Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation

Michael T. Goodrich and S. Rao Kosaraju

Johns Hopkins University, Baltimore, Maryland

Abstract

We present optimal algorithms for sorting on parallel CREW and EREW versions of the pointer machine model. Intuitively, one can view our methods as being based on a parallel mergesort using linked lists rather than arrays (the usual parallel data structure). We also show how to exploit the "locality" of our approach to solve the set expression evaluation problem, a problem with applications to database querying and logic-programming, in $O(\log n)$ time using $O(n)$ processors. Interestingly, this is an asymptotic improvement over what seems possible using previous techniques.

Categories and Subject Descriptors: E.1 [**Data Structures**]: *arrays, lists*; F.2.2. [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: parallel algorithms, PRAM, pointer machine, linking automaton, expression evaluation, mergesort, cascade merging

1 Introduction

One of the primary models of parallel computation is the Parallel Random-Access Machine (or PRAM) model (e.g., see JáJá [15], Karp and Ramachandran [16], and

A preliminary announcement of this research appeared in *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989, 190–195. Professor Goodrich's research was supported by the National Science Foundation under Grants CCR-8810568, CCR-8908092, CCR-9003299, IRI-9116843, and CCR-9300079. Professor Kosaraju's research was supported by the National Science Foundation under Grants CCR-8804284, CCR-8908092, CCR-9107293, and CCR-9508545. Authors' address: Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218; e-mail: goodrich@cs.jhu.edu, kosaraju@cs.jhu.edu.

Reif [14]), where one has a collection of synchronized processors that access a common memory. Each processors may use any of the addressing schemes allowed by the sequential RAM model (see Cormen *et al.* [11]), such as indexed and indirect addressing, to access the memory cells, so long as such an access does not violate any concurrent-access constraints that may have been placed on the common memory. The three principal examples of such constraints include the EREW constraint, where one only allows exclusive reads and exclusive writes in any step; the CREW constraint, where one allows many processors to concurrently read from the same memory cell, but writes must be exclusive; and the CRCW constraint, where one allows concurrent reads and concurrent writes (assuming a suitable conflict resolution mechanism for the writes).

Just as the PRAM model is a generalization of the sequential RAM model, the *Parallel Pointer Machine* (or PPM) model is a generalization of the sequential Pointer Machine model described by Chazelle [8], which is similar to the Linking Automaton model of Knuth [17] and Tarjan [26]. In the PPM, one has a collection of synchronized processors that may access a common memory, just as in the PRAM model. In this model, however, the types of memory accesses are limited to those allowed by the Pointer Machine. In particular, the common storage is modeled as a directed graph, whose vertices correspond to memory cells, where each cell has $O(1)$ *value* fields. The edges of this graph correspond to pointers, and each cell has $O(1)$ *pointer* fields. A processor accesses this storage via $O(1)$ *pointers* that are stored in registers inside the processor itself (there are no addresses). All the information exchange between processors happens exclusively through reading and writing into the cells of the common storage. The operations each processor can perform on (non-pointer) values include the “standard” arithmetic and comparison operations. The allowed pointer operations include copying a pointer to an internal pointer register, copying a pointer register to an external pointer field, or reading the contents of a cell to which a pointer refers (i.e., indirect access). If a pointer π refers to another pointer ρ, then an indirect access of π is allowed, i.e., one may copy ρ to an internal pointer register, and is commonly referred to as a “pointer hop”. A processor may compare two pointer registers for equality (to see if they point to the same cell), but pointer arithmetic (e.g., indexed addressing) is not allowed. In addition, a processor may create a new memory cell in a single time step.

Just as with the PRAM model, one may place constraints on the types of concurrent accesses that processors are allowed to make. Specifically, an EREW PPM prohibits more than one processor reading or writing the same cell of the storage at any instant, a CREW PPM allows common reads of a cell by many processors, but prohibits common writes to any cell in any step, and a CRCW PPM allows both concurrent reads and concurrent writes (assuming some type of conflict resolution). Note that each of these models is no stronger than its PRAM counterpart, since a PRAM can easily simulate each step of a PPM in $O(1)$ time and linear work, but a PPM does not support indexed addressing.

We are interested in the complexity of sorting on a PPM. Let us, then, briefly review a small sample of the voluminous work previously done on parallel sorting. In 1968 Batcher [4] gave what is considered to be the first parallel sorting scheme. Specifically, his was a sorting network that sorted in $O(\log^2 n)$ time using $O(n)$ processors. Since then there has been a considerable amount of work done for this important problem (e.g., see Bitton *et al.* [7], JáJá [15], Karp and Ramachandran [16], and Reif [14]). Nevertheless, it was not until 1983 that it was shown, by Ajtai, Komlós, and Szemerédi [2], that one can sort in $O(\log n)$ time with an $O(n \log n)$ sized sorting network (see also Paterson [23]). In 1985 Leighton [20] extended this result to show that one can produce an $O(n)$-node bounded-degree network capable of sorting n numbers in $O(\log n)$ steps. One drawback of these algorithms, however, is that they are strongly dependent on expander graphs. This dependence on expander graphs is not required for an optimal PRAM solution, however, for in 1988 Cole [10] gave simple methods for optimal sorting in the CREW and EREW PRAM models that do not use expander graphs, but instead are based on an elegant "cascade merging" paradigm using arrays. Interestingly, although Cole's procedure did not improve the asymptotic complexity of sorting on the PRAM model, it did lead to improvements in the asymptotic complexity of a number of computational geometry problems, as it was the key ingredient of the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [3].

Our interest in sorting on the PPM is motivated by a desire for a parallel mergesort procedure that is more closely akin to the linked-list implementation of the sequential mergesort procedure (e.g., see Knuth [18] and Sedgewick [24]). We show that one can achieve this goal, for we give optimal $O(\log n)$-time sorting algorithms for both the CREW PPM and EREW PPM models. Our methods are loosely based on the cascade merging paradigm introduced by Cole [10], but the details of our methods differ considerably from those of Cole's methods. His methods crucially depend on sorted sets being represented in arrays, and it seems impossible to implement his methods in a parallel pointer machine model and still maintain optimal performance.

Moreover, in addition to giving optimal PPM sorting algorithms, we also show that our linked-list parallel implementation of mergesort does, in fact, lead to *asymptotic* improvements to the parallel complexity of other problems. In particular, we generalize our linked-list based approach to optimally solve the *set-expression evaluation problem* in parallel. In this problem one is given a tree T such that each leaf of T stores a singleton set and each internal node is labeled with an operation from the set $\{\cup, \cap\}$, and one wishes to construct a sorted representation of the set defined by the root of T, assuming a bottom-up evaluation. This problem has applications in database querying and logic programming. We do not know of any previous parallel algorithms for this problem, but it appears that the array-based merging methods of Bilardi and Nicolau [6], Shiloach and Vishkin [25], or Hagerup and Rüb [12] and the tree-contraction techniques of Abrahamson *et al.* [1], Miller and Reif [21, 22], or Kosaraju and Delcher [19] will lead to an $O(\log^2 n)$ time solution with a time-processor product of $O(n \log n)$. We achieve $O(\log n)$ time while still maintaining the optimal time-processor product of $O(n \log n)$. This method extends our linked-list based technique in that it is performed on a directed acyclic graph (dag) rather than a binary tree.

In the next section we give some preliminary definitions and lemmas, and in Section 3 we give our optimal sorting algorithm for the CREW PPM model. In Section 4 we extend our approach to do cascade merging in dags with a constant-width tree partition, and we show how to use this to solve the set-expression evaluation problem optimally in parallel. In Section 5 we show how to modify our sorting algorithm to run in the EREW PPM model, and we conclude in Section 6.

2 Some Definitions and Merge Lemmas

Before we describe our sorting methods, we make the following definitions.

Definitions. *Let $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_m)$ be two sorted lists, represented as doubly-linked lists. We consider A to contain "virtual" elements a_0 and a_{n+1} that are respectively smaller and larger than all the elements in A and B (and similarly for B).*

1. *If every element a_i in A has a pointer $\beta(a_i)$ to an element of B, then A is linked into B. A is predecessor-linked into B (denoted $A \xrightarrow{\text{pred}} B$) if $\beta(a_i)$ points to the*

predecessor of a_i in B, i.e., the greatest element in B less than or equal to a_i. A is rank-linked into B (denoted $A \xrightarrow{\text{rank}} B$) if $n = m$ and, for each i, $\beta(a_i)$ points to b_i, which we call the twin of a_i.

2. *A is a sample of B if its elements form a subsequence of B. A is a (c, d)-sample of B if it is a sample of B and for any $0 \leq i \leq n$ the number of elements of B in the interval (a_i, a_{i+1}) is at least c and at most d.*

3. *An element a is in the c-neighborhood of an element b_i in B if $a \in [b_{i-c}, b_{i+c}]$, for $i \in \{1, 2, ..., m\}$, where we make the convention that $b_0 = b_{-1} = \cdots = b_{1-c} < \min\{a, b_1\}$ and $b_{n+1} = \cdots = b_{n+c} > \max\{a, b_n\}$. A c-conforms to B if $n = m$ and a_i is in the c-neighborhood of b_i for all $i = 1, ..., n$.*

4. *A is a c-cover of B if, for any two consecutive elements e and f in A, there are at most c elements of B in the interval $[e, f)$. (This definition is also used by Cole [10]).*

Note that the first definition deals with pointer relationships between A and B, and the last three definitions deal with value relationships. Let us make two observations about these relationships before going on.

