } x \geq y\}$$

One of these x_i's is the correct response to O. To determine which one it is, we create q subproblems where the kth subproblem is that of determining whether O has a nonempty response in

$$\varnothing O_1 \cdots O_{j-1} E(x_1) \cdots E(x_k) O_j, \tag{6.1}$$

i.e., the kth subproblem is obtained by putting just before O_j in $\varnothing S_j$ the sequence $E(x_1)E(x_2) \cdots E(x_k)$. Each such kth subproblem is solved in $T(n)$ time with $P(n)$ processors using the $TEST$ procedure (there are enough processors for this because O_j has $nP(n)$ processors assigned to it). We claim that the response of O_j in $\varnothing S_j$ is then x_s, where s is the maximum k such that the response of O_j in the kth subproblem is not empty. We now show that x_s is indeed the response of O_j in $\varnothing S_j$. Let r_k be the response to O_j in the kth subproblem (possibly r_k is an empty response, i.e., $r_k =$ "set empty"). Observe that the sequence $r_1, r_2, \dots, r_q$ is initially monotonically decreasing, then at some threshold index, consists of "set empty" responses (this monotonicity follows from the way that q subproblems are defined). Let x_t be the response to O_j in $\varnothing S_j$. Then surely the response to O_j is still x_t in every kth subproblem for which $k < t$ (because the $kE(y)$ operations just before O_j in that subproblem remove elements about which O_j "doesn't care" because they are worse than its own response x_t). On the other hand, if $k \geq t$, then surely the response to O_j in the kth subproblem is empty, because otherwise that response is better than x_t, a contradiction (the response to O_j in any kth subproblem cannot be better than its response in $\emptyset S_j$). Therefore, $t = s$, completing the proof (the additive $\log n$ term in the time complexity comes from the max operation needed for computing s). $\square$

Next, we focus on describing a procedure *TEST*(S, O) that has a $T(n) = O(\log^2 n)$ and a $P(n) = O(n^3/\log n)$.

This will imply a weaker version of Theorem 6.4, one with $O(n^5/\log n)$ processors. We then show how to bring down the processor complexity to $O(n^3/\log n)$ by exploiting similarities between the n^2 copies of the *TEST*-ing problem that are created.

Without loss of generality, we may describe *TEST*(S, O) assuming that O is the last operation in S, that is, $S = O_1 O_2 \cdots O_n$ where $O = O_n$. We begin with the observation that solving *TEST*(S, O) amounts to determining the cardinality of a maximum *up-left matching* problem [Leighton and Shor 1986]. Create n distinct points in the plane, as follows: for every operation O_i in S, create a corresponding planar point whose x-coordinate is i and whose y-coordinate is the parameter of O_i (i.e., z if $O_i = I(z)$ or $O_i = E(z)$). The points corresponding to $E(z)$'s are called *plusses*, those corresponding to $I(z)$'s are called *minuses*. The responses to the $E(z)$'s in $\emptyset S$ can be viewed as being the result of the following matching procedure: Scan the plusses in left to right order (i.e., by increasing x coordinates), matching the currently scanned plus with the lowest unmatched minus that is to the left of it and not below it. The correspondence between the matching so produced and the responses in $\emptyset S$ should be obvious: a plus at (i, a) is matched with a minus at (j, b), $j < i$ and $a \leq b$, if and only if the $E(a)$ corresponding to the plus has as its response in $\emptyset S$ the element b inserted by the $I(b)$ corresponding to the minus. Furthermore, one can show [Leighton and Shor 1986] that this greedy left-to-right matching procedure produces a matching of maximum cardinality among all possible up-left matchings (up-left matchings are ones in which a plus can be matched with a minus only if that minus is to its left and not below it). These remarks imply that in order to determine whether O has a response in $\emptyset S$, it suffices to compare the cardinality c of a maximum matching for the configuration of plusses and minuses corresponding to S, with the cardinality c' of a maximum matching for the configuration of plusses and minuses corresponding to $S - O = O_1 O_2 \cdots O_{n-1}$. If $c = c'$, then the presence of O does not make a difference and hence its response in $\emptyset S$ is empty, while $c = c' + 1$ implies that it has a nonempty response.

This reduces the problem of designing *TEST*(S, O) to that of designing a procedure for computing the *size* of the maximum cardinality up-left matching of a configuration of n plusses and minuses. We now give a sketch of such a procedure.