Observation 2.1: *If A is a (c_1, c_2)-sample of B, then A is a $(c_2 + 1)$-cover of B.*

Observation 2.2: *If A c-conforms to B, then A is a $2c$-cover of B.*

There are essentially two types of merging procedures we use in our sorting algorithm, both of which are built upon the following simple lemma:

Lemma 2.3: *Suppose one is given three lists A, B, and C, such that $B \subseteq A$ and B is a d-cover of C, for some constant d. If one has $A \xrightarrow{\text{pred}} B$ and $B \xrightarrow{\text{pred}} C$, then one can compute $A \xrightarrow{\text{pred}} C$ in $O(1)$ time with a processor assigned to each element in A in the CREW PPM model.*

Proof: Let a be an element in A. Follow the pointer from a to its predecessor, b, in B, and follow the pointer from b to its predecessor, c, in C. Since B is a d-cover of C, the predecessor of a in C can be at most d positions away from c (if not, then b would not be the predecessor of a). Thus, given c's position in C, one can locate a's predecessor in $O(d) = O(1)$ additional steps. □

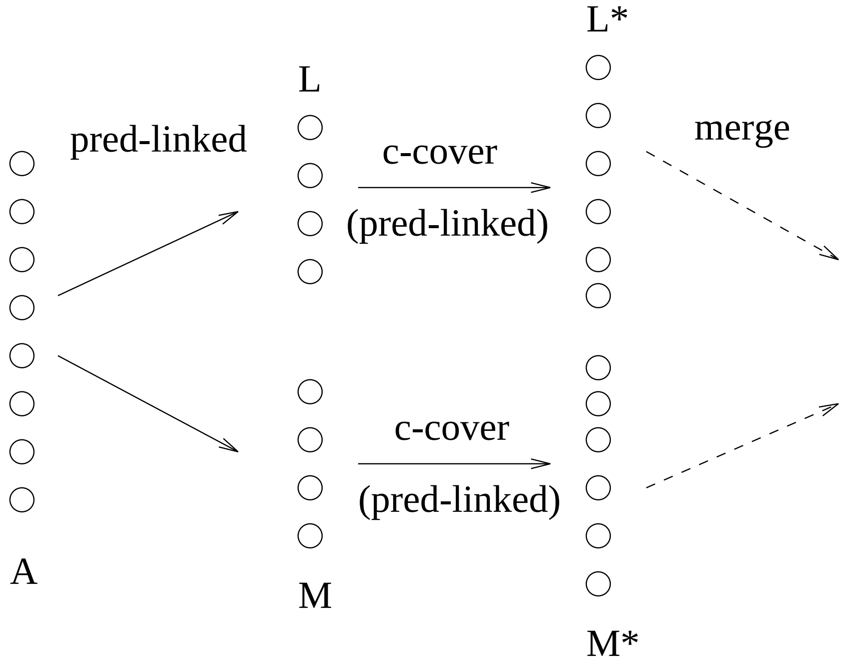

Figure 1: **The c-Cover Merge.**

Note that we placed no restrictions on the number of elements in A that can have the same predecessor in B; hence, there may be a number of concurrent reads in any single application of Lemma 2.3. The following lemma makes two applications of this lemma, and is illustrated in Figure 1.

Lemma 2.4: (The c-Cover Merge Lemma) *Suppose one is given lists A, L, M, L^*, and M^*, such that $A = L \cup M$, L is a c-cover of L^*, and M is a c-cover of M^*, where c is a constant. In addition, suppose one is given $A \xrightarrow{\text{pred}} L$, $A \xrightarrow{\text{pred}} M$, $L \xrightarrow{\text{pred}} L^*$, and $M \xrightarrow{\text{pred}} M^*$. Then one can compute $A^* = L^* \cup M^*$ (with $A^* \xrightarrow{\text{pred}} L^*$ and $A^* \xrightarrow{\text{pred}} M^*$) in $O(1)$ time with a processor assigned to each element in A in the CREW PPM model.*

Proof: By two applications of Lemma 2.3, one can compute $A \xrightarrow{\text{pred}} L^*$ and $A \xrightarrow{\text{pred}} M^*$ in $O(1)$ time. Let a and b be two consecutive elements in A, and let $L^*(a,b)$ and $M^*(a,b)$ be the sublists of L^* and M^*, respectively, that fall in the interval $[a,b)$. In parallel for each such pair a,b one can construct the portion of A^* that falls in the interval $[a,b)$ by merging $L^*(a,b)$ and $M^*(a,b)$ as in the sequential mergesort procedure. Since $L \subseteq A$ and $M \subseteq A$, and the fact that L is a c-cover of L^* and

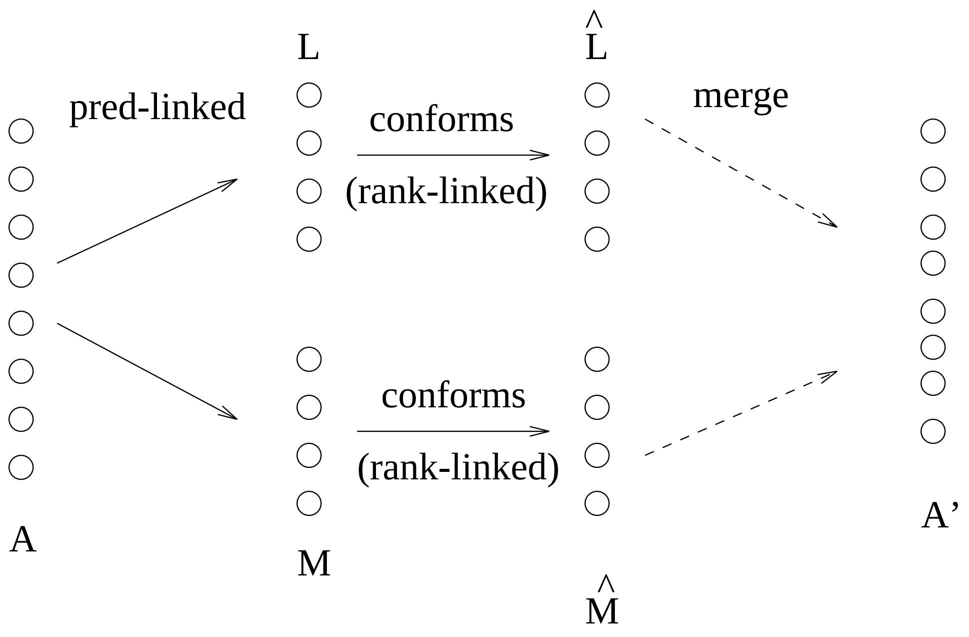

Figure 2: **The Conforming Merge.**

M is a c-cover of M^*, we have that $|L^*(a,b)| \le c$ and $|M^*(a,b)| \le c$. Thus, all the computations for this construction can be performed in $O(c) = O(1)$ time given a processor assigned to each element in A. □

Note that the c-Cover Merge Lemma deals with lists that are predecessor-linked into other lists. The Conforming-Merge Lemma, which we describe next, deals with rank-linked lists.

Lemma 2.5: (The Conforming-Merge Lemma) *Suppose one is given lists A, L, M, $\hat{L}$, and $\hat{M}$, such that $A = L \cup M$, L d-conforms to $\hat{L}$, and M d-conforms to $\hat{M}$, where d is a constant. Suppose further that one is given $A \xrightarrow{\text{pred}} L$, $A \xrightarrow{\text{pred}} M$, $L \xrightarrow{\text{rank}} \hat{L}$, and $M \xrightarrow{\text{rank}} \hat{M}$. Then one can compute $A' = \hat{L} \cup \hat{M}$ (with $A' \xrightarrow{\text{pred}} \hat{L}$ and $A' \xrightarrow{\text{pred}} \hat{M}$) and $A \xrightarrow{\text{rank}} A'$ in $O(1)$ time with a processor assigned to each element in A in the CREW PPM model.*

Proof: First, note that $|A| = |A'|$, since L d-conforms to $\hat{L}$ and M d-conforms to $\hat{M}$ (so $A \xrightarrow{\text{rank}} A'$ is well-defined). Also, by Observation 2.2, note that L is a $2d$-cover of $\hat{L}$ and M is a $2d$-cover of $\hat{M}$. Thus, we can apply the c-Cover Merge Lemma to compute A', with $A' \xrightarrow{\text{pred}} \hat{L}$ and $A' \xrightarrow{\text{pred}} \hat{M}$, in $O(1)$ time.

We have yet to show how to compute $A \xrightarrow{\text{rank}} A'$. That is, for each element a in A we need to find a's twin, a', in A'. Let i denote the rank of a in A. (Note: we may not actually know the value of i during an implementation of our algorithm, but this will not matter for the proof.) Since $A = L \cup M$, there is a copy of a in L or M; without loss of generality, suppose $a \in L$. Let $\hat{a}$ be the twin of a in $\hat{L}$ (with respect to the copy of a in L). Note that to determine the i-th element of A' it is sufficient to determine the rank of $\hat{a}$ in A', described in terms of i (e.g., by saying "$\hat{a}$ has rank $i-2$ in A'"). Let b be the predecessor of a in M, and let $\hat{b}$ be the twin of b in $\hat{M}$. Suppose a has rank j in L and b has rank k in M, so $i = j + k$. Note that $\hat{a}$ has rank j in $\hat{L}$, by definition. So we need only determine the rank of $\hat{a}$ in $\hat{M}$ (in terms of k). Since M d-conforms to $\hat{M}$, the predecessor of $\hat{a}$ in $\hat{M}$ can be at most $d+1$ positions away from $\hat{b}$; hence, we can determine the (signed) difference between the rank of $\hat{a}$ in $\hat{M}$ and the rank of $\hat{b}$ in $\hat{M}$ just by counting the number of elements between $\hat{a}$'s predecessor in $\hat{M}$ and $\hat{b}$. Let $-d-1 \le l \le d+1$ denote this difference, so that the rank of $\hat{a}$ in $\hat{M}$ is $k+l$. Then the rank of $\hat{a}$ in A' is $j+k+l = i+l$. This completes the proof, for to determine the i-th element of A' we need only march $|l|$ positions (left, if $l < 0$, right, if $l > 0$) in A' from $\hat{a}$'s position in A'. □

Note that this proof depended on a rank-based argument, but we did not need to explicitly maintain ranks. All that was needed was the *relative* ranks of various elements.

Having given the main lemmas for our method, we now discuss how to sort n elements in a CREW PPM in $O(\log n)$ time using $O(n)$ processors.

3 Sorting on a CREW PPM

Let $S = \{a_1, a_2, ..., a_n\}$ be a collection of elements taken from some total order, and stored one element per memory cell in a CREW PPM, such that the cell for a_i has a pointer to the cell for a_{i+1} (for $i \in \{1, 2, ..., n-1\}$). In addition, we assume that there are n processors, such that processor i has a pointer to a_i. We also assume that the elements in S are distinct. This introduces no loss of generality, for one can use the convention that if $a_i = a_j$, then take $a_i < a_j$ if and only if $i < j$. Since ours is a comparison-based sorting method, this will produce a stable sorting.

We begin our algorithm by building a complete (balanced) binary tree T with n leaves such that the elements of S are stored in T one element per leaf. T can easily be built in $O(\log n)$ time using the n processors, by, say, a recursive doubling procedure (see JáJá [15], Karp and Ramachandran [16], or Reif [14]). As in parallel mergesort procedure of Cole [10], we view T as the schematic for a parallel mergesort procedure. We define a list $A(v)$ for each v in T to be the sorted list of all elements stored in descendents of v. A high-level description of our algorithm is similar to the parallel mergesort procedure of Cole [10], in that we construct the $A(v)$'s in a pipelined fashion in a series of $O(\log n)$ stages (the details of our method are quite different than those for Cole's method, however).

Throughout, we make the notational convention that v is an arbitrary node, x and y are its children, and u is its parent. We also make the following definition:

Definition: *Let $L = (a_{i_1}, a_{i_2}, ..., a_{i_k})$ be a sample of A and let M be a sample of B. L and M are* parallel *samples if $M = (b_{i_1}, b_{i_2}, ..., b_{i_k})$, i.e., the same indices are chosen.*

3.1 CREW Stage Invariants and Computations

Let $A_t(v)$ denote the list stored at node v in T at the end of stage t. Each $A_t(v)$ list is represented in sorted order as a doubly-linked list, and consists of the elements that have been "passed" to node v from x and y. We also store a sample, $L_t(v)$, of $A_t(v)$, where the elements in $L_t(v)$ are to be passed up to u in the next stage $(t+1)$. As in the algorithms of Atallah *et al.* [3] and Cole [10], we say that a node v is *full* after stage t if $A_t(v)$ contains all the elements stored in the descendents of v, and v is *active* if $A_t(v) \neq \emptyset$ and u is not full. Besides $L_t(v)$ and $A_t(v)$, we also store $L_{t-1}(v)$ and another list $\hat{L}_{t-1}(v)$, at v, which facilitate the proper assimilation of the elements of $L_t(x)$ and $L_t(y)$ into $A_{t+1}(v)$. Our method maintains the following invariants at the end of each stage t, as illustrated in Figure 3.

CREW List Invariants at the end of stage t: *Each active node v stores four sorted lists, $A_t(v)$, $L_{t-1}(v)$, $\hat{L}_{t-1}(v)$, and $L_t(v)$, as doubly-linked lists that satisfy the following properties:*

1. *If v was not full after stage $t-1$, then $A_t(v) = L_{t-1}(x) \cup L_{t-1}(y)$ and we have $A_t(v) \xrightarrow{\text{pred}} L_{t-1}(x)$ and $A_t(v) \xrightarrow{\text{pred}} L_{t-1}(y)$. If v was full after stage $t-1$, then $A_t(v) = A_{t-1}(v)$.*

2. *$L_{t-1}(v)$ 1-conforms to $\hat{L}_{t-1}(v)$, and is rank-linked into $\hat{L}_{t-1}(v)$.*

3. *$\hat{L}_{t-1}(v)$ is a $(0,1)$-sample of $L_t(v)$, and is predecessor-linked into $L_t(v)$.*

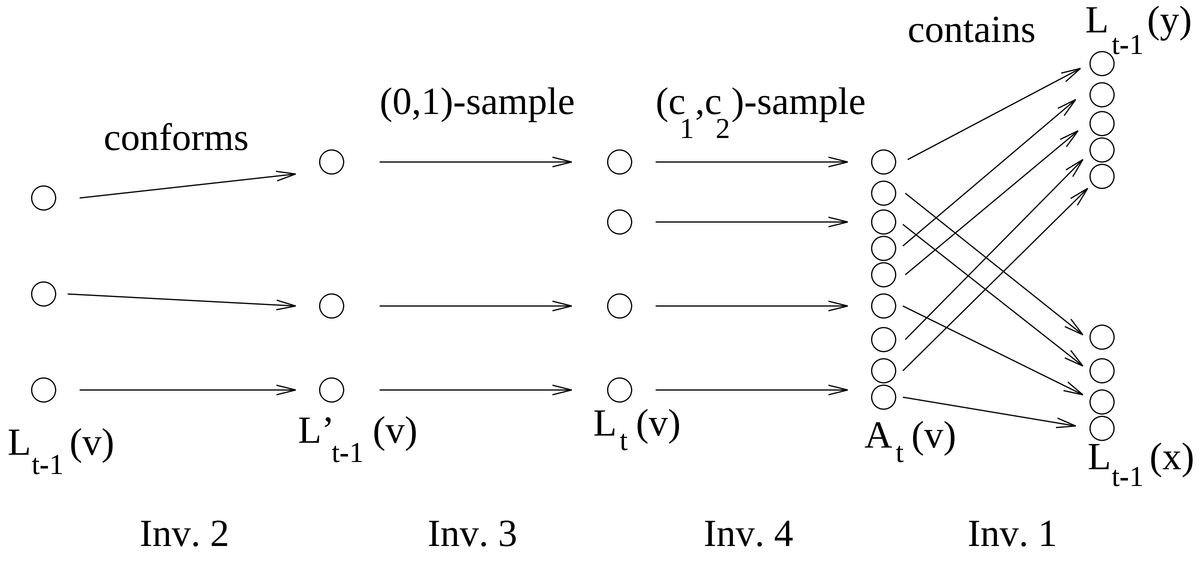

Figure 3: **The stage invariants.**

4. *If v is not full, then $L_t(v)$ is a (c_1, c_2)-sample of $A_t(v)$, where c_1 and c_2 are constants (fixed for the entire procedure, and set in the analysis). If v is full, then $L_t(v)$ is a $(\lfloor c_1/2^i \rfloor, \lfloor c_2/2^i \rfloor)$-sample, where i is the number of stages since v first became full. In any case, $L_t(v)$ is predecessor-linked into $A_t(v)$.*

Stage $t+1$ Computation: *In Step 1, for each active node v, we construct temporary lists $A'_t(v)$ and $L'_t(v)$ and additional linkages to aid us in Step 2. In Step 2 we construct $A_{t+1}(v)$, in Step 3 we construct $\hat{L}_t(v)$, and in Step 4 we construct $L_{t+1}(v)$.*

Step 1: If v is not full, then we construct a temporary list $A'_t(v) = \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$, with $A'_t(v) \xrightarrow{\text{pred}} \hat{L}_{t-1}(x)$ and $A'_t(v) \xrightarrow{\text{pred}} \hat{L}_{t-1}(y)$. If v is full, then we simply take $A'_t(v) = A_t(v)$.

Implementation: This can all be done in $O(1)$ time with a processor assigned to each element of $A_t(v)$, by the Conforming-Merge Lemma. We can apply this lemma by List Invariant 1 at v and List Invariant 2 at x and at y. Recall that the Conforming-Merge Lemma also gives us $A_t(v) \xrightarrow{\text{rank}} A'_t(v)$. We use this ranking to construct the sample $L'_t(v)$ of $A'_t(v)$ that is parallel to $L_t(v)$ in $A_t(v)$.

Comment: Since $L'_t(v)$ and $L_t(v)$ are parallel samples, $L'_t(v)$ is a (c_1, c_2)-sample of $A'_t(v)$, for $L_t(v)$ is a (c_1, c_2)-sample of $A_t(v)$ (by List Invariant 4). (See Figure 4.)

Step 2: If v is not full, then we construct $A_{t+1}(v) = L_t(x) \cup L_t(y)$. If v is full, then we simply take $A_{t+1}(v) = A'_t(v)$.

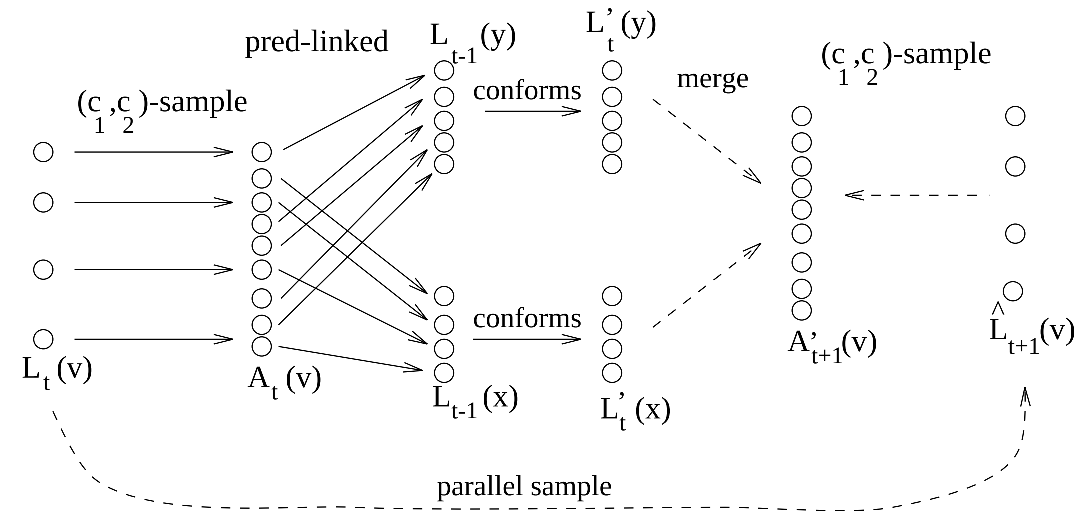

Figure 4: **Illustrating Step 1, the construction of temporary lists $A'_t(v)$ and $L'_t(v)$.**

Implementation: We can perform this step in $O(1)$ time with a processor assigned to each element of $A'_t(v)$, by the c-Cover Merge Lemma. We can apply this lemma, since $A'_t(v)\xrightarrow{\text{pred}}\hat{L}_{t-1}(x)$ and $A'_t(v)\xrightarrow{\text{pred}}\hat{L}_{t-1}(y)$ (by the result of applying the Conforming-Merge Lemma in Step 1), and we have List Invariant 3 satisfied at x and at y. (See Figure 5.)

Comment: This gives us List Invariant 1 at v for after stage $t+1$. In addition, from List Invariant 2 at v's children, it is easy to see that $L'_t(v)$ is a $(c_1, 2c_2+2)$-sample of $A_{t+1}(v)$ (if v is full, then it is a (c_1, c_2)-sample). However, this is not a "good enough" sample of $A_{t+1}(v)$ for our purposes.

Step 3: If v is not full, then we locally shift $L'_t(v)$ in $A_{t+1}(v)$ to construct a $(c_1, 2c_2+1)$-sample, $\hat{L}_t(v)$, of $A_{t+1}(v)$. We also rank-link $L'_t(v)$ into $\hat{L}_t(v)$, which implicitly ranks $L_t(v)$ into $\hat{L}_t(v)$. If v is full, then we take $\hat{L}_t(v) = L'_t(v)$.

Comment: Intuitively, our method is to "take" an element from each large interval of $A_{t+1}(v)$ between two consecutive elements of $L'_t(v)$, and "pass" it down the list, from interval to interval, until we find a small interval to "give" it to.