If $p = (a, b)$ is a plus, then *Region*(p) is the region $(-\infty, a] \times [b, +\infty)$, that is, the closure of the region of the plane that is to the left of p and above it. (See Figure 6.)

If P is a set of plusses, then *Region*$(P) = \bigcup_{p \in P}$ *Region*(p).

The *deficiency* of any region of the plane is the number of plusses in it minus the number of minuses in it. The deficiency of a set of plusses P is that of *Region*(P) and it denoted by *def*(P).

For example, in Figure 7, *def*$(\{u, v, w\}) = 5 - 1 = 4$.

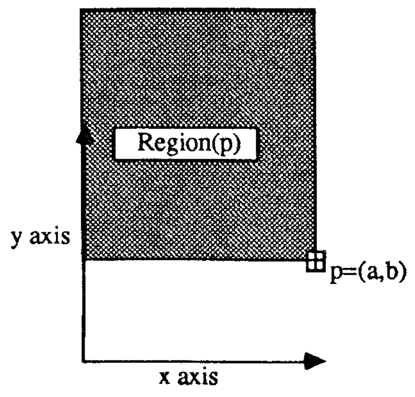

FIG. 6. Illustrating *Region*(p).

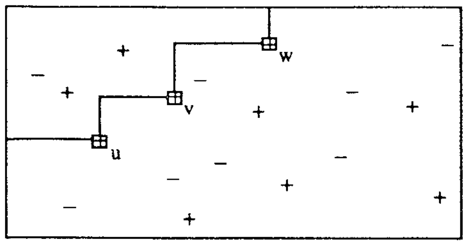

FIG. 7. $\max\{def(P) : P \subseteq \Pi\} = 4$ and occurs for $P = \{u, v, w\}$.

LEMMA 6.2.2 [LEIGHTON AND SHOR 1986]. *Let Π denote the set of plusses. The cardinality of a maximum up-left matching is then equal to*

$$|\Pi| - \max\{def(P) : P \subseteq \Pi\}.$$

PROOF. A straightforward application of Hall's Theorem (see Leighton and Shor [1986] for details). □

The above lemma implies that one can compute $TEST(S, O)$ in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors provided we can compute the quantity $\max\{def(P) : P \subseteq \Pi\}$ within those same bounds. This is what we show how to do next.

Let $G(S)$ be the weighted dag whose vertex set is the set of plusses and two new special vertices s and t, and whose edge set is defined as follows: For every two vertices p and q, there is an arc from p to q if and only if one of the following conditions (i)–(iii) holds:

(i) $p = s$ and $q \neq t$.
(ii) $p \neq s$ and $q = t$.
(iii) p is a point (a, b) and q a point (c, d) such that $a \leq c$ and $b \leq d$ (i.e., q is to the right and above p).

In case (i), the cost of the arc (s, q) is equal to $def(q)$. In case (ii), the cost of the arc (p, t) is zero. In case (iii), the cost of the arc (p, q) is the deficiency of the region $[a, c] \times [d, +\infty)$.

For the situation shown in Figure 7, the cost of arc (s, u) is $2 - 1 = 1$, that of (u, v) is 2, that of (v, w) is 1, and that of (w, t) is 0 (s and t are fictitious vertices to which no points correspond in the figure).

It is not hard to see that the cost of a longest s-to-t path in $G(S)$ is precisely equal to the quantity $\max\{def(P) : P \subseteq \Pi\}$. Since $G(S)$ is acyclic, computing its longest s-to-t path is trivial to do in $O(\log^2 n)$ time with $O(n^3/\log n)$ processors.

The above $O(\log^2 n)$ time, $O(n^3/\log n)$ processor algorithm for $TEST(S, O)$ immediately implies (by Lemma 6.2) an $O(\log^2 n)$ time, $O(n^5/\log n)$ processor algorithm for evaluating sequence S.

However, this is extremely inefficient: We would be creating all n^2 instances of the *TEST*-ing problem suggested by the proof of Lemma 6.2, that is, all n^2 graphs $G(S)$, one for each S of the form 6.1 (in the proof of Lemma 6.2). Instead, we save a factor of n^2 in the processor complexity as follows:

Step 1. We create a graph $G(S)$: the one for S equal to the original sequence of n operations.