Implementation: Let $b \in A_{t+1}(v)$ be an element in $L'_t(v)$. Since b is in $L'_t(v)$, it is also in $A'_t(v)$. Thus, since $A'_t(v) = \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$ and $A'_t(v)$ is predecessor-linked into $\hat{L}_{t-1}(x)$ and $\hat{L}_{t-1}(y)$, we can easily determine b's predecessor in $\hat{L}_{t-1}(x)$ and $\hat{L}_{t-1}(y)$, respectively, one of which must be b itself. Without loss of generality, assume that

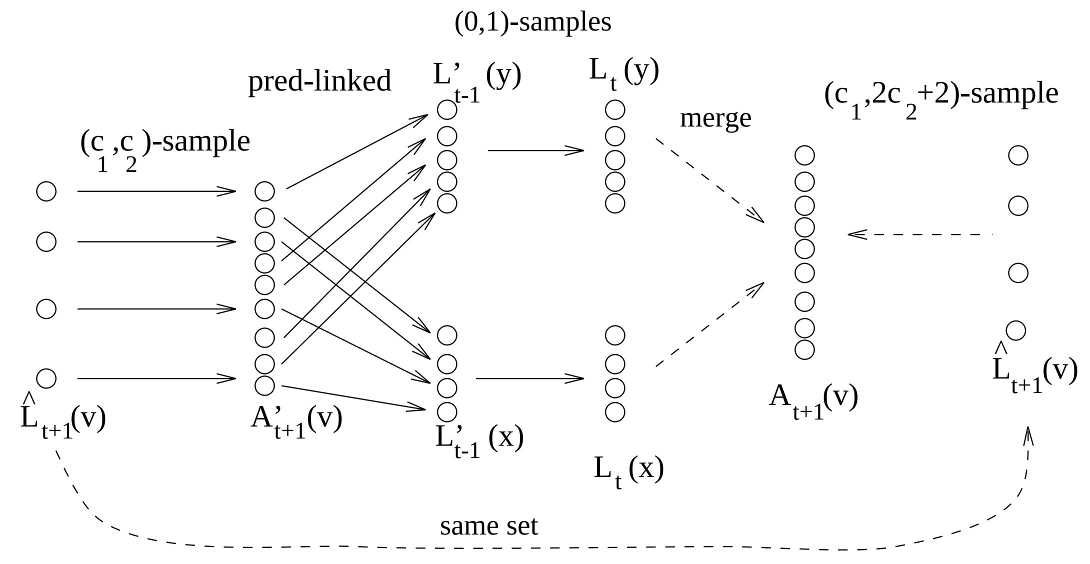

Figure 5: **Illustrating Step 2, the construction of $A_{t+1}(v)$.**

$b \in \hat{L}_{t-1}(y)$. Let e and g be the two consecutive elements of $\hat{L}_{t-1}(x)$ such that b is in the interval (e, g). Since $\hat{L}_{t-1}(x)$ is a $(0,1)$-sample of $L_t(x)$ and is predecessor-linked into $L_t(x)$ (by List Invariant 3 at x), we can easily determine the element, f (if it exists) in $L_t(x)$ such that f is in the interval (e, g). This is the value that determines how we perform our local shift relative to b. In particular, if $f < b$, then we make the immediate predecessor, a, of b in $A_{t+1}(v)$ take b's place in $\hat{L}_t(v)$, i.e., we put a in $\hat{L}_t(v)$ instead of b—so that a is b's twin in $L_t(v)$. On the other hand, if $f > b$, then we let the copy of b in $A_{t+1}(v)$ also be in $\hat{L}_t(v)$—so that b is its own twin in $L_t(v)$. We show below, in Lemma 3.1, that this simple rule forces $\hat{L}_t(v)$ to be a $(c_1, 2c_2+1)$-sample of $A_{t+1}(v)$.

Step 4: We refine $\hat{L}_t(v)$ into $L_{t+1}(v)$. We place each element of $\hat{L}_t(v)$ in $L_{t+1}(v)$, and, for each sublist $B_v(e)$ of $A_{t+1}(v)$ consisting of all elements that are strictly between two consecutive elements e and f of $\hat{L}_t(v)$, we make the median of $B_v(e)$ also be a member of $L_{t+1}(v)$, provided v is full or $|B_v(e)| > c_2$. This completes the computation for Stage $t+1$.

Comment: After completing Step 4 for stage $t+1$, List Invariant 3 holds, since at most one element of $L_{t+1}(v)$ exists in between any two elements of $\hat{L}_t(v)$. Moreover, if v was previously not full, then this forces $L_{t+1}(v)$ to be a (c_1, c_2)-sample of $A_{t+1}(v)$ (assuming $\hat{L}_t(v)$ is a $(c_1, 2c_2+1)$-sample and $c_1 \geq \lfloor (c_2+1)/2 \rfloor$), which satisfies List

Invariant 4. If, on the other hand, v was previously full, then this forces $L_{t+1}(v)$ to be a $(\lfloor c_1/2^i \rfloor, \lfloor c_2/2^i \rfloor)$-sample of $A_{t+1}(v)$, where i is the number of stages since v first became full.

End of Stage $t+1$.

Having completed the description of our method, we next prove its correctness.

3.2 Analysis of Our Method for CREW PPM sorting

As observed above, when Step 4 completes, we will have satisfied List Invariants 1 and 3. The next lemmas show that our sample-refining method of Step 4 will correctly give us List Invariants 2 and 4, as well.

Lemma 3.1: *$\hat{L}_t(v)$ is a $(c_1, 2c_2+1)$-sample of $A_{t+1}(v)$.*

Proof: Recall that $L'_t(v)$ is a (c_1, c_2)-sample of $A'_t(v)$, that $A'_t(v) = \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$, and that $\hat{L}_{t-1}(x)$ (resp., $\hat{L}_{t-1}(y)$) is a $(0,1)$-sample of $L_t(x)$ (resp., $L_t(y)$). Thus, $L'_t(v)$ is a $(c_1, 2c_2+2)$-sample of $A_{t+1}(v)$. Let $I = (a, b)$ be an interval defined by two consecutive elements, a and b, in $L'_t(v)$. So the number of elements of $A_{t+1}(v)$ in I, which we denote by c, is at most $2c_2+2$. Let $\hat{I}$ denote the interval $(\hat{a}, \hat{b})$ defined by the twins, $\hat{a}$ and $\hat{b}$, of a and b, respectively, in $\hat{L}_t(v)$. Then, if we let $\hat{c}$ denote the number of elements of $A_{t+1}(v)$ in $\hat{I}$, then we wish to show that $\hat{c} \le 2c_2+1$.

Since both a and b are in $L'_t(v)$, they both are also in $A'_t(v) = \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$. Let α denote the child of v (i.e., one of x or y) such that a is not in $\hat{L}_{t-1}(\alpha)$. Also let e and g be the two consecutive elements of $L_{t-1}(\alpha)$ such that a is in the interval (e, g), and let f denote the element (if it exists) of $L_t(\alpha)$ in this same interval. Similarly, let β denote the child of v such that b is not in $\hat{L}_{t-1}(\beta)$, and let p and r be the two consecutive elements of $L_{t-1}(\beta)$ such that b is in the interval (p, r), and let q denote the element (if it exists) of $L_t(\beta)$ in this same interval. We distinguish three cases for the magnitude of c:

Case 1. $c = 2c_2+2$. Note that for this to be the case, f and q must both be in I. But if q is in I, then q is less than b; hence, the predecessor of b in $A_{t+1}(v)$ is the twin of b in $\hat{L}_{t+1}(v)$. Moreover, if f is in I, then f is greater than a; hence, a is its own twin in $\hat{L}_{t+1}(v)$. Thus, $\hat{c} = 2c_2+1$.

Case 2. $c = 2c_2+1$. Note that for this to be the case, f or q must be in I, but not both. If $f \in I$ and $q \notin I$, then a (resp., b) is its own twin in $\hat{L}_t(v)$; hence, $\hat{c} = 2c_2+1$

in this case. If, on the other hand, $f \notin I$ and $q \in I$, then the predecessor of a (resp., b) is a's twin (resp., b's twin) in $\hat{L}_t(v)$; hence, $\hat{c} = 2c_2 + 1$ in this case, as well.

Case 3. $c < 2c_2 + 1$. If both f and g are in I, then, by a previous argument, $\hat{c} = c - 1$, and we're done. If exactly one of f or q is in I, then we are also done, since, by another previous argument, $\hat{c} = c$ in this case. If, on the other hand, neither f nor q is in I, then the predecessor of a is a's twin in $\hat{L}_t(v)$ while b is its own twin in $\hat{L}_t(v)$. In this case, $\hat{c} = c + 1$; but this still gives us $\hat{c} \leq 2c_2 + 1$. This completes the proof. □

Lemma 3.2: *$A_t(v)$ 2-conforms to $A'_t(v)$.*

Proof: If v is full, then $A_t(v) = A'_t(v)$; hence, the lemma is trivially true in this case. So consider the case when v is not full. Then $A_t(v) = L_{t-1}(x) \cup L_{t-1}(y)$ and $A'_t(v) = \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$. Let a_i denote the i-th element of $A_t(v)$ and let a'_i denote the i-th element of $A'_t(v)$ (i.e., a_i's twin), for $i \in \{1, 2, ..., |A_t(v)|\}$. Also, let $\hat{a}_i$ denote the twin in $\hat{L}_{t-1}(\alpha)$ of the copy of a_i in $L_{t-1}(\alpha)$, where α is x if a_i came from $L_{t-1}(x)$, and α is y otherwise. Suppose, for the sake of contradiction, that there is an $i \in \{3, 4, ..., |A_t(v)|\}$ such that $a_i < a'_{i-2}$. Without loss of generality, a_i came from $L_{t-1}(x)$. Note that each a'_j is an $\hat{a}_k$ for some k, i.e., the j-th element of $A'_t(v)$ must be the twin (in some $\hat{L}_{t-1}(\alpha)$) of some element in $A_t(v)$ (i.e., the k-th).
Claim: There must be at least two elements a_g and a_h in $A_t(v)$, with $a_g < a_h < a_i$, such that $a_i < \hat{a}_g$ and $a_i < \hat{a}_h$.
Proof of claim: This claim follows by a simple pigeonhole argument. First, note that a'_{i-2} is the $(i-2)$-nd element in $A'_t(v)$ and a_i is smaller than a'_{i-2}; hence, there are at most $i - 3$ elements of $A'_t(v)$ less than a_i. But there are $i - 1$ elements less than a_i in $A_t(v)$, and each a'_j is an $\hat{a}_k$ for some k. Therefore, there must be two a_k's (i.e., a_g and a_h) that meet the conditions of the claim. □ *(for proof of claim)*

This claim immediately implies that both a_g and a_h came from $L_{t-1}(y)$. If this were not so, then it would contradict the List Invariant 2 at x (for stage $t - 1$), i.e., that $L_{t-1}(x)$ 1-conforms to $\hat{L}_{t-1}(x)$ (for, otherwise, the copy of a_i in $L_{t-1}(x)$ would not be in the 1-neighborhood of $\hat{a}_i$ in $\hat{L}_{t-1}(x)$). But this claim also implies that $a_h < \hat{a}_g$ (since $a_h < a_i$), which contradicts List Invariant 2 at y (for it implies that the copy of a_h in $L_{t-1}(y)$ cannot be in the 1-neighborhood of $\hat{a}_h$ in $\hat{L}_{t-1}(y)$). Thus, $a'_{i-2} \leq a_i$. By a similar argument we also have that $a_i \leq a'_{i+2}$, for $i \in \{1, 2, ..., |A_t(v)| - 2\}$. Therefore, $A_t(v)$ 2-conforms to $A'_t(v)$. □

As a simple corollary to this lemma, we have that List Invariant 2 is satisfied, provided c_1 is sufficiently large:

Corollary 3.3: *If $c_1 \geq 2$, then $L_t(v)$ 1-conforms to $\hat{L}_t(v)$.*

Proof: If v is full, then $L_t(v) = \hat{L}_t(v)$, and we're done. So suppose v is not full. As an immediate consequence of Lemma 3.2, we have that $a'_{i-3} < a_i < a'_{i+3}$, where a_i (resp., a'_i) denotes the i-th element of $A_t(v)$ (resp., $A'_t(v)$). Suppose a_i is in $L_t(v)$, and let a_{i+j} be the next element of $A_t(v)$ in $L_t(v)$ (i.e., the smallest element of $L_t(v)$ greater than a_i). Then a'_i and a'_{i+j} are consecutive elements in $\hat{L}_t(v)$. By List Invariant 4, if $c_1 \geq 2$, then $j \geq 3$. Thus, $L_t(v)$ 1-conforms to $\hat{L}_t(v)$. $\square$

So, after Step 4 completes, we are ready to begin the next stage, as all the invariants have been satisfied. Also we have placed some restrictions on the values of c_1 and c_2, namely that $c_1 \geq \max\{2, \lfloor (c_2+1)/2 \rfloor\}$ and $c_2 \geq 2c_1$. So, for example, we could take $c_1 = 2$ and $c_2 = 4$, which would guarantee that if the nodes on a particular level of T become full in stage t, then the nodes on the parent level will become full in stage $t+4$. In particular, $(c_1, c_2) = (2, 4)$ after the 1st iteration, $(c_1, c_2) = (0, 2)$ after the 2nd iteration, $(c_1, c_2) = (0, 1)$ after the 3rd iteration, and $(c_1, c_2) = (0, 0)$ after the 4th iteration. Thus, after $4\lceil \log n \rceil$ stages the root becomes full.

From the description of each step in our stage computation, it should be clear that each step can be implemented using only pointer machine operations. Moreover, each step requires only $O(1)$ time given a processor and an $O(1)$-sized block of memory assigned to each element in $A_t(v)$. Therefore, our sorting algorithm runs in $O(\log n)$ time, so long as we can efficiently solve the space- and processor-allocation problems.

3.3 Space and Processor Allocation

Our method for allocating space and processors is based on a token-passing scheme, which can be viewed as "de-amortization" of the total computational effort. Specifically, we let a token represent a processor assignment, as well as an $O(1)$-size block of memory cells, and maintain the following invariant for tokens:

CREW Token Invariant (for after stage t): *Each element e of $A_t(v)$ has a token associated with it, unless e is also in $\hat{L}_{t-1}(v)$.*

Initially, the element in each leaf node has a token associated with it. The token assigned to an element e in $A_t(v)$ is passed to e's twin in $A'_t(v)$ (if e has a token to pass) in Step 1. Thus, an e in $A'_t(v)$ will have a token unless e is also in $L'_t(v)$. In Step 2, when v receives new elements from x and y, we pass a token with each element v receives. Finally, in Step 3, we shift tokens so as to satisfy the token invariant for stage $t+1$. Thus, we can maintain the token invariant throughout the computation. Note that this token passing scheme is sufficient for solving the processor assignment problem, because, for any v that is not full, each element without a token is adjacent to an element with a token (by List Invariant 4). In other words, if we let each token represent an $O(1)$ block of storage, and a processor assignment, we need only make sure that for each element e that has no token we store the record for e in the block for one of e's neighbors f (since f must have a token), and let the processor for f perform the computation for e. Since we start with n tokens, this implies that we can implement our algorithm in $O(n)$ space using $O(n)$ processors. Another interesting aspect of our token-passing scheme is that the space and processors associated with inactive nodes are automatically re-allocated to active nodes by a completely local strategy, i.e., by using only pointer machine operations. Thus, we have the following theorem.

Theorem 3.4: *Given a set S of n elements, one can sort S in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PPM model.* $\Box$

In the next section we apply our approach to a problem with applications to database querying and logic programming so as to achieve an algorithm that is asymptotically more efficient than what seems possible using previously known (array-based) strategies.

4 Cascade Merging in a DAG

In this section we generalize our linked-list based approach to cascade merging problems defined on certain directed acyclic graphs (dags), dags with a bounded-width tree partition, and give an application to the set-expression evaluation problem.

We begin with some definitions. Let $G = (V, E)$ be a connected dag. If there is an edge from u to v in G, then u is an *in-node* of v and v is an *out-node* of u. The *in-degree* (resp., *out-degree*) of a node v in G is the number of in-nodes (resp., out-nodes) v has. We let *degree*(G) denote the maximum, over all $v \in G$, of the sum of the in-degree of v and the out-degree of v. A node without any in-nodes is a *source*, and a node without any out-nodes is a *sink*. A *tree partition* Π of G is a partitioning of V into $V_1, V_2, ..., V_m$ such that if the nodes in each V_i are compressed to a single node, and induced parallel edges are removed, then the resulting graph is a tree $T(\Pi)$, called the *underlying tree for G with respect to* Π, or simply the *underlying tree* if the context is understood. There is an arc from V_i to V_j if there is an edge in G from a node in V_i to a node in V_j (and there is no edge from a node in V_j to a node in V_i). By a slight abuse of notation we use V_i to denote both a set of vertices in G and the single node in the underlying tree for G. We call each V_i a *super node* in $T(\Pi)$. The *width* of a tree partition $\Pi = \{V_1, V_2, \ldots, V_m\}$ is defined as

$$width(\Pi) = \max_{V_i \in \Pi} |V_i|,$$

where $|V_i|$ denotes the number of nodes in V_i. $T(\Pi)$ is *rooted* if it has only one sink node, called the *root* of T. The *height* of a node v in $T(\Pi)$ is the length of the longest leaf-to-v path in $T(\Pi)$. If $T(\Pi)$ is rooted, then the *height* of $T(\Pi)$ is the height of the root node; we let $height(T(\Pi))$ denote this quantity.

Suppose one is given a dag G. As a straightforward generalization of the sorting framework of the previous sections, one can define a cascade merging structure on G. In particular, one can associate a singleton set $A(v_i) = \{a_i\}$ with each source node v_i in G, and define a set $A(v)$ for each non-source node v to be the sorted union of $A(w_1), A(w_2), ..., A(w_{m_v})$, where $w_1, w_2, ..., w_{m_v}$ are the in-nodes for v. The *cascade merging* problem for G, then, is to construct $A(v)$ for each v in G. In this section we prove the following theorem:

Theorem 4.1: *Suppose one is given an n-node dag G with a bounded-width tree partition Π. If the underlying tree $T(\Pi)$ is a rooted binary tree, then one can solve any cascade merging problem for G in $O(height(T(\Pi)))$ time using $O(n)$ processors in the CREW PPM model.*

4.1 Some Simplifying Assumptions

Before we give our method for cascading in a dag, we first make some simplifying assumptions. Note that if G has a tree partition Π of width k, then the out-degree of each node in G must be at most k. Hence, if Π has constant width, then there must be at most $kn = O(n)$ edges in G. Moreover, if $T(\Pi)$ is binary, then the in-degree of each node in G must be at most $2k$. We make the following two assumptions regarding G:

- *Each V_i is an independent set.*

- *The in-degree and out-degree of each node in G is at most 2.*

These two assumptions are made without loss of generality. Our justification for this is based on the fact that one can easily embed any dag G into a "functionally equivalent" dag H that has these properties. Formally, an *embedding* of G in a graph H is a one-to-one mapping f of the nodes of G to the nodes in H, and edges in G to paths in H. Such a mapping is *functionality preserving* if $A(v) = A(f(v))$ for each $v \in G$ (assuming we define $A(w)$ for each $w \in H$ as above). Two important measures of the goodness of such an embedding are its *dilation cost* which is the length of the longest path in H to which an edge in G is mapped, and its *expansion cost*, which is the ratio of the number of nodes in H to the number of nodes in G (e.g., see Hong, Mehlhorn, and Rosenberg [13]). The following lemmas use these measures to analyze how the above independence assumption can be made without loss of generality.

Lemma 4.2: *Suppose one is given a dag $G = (V, E)$ with tree partition $T(\Pi)$. Then there is a functionality-preserving embedding f of G into a dag H, with tree partition $T(\Gamma)$, such that*

- *each set $W_i \in \Gamma$ is an independent set,*

- *$width(\Gamma) = width(\Pi)^2$, and*

- *f has dilation cost $width(\Pi)$ and expansion-cost $width(\Pi)^2$.*

Proof: Our proof is constructive, in that we describe how to build f, H, and Γ from G and Π. Let V_i be a member of Π that is not an independent set (if there are no such V_i's, then we are done). We refine V_i into a collection of independent sets $V_{i,1}, V_{i,2}, ..., V_{i,l}$, where $l \leq k$, such that for each edge (v, w), with $v, w \in V_i$, we have $v \in V_{i,j_1}$ and $w \in V_{i,j_2}$, with $j_1 < j_2$. This essentially amounts to a topological ordering of the nodes in V_i. We add "dummy nodes" $u_1, u_2, ..., u_m$ for each edge (v, w), with $v \in V_{i,j_1}$ and $w \in V_{i,j_2}$, to change (v, w) into a path $(v, u_1, u_2, ..., u_m, w)$, where $u_1 \in V_{i,j_1+1}$, $u_2 \in V_{i,j_1+2}$, ..., $u_m \in V_{i,j_2-1}$. The addition of these dummy nodes clearly preserves functionality and results in a dilation cost of $width(\Pi)$ for the associated embedding f. Moreover, it adds at most $width(\Pi)^2$ dummy nodes to each V_{i,j_k}; hence, implying that $width(\Gamma) = width(\Pi)^2$ and that f has an expansion cost of $width(\Pi)^2$. □

It is straightforward, given the above proof, to show that the construction of f, H, and Γ can be done in $O(width(\Pi)^2)$ time using $O(|G|)$ processors; we leave this to the reader. The next lemma shows that our degree restriction is also made without loss of generality.

Lemma 4.3: *Suppose one is given a dag $G = (V, E)$ with tree partition $T(\Pi)$. Then there is a functionality-preserving embedding f of G into a dag H, with tree partition $T(\Gamma)$, such that*

- *the in-degree and out-degree of each node in H are both at most 2,*
- *$width(\Gamma) = width(\Pi)$, and*
- *f has dilation cost at most $2\lceil \log degree(G) \rceil$ and expansion-cost at most $2degree(G)$.*

Proof: The proof is based on the idea, for each node $v \in G$, of combining the in-coming (resp., out-going) edges for v into a binary tree of height at most $\lceil \log degree(G) \rceil$. The interested reader is referred to the work of Atallah *et al.* [3] and Chazelle and Guibas [9] for examples of this type of transformation. $\Box$

The construction of such an f and H can easily be done in $O(\log degree(G))$ time given a processor assigned to each edge in G using a method of Atallah *et al.* [3]. Taken together, the above two lemmas, immediately imply the following corollary:

Corollary 4.4: *Suppose one is given a dag $G = (V, E)$ with tree partition $T(\Pi)$. Then there is a functionality-preserving embedding f of G into a dag H, with tree partition $T(\Gamma)$, such that*

- *each set $W_i \in \Gamma$ is an independent set,*
- *the in-degree and out-degree of each node in H is at most 2,*
- *$width(\Gamma) = width(\Pi)^2$, and*
- *f has dilation cost at most $2width(\Pi)\lceil \log degree(G) \rceil$ and expansion-cost at most $2width(\Pi)^2 degree(G)$.*

In our primary application, set-expression evaluation, we have that $degree(G)$ and $width(\Pi)$ are both $O(1)$. Thus, the above corollary gives a functionality-preserving embedding with constant dilation and expansion costs in this case.