Step 2. We solve the all-pairs longest paths problem on the $G(S)$ created in Step 1, obtaining an all-pairs longest paths matrix M. This is trivial to do in time $O(\log^2 n)$ and with $O(n^3/\log n)$ processors.

Step 3. We partition our $n^3/\log n$ processors into n groups of $n^2/\log n$ processors each, and assign one group to each $E(y)$ in the original (input) sequence S. We now describe the algorithm performed by one typical such group, say, the group assigned to O_j. The task this group of $n^2/\log n$ processors faces is to use the matrix M computed in Step 2 to determine the response of O_j in $\emptyset S$. Refer to 6.1, in the proof of Lemma 6.2.1, and recall that the response of O_j is one of $x_1, \ldots, x_q$. To determine which one it is, we already know that it suffices to compute the length of a longest s-to-t path in each of the $q + 1$ graphs $G_1, \ldots, G_{q+1}$, where

$$G_k = G\big(O_1 \cdots O_{j-1} E(x_1) \cdots E(x_k)\big),$$

using the notational convention $E(y_{q+1}) = O_j(= E(y))$. We therefore need only concern ourselves with the problem of computing the lengths of these s-to-t paths. Observe that no path can go through more than one of the $q + 1$ plusses corresponding to $\{E(y_1), \ldots, E(y_{q+1})\}$ (because $y_1 \geq \cdots \geq y_q \geq y$). Let $Plus(E)$ denote the plus corresponding to E. The length (call it $Best(k, l)$) of a longest s-to-t path in G_k that goes through $Plus(E(y_l))$ $(l \leq k)$ is equal to the maximum, over all $i \in \{1, \ldots, j - 1\}$ for which O_i is an E, of the quantity

$$M(s, Plus(O_i)) + \text{the cost of the } Plus(O_i)\text{-to-}Plus(E(y_l)) \text{ arc in } G_k.$$

We use the $n^2/\log n$ processors available to compute $Best(k, l)$ for all pairs k, l in $O(\log n)$ time. Then we use $n/\log n$ processors for each k to compute, in $O(\log n)$ time, the length of a longest s-to-t path in G_k, which is equal to

$$\max_{1 \leq l \leq k} Best(k, l).$$

The time and processor complexities of the above algorithm are clearly dominated by those needed for the all-pairs longest paths computation of Step 2. This establishes the following theorem:

THEOREM 6.2.3. *Given a sequence S of n $I(x)$ and $E(y)$ operations, one can evaluate $\emptyset S$ in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors in the CREW PRAM model.*

In the next subsection, we study an important special case of this evaluation problem.

6.3. A SPECIAL CASE OF THE OFF-LINE BARRIER-EXTRACTMIN PROBLEM. The main result of this subsection is an NC^1 algorithm for the special case of evaluating $\varnothing S$ when S contains $I(x)$ and $E(y)$ operations, where the $E(y)$ operations in S are such that the sequence of y's is in nondecreasing order. As a consequence of this result, we can obtain an NC^1 algorithm for finding a maximum matching in a convex bipartite graph, a time improvement by a factor of $\log n$ over the previous fastest parallel algorithm for this problem, by Dekel and Sahni [1984].

Let m denote the number of $E(y)$ operations in S, and let $E(y_i)$ denote the ith such operation. Note that, by hypothesis, we have $y_1 \leq y_2 \leq \cdots \leq y_m$. Let A_i denote the set of elements inserted by the $I(x)$ operations between $E(y_{i-1})$ and $E(y_i)$, so that the sequence S can be written $S = A_1 E(y_1) A_2 E(y_2) \cdots A_m E(y_m)$ (some of the A_i's may be empty). Without loss of generality, we assume that no A_i contains an element less than y_i (such an element would be useless anyway).

The longest-paths characterization of the previous section apparently does not result in an $O(\log n)$ time algorithm: that $y_1 \leq y_2 \leq \cdots \leq y_m$ implies that the plusses form an increasing chain, but this in itself does not give an $O(\log n)$-time algorithm for the resulting longest-path problem. Our solution actually avoids the characterization of the previous section. Instead, we replace the problem with a polynomial number of subproblems each of which is such that the first $E(y)$ occurs after the last $I(x)$. The next lemma observes that this type of problem is solvable in $O(\log n)$ time.