4.2 Our Method for Cascading in a DAG

Having made some simplifying assumptions, we are now ready to describe our cascade merging method. For notational consistency with the other sections in this paper, in the remainder of this section whenever we use v for a node in G we let x and y be the in-nodes of v and let u and w be the out-nodes of v (note that y and/or w may not actually exist for every v in G, however). In addition, to simplify our notation, we use k as a shorthand for $width(\Pi)$.

As in our approach for cascade merging in trees, the computation proceeds in stages. For each node v in G we store a list $A_t(v)$ for v after stage t. As before, we will maintain a sample, $L_t(v)$ of each $A_t(v)$. Our stage invariant is the same as in the sorting algorithm, as is our computation for stage $t+1$.

Computation for Stage $t+1$**:** In Step 1, for each active node v, we construct temporary lists $A'_t(v)$ and $L'_t(v)$ and additional linkages to aid us in Step 2. In Step 2 we construct $A_{t+1}(v)$, in Step 3 we construct $\hat{L}_t(v)$, and in Step 4 we construct $L_{t+1}(v)$. The details are as in Section 3.

There are three significant differences between this algorithm and our sorting algorithm, however. The first difference deals with the fact that the nodes in a super node V_i may be at different heights in G. Because of this, we modify Step 4 of our method for a node $v \in V_i$ (where we refine $\hat{L}_t(v)$ into $L_{t+1}(v)$) so that if v is full, then we only mark additional members of $A_{t+1}(v)$ to be in $L_{t+1}(v)$ if all the nodes in V_i are full. This provides a mechanism to synchronize our computation so that the first stage, t, such that $L_t(v) = A_t(v)$ is the same for all v in V_i.

The second difference is that we must have some way of avoiding an explosion in the number of multiple copies of a single element in any $A(v)$ list. This is crucial, for if our method does not eliminate multiple copies (in an on-line fashion), then it is easy to construct dags that give rise to exponential-sized $A(v)$ lists. (This is in fact the main reason why it seems impossible to optimally apply the array-based cascade merging methods of Cole [10] to solve this problem.) Our method for dealing with this difficulty is actually quite simple. When a node v becomes full, we adjust $A_t(v)$ by contracting any multiple copies of an element e into a single copy. This simple strategy gives us the following lemma:

Lemma 4.5: *For any node v in G and any element e in $A_t(v)$, there can be at most two copies of e in $A_t(v)$.*

Proof: The proof is by induction on t. If v becomes full after stage $t-1$, then by our adjusting procedure, there can be only one copy of e in $A_t(v)$. If v is not full, but x and y were full after stage $t-1$, then by our adjustment procedure at x and y, there can be at most one copy of e in each of $A_{t-1}(x)$ and $A_{t-1}(y)$; hence, there can be at most two copies of e in $A_t(v)$. If, on the other hand, v and its in-nodes are not full, then by induction, there can be at most two copies of e in each of $A_{t-1}(x)$ and $A_{t-1}(y)$. Thus, if $c_1 \geq 1$, there can be at most one copy of e in each of $L_{t-1}(x)$ and $L_{t-1}(y)$; hence, there can be at most two copies of e in $A_t(v)$. $\Box$

This lemma immediately implies that our adjustment procedure can be implemented in $O(1)$ time, since we contract at most two copies of any element e. Also, since we only perform this adjustment after a node v becomes full, our adjustment procedure cannot affect any of our stage invariants (for we can still maintain that $L_{t-1}(v) \subseteq L_t(v)$).

Finally, the third difference between our method for cascade merging in a dag and our sorting algorithm is in our token-passing method. Our strategy for token passing is necessarily different, for some nodes in G may have two out-nodes. Moreover, an element e may be present in several (i.e., k) different lists in the same super-node of T. We still pass exactly one token with each element that we send from a node v to one of its parents. Thus, we may need to send two tokens for the elements in some $L_t(v)$ lists. We deal with these difficulties by basing our token-passing strategy on a "token stealing" paradigm. For each such element, e in $L_t(v)$, we "steal" a token from one of e's neighbors in $A_t(v)$. And, so long as v is not full, we maintain that the neighbor of e's twin is deficient one token (by passing a token "down the line" any time a new element becomes the neighbor of e's twin). Of course, once v becomes full we must "give back" the token that we stole for e. To accommodate this restitution, for each super node V_i, we select one copy of each element e in an $A_t(v)$ list, with $v \in V_i$, to be the *representative* copy of e at V_i. Thus, even though there may be k copies of an element e in the lists for V_i, we have one that is distinguished. Initially (i.e., at $t = 0$), of course, there is only one copy of any element e, and this copy is stored in a unique source node for e. Thus, at $t = 0$ each element is a representative copy of itself and there are no other copies. We assign $2k$ tokens to this copy.

We modify our merging procedure so as to maintain $L_t(v) \cup L_t(w)$ for each pair of nodes v and w in a super node V_i. We can use the procedures of the standard sorting methods to maintain these lists. The elements in these unions are not sent anywhere, so we do not need to worry about assigning any tokens (i.e., processors) to them—we simply use the token for e in $L_t(v)$ to maintain e's copy in each union $L_t(v) \cup L_t(w)$. This adds an overhead of $O(k) = O(1)$ time per stage, and implies that each cell for e must have $O(k) = O(1)$ additional pointers. The purpose of these extra unions is to allow the processor assigned to the representative of an element e to quickly locate all the other copies of e in lists at V_i. In particular, once all the nodes at V_i become full, this processor can locate all the other copies of e in lists at V_i in $O(k) = O(1)$ time. Let $C_i(e)$ denote the set of copies of e in lists at V_i. Once we have determined $C_i(e)$, we then make two passes through $C_i(e)$. In this first pass we collect a token from each copy of e that was formed by contracting two copies of e. Since at most one of any two contracted copies of e can be in an $L_t(v)$, this first pass will collect a single token for each contracted copy of e. In the second pass we distribute an extra token to each copy of e that is at a node v with two parents. If this copy of e is already in $L_t(v)$, then e can use this extra token to "pay back" its neighbor. If e is not in $L_t(v)$, then e can use this extra token to avoid stealing a token when e is eventually placed into $L_{t^*}(v)$ (for some $t^* > t$). We then let the representative copy store any unneeded tokens. In the stage that the nodes at the parent, V_j, of V_i are to become full we pass these extra tokens from the representative for e at V_i to a copy of e in some list at V_j, and make that copy the new representative of e (at V_j). The following lemma establishes that this scheme will allow us to always pass a token any time we copy an element e to a new list. In short, we can maintain our token-passing strategy.

Lemma 4.6: *When a super-node V_j becomes full, for each element e in a list at V_j, the representative for e has $2k - m_e$ extra tokens, where m_e is the number of copies of e at lists at V_j before compression.*

Proof: The proof is by induction on the levels of T. The base case is for V_j being a leaf in T. In this case there is only one copy of any element e in a list at V_j, and the (unique) copy of e has $2k$ tokens assigned. Thus, we have 1 copy of e and $2k - 1$ extra tokens. For the inductive step, suppose the claim is true for the children of V_j, and consider the stage t when V_j becomes full. Note that in stage $t - \lceil \log c_2 \rceil$ the children of V_j became full. Let V_i be the child of V_j that stores copies of e. By induction, the representative of e at V_i had $2k - m'_e$ extra tokens when V_j became full, where m'_e was the number of copies of e in lists at V_i before contracting. Thus, V_i had to redistribute $m_e - m'_e$ tokens from the representative in order to send all m_e copies of e to V_j. After this redistribution, the representative for e at V_j would have

$2k - m'_e - (m_e - m'_e)$ tokens, and these would be transferred to the representative of e at V_i. Noting that $2k - m'_e - (m_e - m'_e) = 2k - m_e$ completes the proof. □

Noting that $m_e \le 2k$ (by Lemma 4.5), the above lemma implies that the number of extra tokens at the representative will always be non-negative. Thus, there will always be a sufficient number of tokens to pass a token with each element sent from a child list to a parent list. This implies that we can implement the entire computation with $O(kn) = O(n)$ processors. In addition, once two nodes x and y with the same out-node, v, become full, v will become full $O(1)$ stages later. Thus, since we have already seen that each stage can be implemented in $O(1)$ time, this implies that the total time for the entire computation is $O(k\, height(T)) = O(height(T))$ time. This gives us Theorem 4.1. In the next subsection we give an important application of this theorem.

4.3 Application: Set-Expression Evaluation

Suppose one is given an expression tree T such that all the operands (stored at its leaves) are singleton sets and the allowable operations at internal nodes are union ($\cup$) and intersection ($\cap$). Recall that the *set-expression evaluation problem* is to produce the set determined by T, listed in sorted order. For example, T could be the tree illustrated in Figure 6.

Note that the set-expression evaluation problem contains the sorting problem as a special case, but also includes problem instances where expression tree T can have $O(n)$ height and can have many paths with alternating intersection and union operations. Thus, this problem has an $\Omega(n \log n)$ sequential lower bound in the comparison model. In fact, even if we do not insist on the output being sorted, this problem still has an $\Omega(n \log n)$ lower bound (in the ACT model), by a simple reduction from the set equality problem, which was shown to have and $\Omega(n \log n)$ lower bound in this model by Ben-Or [5].

We can solve the set-expression evaluation problem by cascade merging in a dag with a bounded-width tree partition. Specifically, we convert the tree T into such an expression dag G using the rake-and-compress paradigm of Abrahamson *et al.* [1], Miller and Reif [21, 22], and Kosaraju and Delcher [19]. This dag will have a tree partition with $O(\log n)$ height and have width equal to 3, with each node being labeled by an intersection or union operation.

We use these properties to perform a cascaded merge procedure in G, as if each

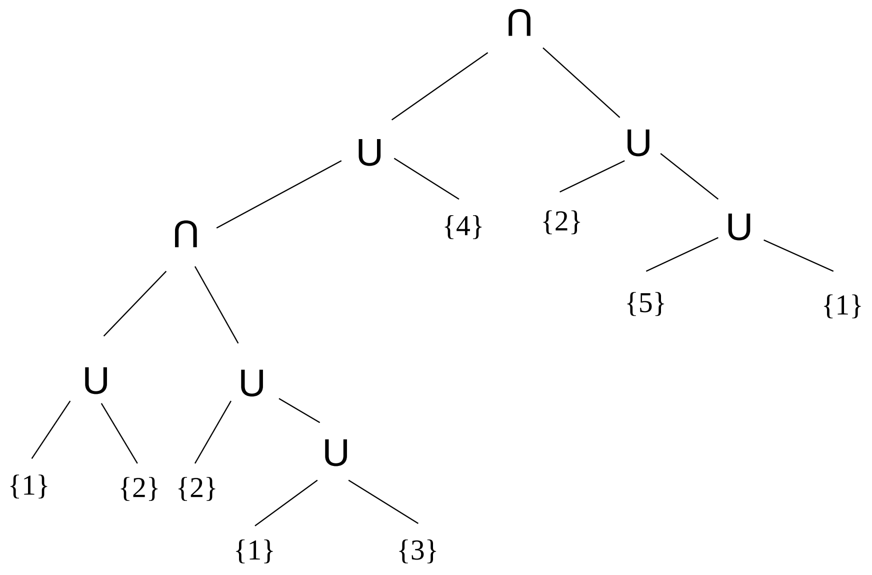

Figure 6: **An example expression tree.** In this case the evaluation of T gives the list $(1, 2)$.

internal-node operation were ∪. We maintain a flag $alive(e)$ for each element e in the list at a full node v, where $alive(e)$ is true if and only if e belongs in the set defined by the sub-dag "rooted" at v (when intersections are also taken into consideration). Thus, we implicitly remove elements during the cascading, but do not actually remove them until the cascade merging has completed.

4.3.1 Using a rake-and-compress scheme to build a tree partition for G

For completeness, we give the details for converting T into G using the rake-and-compress paradigm. This paradigm provides a procedure that iteratively "shrinks" T in a series of "rounds" to reduce T eventually to a single node. In each round there are two types of operations, a rake operation and a compress operation, that are applied to the nodes of T. A rake operation applied at a node v amounts to removing any children of v that are leaves. A compress operation applied at a node v amounts to contracting v with its child, provided v has only one child. We refer the reader to Abrahamson *et al.* [1], Miller and Reif [21, 22], and Kosaraju and Delcher [19] for the details on how to use these two operations to reduce a constant fraction of the nodes in T in each round. The only fact we need from these papers is that we can shrink T to a single node in $O(\log n)$ rounds and $O(n)$ work, where each round is implemented using rake and compress operations.

Such a scheme can alternatively be viewed as a method for converting T into an equivalent dag G, where the sources of G are the leaves of T and each non-source node v of G is labeled with an operation in $\{\cup, \cap\}$, so that the value of v is defined by applying this operation to the sets at v's in-nodes. The sink node in G represents the same set as the root of T. In each application of a rake or compress operation we "shrink" T. In each intermediate "snap shot" of T, for each node v we will store a dag $G(v)$. If v has two children, then $G(v)$ is empty. If v has one child, then the dag $G(v)$ has two sink nodes a and b that represent two sets A and B, respectively, such that if the set for v's child is X, then the set for v is $A \cap X \cup B$. If the set is $X \cup B$, then the a node of the dag will be specially marked as the *identity*. Finally, if v is a leaf, then the dag $G(v)$ has a single sink node that represents the set for v. Thus, initially, $G(v)$ is empty for each internal node v in T, and $G(v)$ is a single node that represents the singleton set at v for each leaf v of T. Let us consider how the $G(v)$'s change for each type of shrinking operation:

1. *Raking a leaf w into a node v with two children (w is a child of v):* Let u be v's other child. In this case we create $G(v)$ depending on the operation to be performed at v:

   (a) If $op(v) = \cup$, then if Y is the set for w and X is the set for u, then the set for v is $X \cup B$, where $B = Y$. Thus, we construct $G(v)$ so that its b node corresponds to the root of $G(w)$ and $G(v)$'s a node is an "identity" node.

   (b) If $op(v) = \cap$, then if Y is the set for w and X is the set for u, then the set for v is $A \cap X$, where $A = Y$. Thus, we construct $G(v)$ so that its a node corresponds to the root of $G(w)$ and $G(v)$'s b node is an "empty set" node.

2. *Raking a leaf w into a node v with one child (w is the child of v):* Let x be the root of $G(w)$ and let X denote the set that x represents. In a previous raking step we must have raked the other child of v. Thus, $G(v)$ has two sink nodes a and b that correspond to sets A and B, respectively, so that $A \cap X \cup B$ is the set at v. So to update $G(v)$ we create two new nodes i and u with $op(i) = \cap$ and $op(u) = \cup$, and let a and x be the in-nodes of i and let i and b be the in-nodes of b.

3. *Compressing a node w with one child into a node v with one child (w is the child of v):* Let u be w's child, let X denote the set for u, and let Y denote the set for w. Thus, $G(w)$ has two nodes a_w and b_w representing sets A_w and B_w so that $Y = A_w \cap X \cup B_w$. Similarly, $G(v)$ has two nodes a_v and b_v representing sets A_v and B_v so that the set for v is $A_v \cap Y \cup B_v$. Therefore, the new set for v should be $(A_v \cap A_w) \cap X \cup (A_v \cap B_w \cup B_v)$. So to update $G(v)$ we create three new nodes a'_v, i, and b'_v, with $op(a'_v) = \cap$, $op(i) = \cap$, and $op(b'_v) = \cup$. In addition, a'_v has in-nodes a_v and a_w, i has in-nodes a_v and b_w, and b'_v has in-nodes i and b_v.

When we complete the rake-and-compress computation the tree T will be reduced to a single node r (the root of T), and the graph $G = G(r)$ will have a single sink node that represents the set for r. Thus, to construct this set we must perform a cascade merge procedure in G. The following lemma shows that we can use our cascade merging method for dags:

Lemma 4.7: *G has a tree partition Π with width equal to 3, $T(\Pi)$ is a rooted binary tree, and the in-degree or out-degree of any node in G is at most* 2.

Proof: The proof is by induction on the number of rake-and-compress steps. Initially, each $G(v)$ is a single node; hence, the three properties trivially hold. Otherwise, note that any rake or compress operation applied to a single node adds at most three nodes to create a new dag $G(v)$ from some $G(w)$ dag(s). The only in-nodes for these three are the former sink nodes for the $G(w)$ dag(s), and these former sink nodes are not used as in-nodes for any nodes in G. Moreover, the underlying super-node for each collection of newly created nodes has at most two children, and the in-degree and out-degree of the newly created nodes are all at most 2. □

4.3.2 Cascade merging in the tree partition produced by the rake-and-compress procedure

We have yet, then, only to show how to use a cascade merging procedure in G to solve the set-expression evaluation problem. Our method is to perform the cascade merging in a dag, as described above, imagining that each node in G is simply labeled with a union operation. Since some nodes in G may actually be labeled with intersection operations, however, we perform one extra computation each time a node v, with two in-nodes x and y, becomes full. The computation we perform for v depends on the

actual operation, $op(v)$, that labels v:
Case 1: $op(v) = \cup$. In this case, for each $e \in A_t(v)$, we mark e "alive" if and only if there is an "alive" copy of e in $L_t(x)(= A_t(x))$ or $L_t(y)(= A_t(y))$.
Case 2: $op(v) = \cap$. In this case, for each $e \in A_t(v)$, we mark e "alive" if and only if there is an "alive" copy of e in $L_t(x)$ and an "alive" copy of e in $L_t(y)$.

Note that this extra computation clearly adds only $O(1)$ extra steps to each stage in the cascade merging procedure for G.

By a simple inductive argument and the semantics of our labeling computation, it follows that an element will be marked "alive" if and only if it belongs in the list for v in G (when intersections are taken into consideration). Moreover, each copy of an element e in an $A_t(v)$ list will have the same label (this also follows by induction). Thus, by Theorem 4.1, after $O(\log n)$ stages the sink of G will be full, and the "alive" elements in $A_t(r)$ are exactly the elements that belong to the solution to the set-expression evaluation problem for T (where r is the root). Also, the total time needed to do this is $O(\log n)$ using $O(n)$ processors. We form the answer then, by compressing out the "dead" elements of $A_t(r)$ in $O(\log n)$ time using $O(n)$ processors, by a simple list-ranking procedure. This gives us the following theorem:

Theorem 4.8: *Given an expression tree T whose operands are singletons and whose operations come from the set $\{\cup, \cap\}$, one can evaluate T in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PPM model.*

5 Sorting on an EREW PPM

In this section we outline a simple EREW PPM sorting method, based on generalizations of the methods of the previous sections. As in our CREW PPM sorting method, we let T be a complete binary tree such that each of its leaves is associated with one of the elements in the input set S. Our algorithm again proceeds in stages, with each node v in T storing lists $A_t(v)$, $L_{t-1}(v)$, $L'_t(v)$, and $L_t(v)$ at the end of stage t. The difference here is that the list $A_t(v)$ is the sorted merge of samples sent from all the nodes adjacent to v in T, including v's parent, u. Thus, we allow some elements to be cascading down the tree T while others are cascading up. This approach is similar in spirit to the EREW PRAM sorting method of Cole [10], but is considerably different otherwise.

Specifically, we define $A_t(v)$, for each non-full node v, as

$$A_t(v) = L_{t-1}(u) \cup L_{t-1}(x) \cup L_{t-1}(y).$$

Changing the definition of $A_t(v)$ in this way has some significant impacts on our procedure. Before we make this more precise, we give some generalizations to the merge lemmas we needed for our CREW PPM sorting procedure.

5.1 Generalizing the Merge Lemmas

We must generalize the merge lemmas of Section 2 in order to be used in our EREW PPM sorting algorithm.

Lemma 5.1: *Suppose one is given three lists A, B, and C, such that $B \subseteq A$, B is a c_1-cover of A and a c_2-cover of C, for constants c_1 and c_2. If one has $A \xrightarrow{\text{pred}} B$ and $B \xrightarrow{\text{pred}} C$, then one can compute $A \xrightarrow{\text{pred}} C$ in $O(1)$ time with a processor assigned to each element in A in the EREW PPM model.*

Proof: Let a be an element in A. The method is to simulate the proof of Lemma 2.3 so as to avoid concurrent reads. Recall that the method of that proof was to follow the pointer from a to its predecessor, b, in B, follow the pointer from b to its predecessor, c, in C, and, from there, march at most c_2 positions to the predecessor of a in C. We can simulate each step of this process in $O(\log c_1) = O(1)$ time by using fan-out broadcasting to implement what would otherwise be concurrent reads. Thus, we can still perform the entire computation in $O(c_2) = O(1)$ time. $\square$

The following lemma generalizes the c-Cover Merge Lemma to the EREW PPM model, and makes three applications of Lemma 5.1.