LEMMA 6.3.1. *If S is of the form $AE(y_1)E(y_2) \cdots E(y_m)$, then all the responses can be computed in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model.*

PROOF. Let $L(i, k)$, $1 \leq i \leq m$, $0 \leq k \leq |A|$, denote the set consisting of the largest k elements of A. Note that $L(0, |A|) = A$. Let the *successor* of $L(i, k)$ be the set obtained by removing all the elements that are less than y_i from the set resulting from $L(i, k)E(y_i)$. It is easy to see that the successor of $L(i, k)$ is equal to $L(i + 1, p)$ for some integer $p < k$, since all the $E(y_i)$'s come after A and the y_i's are monotonically nondecreasing. An $L(i, k)$ with $i < m$ has exactly one successor and hence the successor function defines a tree whose $O(n^2)$ nodes are the $L(i, k)$'s and such that the parent of $L(i, k)$ is its successor. The root of this tree is $L(m, q)$ for some integer q. The successor function is easily computed, since the successor of $L(i, k)$ is $L(i + 1, k - 1)$ if $k \leq n_i$ and is $L(i + 1, n_i)$ otherwise, where n_i is the number of elements in A greater than or equal to y_i. In the tree defined by the successor function, consider the path originating from the leaf $L(0, |A|)$ and terminating at the root $L(m, q)$. This path constitutes a complete description of the responses to the $E(y_i)$'s, as follows: If $L(i, k)$ is on this path and $k > 0$, then the response to $E(y_i)$ is the smallest element in $L(i, k)$. If $L(i, k)$ is on this path and $k = 0$, then $E(y_i)$ has a "set empty" response. Tracing this path is trivial to do in time $O(\log n)$ with $O(n^2)$ processors. We can achieve $O(\log n)$ time using only $O(n)$ processors, however. The method is very similar to that used in Subsection 3.2. In this case, however, one merges singleton sets instead of lists, so that the time is $O(\log n)$ instead of $O(\log n \log\log n)$. This is because for any collection of columns $i, i+1, \dots, j$ there is only one critical rank; namely, the rank that has $L(j, n_j)$ as its successor. $\square$

We now show how to solve the problem for $S = A_1E(y_1)A_2E(y_2)\cdots A_mE(y_m)$ by solving a polynomial number of problems each of which is of the type considered in Lemma 6.3.1.

Notation. Let $A_{ij} = A_i \cup A_{i+1} \cup \cdots \cup A_j$. Let r_{ij} be the response to $E(y_j)$ in $A_{ij}E(y_i)E(y_{i+1})\cdots E(y_j)$. Let z_{ij} be the response to $E(y_j)$ in $A_iE(y_i)A_{i+1}E(y_{i+1})\cdots A_jE(y_j)$.

Note that in this notation the response to $E(y_j)$ in S is z_{1j}. Also note that the r_{ij}'s can be computed in $O(\log n)$ time because of Lemma 6.3.1. The following theorem establishes a crucial link between the r_{ij}'s and the z_{1j}'s and implies that the z_{1j}'s can also be computed in $O(\log n)$ time:

LEMMA 6.3.2. *For every j, $1 \le j \le m$, $z_{1j} = min_{1 \le i \le j} r_{ij}$.*

PROOF. The proof is in two steps (claims 1 and 2).

CLAIM 1. $z_{1j} \le min_{1\le i\le j} r_{ij}$.

PROOF OF CLAIM 1. First, note that $z_{1j} \le z_{ij}$ for every $i \le j$. Hence, it suffices to prove that $z_{ij} \le r_{ij}$ for every $i \le j$. We prove this by induction on $j - i$, the basis $(j = i)$ being trivial. For the inductive step, we distinguish two cases.

Case 1. In $A_{ij}E(y_i)E(y_{i+1})\cdots E(y_j)$, no element of A_i gets extracted. In this case, we have

$$\begin{aligned} r_{ij} &= \text{the response to } E(y_j) \text{ in } A_{i+1,j}E(y_i)E(y_{i+1})\cdots E(y_j) \\ &\ge \text{the response to } E(y_j) \text{ in } A_{i+1,j}E(y_{i+1})E(y_{i+2})\cdots E(y_j). \qquad (6.2) \end{aligned}$$

Let $\hat{A}_i$ be obtained from A_i by removing from it the smallest element, and all the elements $< y_{i+1}$. The definition of z_{ij} implies:

$$\begin{aligned} z_{ij} &= \text{the response to } E(y_j) \\ &\quad \text{in } \left(A_{i+1} \cup \hat{A}_i\right)E(y_{i+1})A_{i+2}E(y_{i+2})\cdots A_jE(y_j) \\ &\le \text{the response to } E(y_j) \text{ in } A_{i+1}E(y_{i+1})A_{i+2}E(y_{i+2})\cdots A_jE(y_j), \end{aligned}$$

which, using the induction hypothesis, gives us the following:

$$z_{ij} \le \text{the response to } E(y_j) \text{ in } A_{i+1,j}E(y_{i+1})E(y_{i+2})\cdots E(y_j).$$

This and (6.2) imply that $z_{ij} \le r_{ij}$.

Case 2. In $A_{ij}E(y_i)E(y_{i+1})\cdots E(y_j)$, at least one element of A_i gets extracted. Since $y_1 \le y_2 \le \cdots \le y_m$, the smallest element in A_i gets extracted. Let $\hat{A}_i$ be obtained from A_i by removing from it the smallest element, and all the elements $< y_{i+1}$. The definition of r_{ij} implies

$$r_{ij} = \text{the response to } E(y_j) \text{ in } \left(A_{i+1,j} \cup \hat{A}_i\right)E(y_{i+1})E(y_{i+2})\cdots E(y_j). \quad (6.3)$$

The definition of z_{ij} implies

$$\begin{aligned} z_{ij} &= \text{the response to } E(y_j) \\ &\quad \text{in } \left(A_{i+1} \cup \hat{A}_i\right) E(y_{i+1}) A_{i+2} E(y_{i+2}) \cdots A_j E(y_j), \end{aligned}$$

which, using the induction hypothesis, in turn, implies

$$\begin{aligned} z_{ij} &\le \text{the response to } E(y_j) \text{ in } \left(A_{i+1,j} \cup \hat{A}_i\right) E(y_{i+1}) E(y_{i+2}) \cdots E(y_j) \\ &= r_{ij}, \end{aligned}$$

where (6.3) was used. This completes the proof of Claim 1.

CLAIM 2. $z_{1j} \ge min_{1 \le i \le j} r_{ij}$.

PROOF OF CLAIM 2. We prove, by induction on j, that for every j there is an $i \le j$ such that $z_{1j} \ge r_{ij}$. The basis ($j = 1$) holds trivially. For the inductive step, we again distinguish two cases.

Case 1. $z_{11} > z_{1j}$. Let $\hat{A}_2 = A_2 \cup A_1 - \{z_{11}\}$, and let $\hat{A}_i = A_i$ if $2 < i \le j$. Then we have that

$$z_{1j} = \text{the response to } E(y_j) \text{ in } \hat{A}_2 E(y_2) \hat{A}_3 E(y_3) \hat{A}_4 \cdots \hat{A}_j E(y_j).$$

By the induction hypothesis, there is an i, $2 \le i \le j$, such that

$$\begin{aligned} z_{1j} &\ge \text{the response to } E(y_j) \\ &\quad \text{in } \left(\hat{A}_i \cup \hat{A}_{i+1} \cup \cdots \cup \hat{A}_j\right) E(y_i) E(y_{i+1}) \cdots E(y_j). \end{aligned} \tag{6.4}$$

If $i \ge 3$, then the right-hand side of (6.4) is r_{ij}, and hence $z_{1j} \ge r_{ij}$. If $i = 2$, then

$$\begin{aligned} z_{1j} &\ge \text{the response to } E(y_j) \text{ in} \\ &\quad \left((A_1 - \{z_{11}\}) \cup A_2 \cup \cdots \cup A_j\right) E(y_2) E(y_3) \cdots E(y_j). \end{aligned} \tag{6.5}$$

Since $z_{11} > z_{1j}$ and z_{11} is the smallest element of A_1, all the elements of $A_1 - \{z_{11}\}$ are larger than z_{1j} and hence, by (6.5), larger than the right-hand side of (6.5). Consequently, the right-hand side of (6.5) is the same as the response to $E(y_j)$ in $A_{2j} E(y_2) E(y_3) \cdots E(y_j)$, that is, r_{2j}.

Case 2. $z_{11} \le z_{1j}$. Let $\hat{A}_2 = A_1 \cup A_2 - \{z_{11}\} - \{\text{all elements} < y_2\}$, and let $\hat{A}_i = A_i$ if $2 < i \le j$. Then we have the following:

$$z_{1j} = \text{the response to } E(y_j) \text{ in } \hat{A}_2 E(y_2) \hat{A}_3 E(y_3) \hat{A}_4 \cdots \hat{A}_j E(y_j).$$

By the induction hypothesis, there is an i, $2 \le i \le j$, such that

$$\begin{aligned} z_{1j} &\ge \text{the response to } E(y_j) \\ &\quad \text{in } \left(\hat{A}_i \cup \hat{A}_{i+1} \cup \cdots \cup \hat{A}_j\right) E(y_i) E(y_{i+1}) \cdots E(y_j). \end{aligned} \tag{6.6}$$

If $i \geq 3$, then the right-hand side of (6.6) is r_{ij}, and hence $z_{1j} \geq r_{ij}$. If $i = 2$, then

$$z_{1j} \geq \text{the response to } E(y_j)$$
$$\text{in } \left(\hat{A}_2 \cup A_3 \cup \cdots \cup A_j\right) E(y_2) E(y_3) \cdots E(y_j). \tag{6.7}$$

From (6.7), and the fact that any element in $A_1 \cup A_2 - \hat{A}_2$ is $\leq z_{1j}$, it follows that

$$\begin{aligned} z_{1j} &\geq \text{the response to } E(y_j) \text{ in } \left(A_1 \cup A_2 - \hat{A}_2\right) \\ &\quad \cup \left(\hat{A}_2 \cup A_3 \cup \cdots \cup A_j\right) E(y_2) E(y_3) \cdots E(y_j) \\ &= \text{the response to } E(y_j) \\ &\quad \text{in } (A_1 \cup A_2 \cup A_3 \cup \cdots \cup A_j) E(y_2) E(y_3) \cdots E(y_j) \\ &= r_{1j}. \end{aligned}$$

This completes the proof of Claim 2, and hence of Lemma 6.6. □

We are now ready to state the main result of this subsection.

THEOREM 6.3.3. *Given a sequence* $S = A_1 E(y_1) A_2 E(y_2) \cdots A_m E(y_m)$ *where* $y_1 \leq y_2 \leq \cdots \leq y_m$, *one can evaluate* $\emptyset S$ *in* $O(\log n)$ *time using* $O(n^3)$ *processors in the EREW PRAM model.*

PROOF. Assign n processors to every pair i and j, $i \leq j$, and use them to compute r_{ij} in $O(\log n)$ time. Then, assign $n/\log n$ processors to every $E(y_i)$ and use them to compute $z_{1j} = \min_{1 \leq i \leq j} r_{ij}$. The overall time complexity is clearly $O(\log n)$ using $O(n^3)$ processors. □

COROLLARY 6.3.4. *The problem of computing a maximum matching for a convex bipartite graph is in* NC^1.

PROOF. An immediate consequence of Theorems 6.1.1 and 6.3.3. □

7. *Final Remarks*

The problem of efficiently evaluating an off-line sequence of data structure operations has been extensively studied for sequential models of computation. However, surprisingly little work had previously been done on the parallel complexity of such problems. This paper provides a first step in the study of the parallel complexity of these problems. Here we focussed primarily on problems whose membership in *NC* was nonobvious, due to the behavior of *ExtractMin* and *ExtractMin*(y) operations. The main open question that remains is whether the problem is in *NC* when S contains $I(x)$, $D(x)$ and $E(y)$ operations.

REFERENCES

ABRAHAMSON, K., DADOUN, N., KIRKPATRICK, D. G., AND PRZYTYCKA, T. 1989. A simple parallel tree contraction algorithm. *J. Algorithms 10*, 287–302.

AHO, A. V., HOPCRAFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

AJTAI, M., KOMLOS, J., AND SZEMEREDI, E. Sorting in $c \log n$ parallel steps. *Combinatorica 3*, 1–19.

ATALLAH, M. J., COLE, R., AND GOODRICH, M. T. 1989. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput. 18*, 3, 499–532.

BILARDI, G., AND NICOLAU, A. 1989. Adaptive Bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM J. Comput. 18*, 2, 216–228.

BORODIN, A., AND HOPCRAFT, J. E. 1985. Routing, merging, and sorting on parallel models of computation. *J. Comput. Syst. Sci. 30*, 1, 130–145.

CHAZELLE, B. M. 1985. How to search in history. *Inf. Control 64*, 77–79.

COLE, R. 1986. Searching and storing similar lists. *J. Algorithms 7*, 202–229.

COLE, R. 1988. Parallel merge sort. *SIAM J. Comput. 17*, 4, 770–785.

DEKEL, E., AND SAHNI, S. 1983. Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput.*, 307–315.

DEKEL, E., AND SAHNI, S. 1984. A parallel matching algorithm for convex bipartite graphs and applications to scheduling, *J. Para. Distr. Comput.*, 185–205.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., AND TARJAN, R. E. 1989. Making data structures persistent. *J. Comput. Syst. Sci. 38*, 1, 86–124.

DYMON, P. W., AND COOK, S. A. 1980. Hardware complexity and parallel computer. In *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 360–372.

EDELSBRUNNER, H., AND OVERMARS, M. H. 1985. Batched dynamic solutions to decomposable searching problems. *J. Algorithms 6*, 515–542.

EL GINDY, H., AND GOODRICH, M. T. 1988. Parallel algorithms for shortest path problems in polygons. *The Visual Computer: Int. J. Comput. Graph. 3*, 6, 371–378.

GABOW, H. N., AND TARJAN, R. E. 1983. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, pp. 246–251.

GLOVER, F., 1967. Maximum matching in a convex bipartite graph. *Naval Res. Log. Q.*, 313–316.

GOODRICH, M. T. 1987. Efficient parallel techniques for computational geometry. Ph.D. dissertation. Dept. of Comput. Sci., Purdue Univ., West Lafayette, Ind.

GOODRICH, M T. 1991. Intersecting line segments in parallel with an output-sensitive number of processors *SIAM J. Comput. 20*, 4, 737–755.

GOODRICH, M. T., GHOUSE, M., AND BRIGHT, J. 1990. Generalized sweep methods for parallel computational geometry. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece, July 2–6). ACM, New York, pp. 280–289.

HORN, W. A. 1974. Some simple scheduling algorithms. *Nav. Res. Log. Q. 21*, 177–85.

KOSARAJU, S. R., AND DELCHER, A. L. 1988. Optimal parallel evaluation of tree-structured computations by raking. In *AWOC88*. Lecture Notes in Computer Science. vol. 319. Springer-Verlag, New York, pp. 101–110.

KRUSKAL, C. P. 1983. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput. C-32*, 10, 942–946.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. 1985. The power of parallel prefix. In *Proceedings of the 1985 International Conference on Parallel Processing*. IEEE, New York, pp. 180–185.

LADNER, R. E., AND FISCHER, M. J. 1980. Parallel prefix computation. *J. ACM 27*, 4, 831–838.

LEIGHTON, T., AND SHOR, P. 1986. Tight bounds for minimax grid matching, with applications to the average case analysis of algorithms. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing* (Berkeley, Calif., May 28–30). ACM, New York, pp. 91–103.

LIPSKI, W., AND PREPARATA, F. P. 1981. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Inf.*, 329–346.

MILLER, G. L., AND REIF, J. H. 1985. Parallel tree contraction and its application. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 478–489.

PAUL, W., VISHKIN, U., AND WAGENER, H. 1983. Parallel computation on a 2–3 tree. Tech Rep. 70. Courant Inst., New York Univ., New York.

RODGER, S. H. 1988. An optimal parallel algorithm for preemptive job scheduling that minimizes maximum lateness. In *Proceedings of the 26th Allerton Conference on Communications, Control, and Computers* (Allerton, Ill.). Univ. Illinois, pp. 293–302.

RODGER, S. H. 1989. Parallel scheduling algorithms. Ph.D. dissertation, Purdue Univ., West Lafayette, Ind.

RUZZO, W. L. 1981. On uniform circuit complexity. *J. Comput. Syst. Sci. 22*, 3, 365–383.

VALIANT, L. G. 1975. Parallelism in comparison problems. *SIAM J. Comput. 4*, 3, 348–355.

RECEIVED SEPTEMBER 1990, REVISED JULY 1993; ACCEPTED JULY 1993