Lemma 5.2: (The EREW c-Cover Merge Lemma) *Suppose one is given lists A, L_1, L_2, L_3, L_1^*, L_2^*, and L_3^*, such that $A = L_1 \cup L_2 \cup L_3$, and L_i is a c-cover of L_i^* and a d-cover of A, for $i = 1, 2, 3$, where c and d are constants. In addition, suppose one is given $A \xrightarrow{\text{pred}} L_i$ and $L_i \xrightarrow{\text{pred}} L_i^*$, for $i = 1, 2, 3$. Then one can compute $A^* = L_1^* \cup L_2^* \cup L_3^*$ (with $A^* \xrightarrow{\text{pred}} L_i^*$ for $i = 1, 2, 3$) in $O(1)$ time with a processor assigned to each element in A in the EREW PPM model.*

Proof: By three applications of Lemma 2.3, one can compute $A \xrightarrow{\text{pred}} L_i^*$, for $i = 1, 2, 3$, in $O(1)$ time. Let a and b be two consecutive elements in A, and let $L_1^*(a, b)$, $L_2^*(a, b)$, and $L_3^*(a, b)$ be the respective L_i^* sublists that fall in the interval $[a, b)$. In parallel for each such pair a, b one can construct the portion of A^* that falls in the interval $[a, b)$ by merging the three $L_i^*(a, b)$ lists by a simple three-way sequential merge. The total time is $O(c + d) = O(1)$, given a processor assigned to each element in A. $\Box$

We next generalize the Conforming-Merge Lemma.

Lemma 5.3: (The EREW Conforming-Merge Lemma) *Suppose one is given lists A, L_1, L_2, L_3, $\hat{L}_1$, $\hat{L}_2$, and $\hat{L}_3$, such that $A = L_1 \cup L_2 \cup L_3$, L_i is a c-cover for A, L_i d-conforms to $\hat{L}_i$, for $i = 1, 2, 3$, where c and d are constants. Suppose further that one is given $A \xrightarrow{\text{pred}} L_i$ and $L_i \xrightarrow{\text{rank}} \hat{L}_i$, for $i = 1, 2, 3$. Then one can compute $A' = \hat{L}_1 \cup \hat{L}_2 \cup \hat{L}_3$ (with A' predecessor-linked into each $\hat{L}_i$) and $A \xrightarrow{\text{rank}} A'$ in $O(1)$ time with a processor assigned to each element in A in the EREW PPM model.*

Proof: First, note that one can apply the EREW c-Cover Merge Lemma to compute A', with A' predecessor-linked into each $\hat{L}_i$, in $O(1)$ time. Thus, we have yet to show how to compute $A \xrightarrow{\text{rank}} A'$, by determining, for each element a in A, the twin, a' of a in A'. Let i denote the rank of a in A. Since $A = L_1 \cup L_2 \cup L_3$, there is a copy of a in one of the L_i lists; without loss of generality, suppose $a \in L_3$. Let $\hat{a}$ be the twin of a in $\hat{L}_3$ (with respect to the copy of a in L_3). To determine the i-th element of A' it is sufficient to determine the rank of $\hat{a}$ in A', described in terms of i. Let a_1 be the predecessor of a in L_1, and let $\hat{a}_1$ be the twin of a_1 in $\hat{L}_1$. Similarly define a_2 and $\hat{a}_2$ with respect to L_2 and $\hat{L}_2$. Suppose a has rank j in L, a_1 has rank k_1 in L_1 and a_2 has rank k_2 in L_2, so $i = j + k_1 + k_2$. Note that $\hat{a}$ has rank j in $\hat{L}$, by definition. So we need only determine the rank of $\hat{a}$ in $\hat{L}_1$ (in terms of k_1) and the rank of $\hat{a}$ in $\hat{L}_2$ (in terms of k_2). Let us restrict our attention to $\hat{L}_1$. Since L_1 d-conforms to $\hat{L}_1$, the predecessor of $\hat{a}$ in $\hat{L}_1$ can be at most $d + 1$ positions away from $\hat{a}_1$; hence, we can determine the (signed) difference between the rank of $\hat{a}$ in $\hat{L}_1$ and the rank of $\hat{a}_1$ in $\hat{L}_1$ just by counting the number of elements between $\hat{a}$'s predecessor in $\hat{L}_1$ and $\hat{a}_1$. Let $-d - 1 \leq l_1 \leq d + 1$ denote this difference, so that the rank of $\hat{a}$ in $\hat{L}_1$ is $k_1 + l_1$. By a similar method we can determine the rank of $\hat{a}$ in $\hat{L}_2$ as $k_2 + l_2$, with $-d - 1 \leq l_2 \leq d + 1$. Then the rank of $\hat{a}$ in A' is $j + k_1 + k_2 + l_1 + l_2 = i + l_1 + l_2$. Therefore, to determine the i-th element of A' we need only march $|l_1 + l_2| = O(d)$ positions in A' from $\hat{a}$'s position in A'. As in the proof for the EREW c-Cover Lemma, we can avoid concurrent reads by broadcasting values to all the processors that need them. Since each L_i is a c-cover of A, each step in this method can be implemented in the EREW PPM model in $O(\log c) = O(1)$ time. Thus, the entire procedure requires only $O(d) = O(1)$ time, given a processor assigned to each element in A. $\Box$

Having generalized the important lemmas in our method, we are now ready to present our procedure for sorting in the EREW PPM model in $O(\log n)$ time using $O(n)$ processors.

5.2 EREW Stage Invariants and Computations

As mentioned earlier, we changed the definition of $A_t(v)$ to be the merge of the samples at nodes adjacent to v (not just from v's children). One of the most important implications of this new definition is that, as we will show (in Lemma 5.7), it forces $A_t(v)$ to be a c-cover of $L_t(u)$ and $L_t(u)$ to be a c-cover of $A_t(v)$, for some constant c. This allows us to use the EREW merge lemmas of the previous subsection to perform all the merges needed without using concurrent reads. Another significant impact of our $A_t(v)$ definition is that it requires us to modify the token assignment method so that tokens can be passed from parent to child, as well as from child to parent. This complication seriously affects our token-passing strategy, as we must take care to maintain an $O(n)$ number of tokens, in spite of there being many cycles in the definitions of the $A_t(v)$'s. Before we describe the modifications to our token-passing strategy, however, let us give the steps of our procedure. We begin with our list of invariants.

EREW List Invariants at the end of stage t: *Each active node v stores four sorted lists, $A_t(v)$, $L_{t-1}(v)$, $\hat{L}_{t-1}(v)$, and $L_t(v)$, as doubly-linked lists that satisfy the following properties:*

1. *If v was not full after stage $t-1$, then $A_t(v) = L_{t-1}(u) \cup L_{t-1}(x) \cup L_{t-1}(y)$, and $A_t(v)$ is predecessor-linked into each of $L_{t-1}(u)$, $L_{t-1}(x)$, and $L_{t-1}(y)$. Otherwise, if v was full after stage $t-1$, then $A_t(v) = A_{t-1}(v)$.*

2. *$L_{t-1}(v)$ 1-conforms to $\hat{L}_{t-1}(v)$, and is rank-linked into $\hat{L}_{t-1}(v)$.*

3. *$\hat{L}_{t-1}(v)$ is a $(0,1)$-sample of $L_t(v)$, and is predecessor-linked into $L_t(v)$.*

4. *If v is not full, then $L_t(v)$ is a (c_1, c_2)-sample of $A_t(v)$, where c_1 and c_2 are constants (fixed for the entire procedure). If v is full, then $L_t(v)$ is a $(\lfloor c_1/2^i \rfloor, \lfloor c_2/2^i \rfloor)$-sample, where i is the number of stages since v first became full. In any case $L_t(v)$ is predecessor-linked into $A_t(v)$.*

Note that List Invariants 2, 3, and 4 are the same as in the CREW algorithm, and List Invariant 1 is similar to the corresponding CREW invariant. Nevertheless, these simple modifications will allow us to implement the entire procedure in the EREW PPM model. Before we give the details of our implementation, then, let us show that these invariants give us the pre-conditions necessary for the EREW merge lemmas. We begin by making two additional observations regarding c-covers.

Observation 5.4: *If $B \subseteq A$, then A is a 1-cover of B.*

Observation 5.5: *If A is a c_1-cover of B and B is a c_2-cover of C, then A is a c_1c_2-cover of C.*

Having made these observations, we show, in the next two lemmas, that the sample at a child is a c-cover of the list at the parent, and vice versa.

Lemma 5.6: *For any node v, $L_{t-1}(v)$ is a 4-cover of $L_t(v)$.*

Proof: Note that, by List Invariant 2, $L_{t-1}(v)$ 1-conforms to $\hat{L}_{t-1}(v)$; hence, by Observation 2.2, $L_{t-1}(v)$ is a 2-cover of $\hat{L}_{t-1}(v)$. Also note that, by List Invariant 3, $\hat{L}_{t-1}(v)$ is a $(0,1)$-sample of $L_t(v)$; hence, by Observation 2.1, $\hat{L}_{t-1}(v)$ is a 2-cover of $L_t(v)$. Our lemma, then, follows immediately, by Observation 5.5. $\square$

Lemma 5.7: *$A_t(v)$ is a c-cover of $L_t(u)$ and $L_t(u)$ is a c-cover of $A_t(v)$, where c is a constant.*

Proof: By Observation 5.4 and the previous lemma, if v is not full, then $A_t(v)$ is a 4-cover of $L_t(u)$, for $L_{t-1}(u) \subset A_t(v)$ by List Invariant 1 at v. If, on the other hand, v became full in stage t^*, then $L_{t^*-1}(u) \subset A_{t^*}(v) = A_t(v)$, by List Invariant 1 at v for stages $t^*, t^*+1, \ldots, t^*+d = t$, where d is at most $\lceil \log c_2 \rceil = O(1)$ (by List Invariant 4 at v). By a repeated application of the previous lemma, and Observation 5.5, $L_{t^*-1}(u)$ is a 4^d-cover of $L_t(u)$. Therefore, $A_t(v)$ is at worst a c-cover of $L_t(u)$, if v is full, where c is the constant $4^{\lceil \log c_2 \rceil}$. The proof that $L_t(u)$ is a c-cover of $A_t(v)$ follows, by a repeated application of Observation 5.5, from the following observations. $L_t(u)$ is a (c_2+1)-cover of $A_t(u)$, by List Invariant 4 at u. $A_t(u)$ is a 1-cover of $L_{t-1}(v)$, since $L_{t-1}(v) \subset A_t(u)$, by List Invariant 1 at u. $L_{t-1}(v)$ is a 4-cover of $L_t(v)$, by the previous lemma. Finally, $L_t(v)$ is a (c_2+1)-cover of $A_t(v)$, by List Invariant 4 at v. $\square$

The steps of our method are also quite similar to the steps in our CREW procedure. Nevertheless, there are a number of important differences, which we highlight in the discussion below.

Stage $t+1$ Computation: *In Step 1, for each active node v, we construct temporary lists $A'_t(v)$ and $L'_t(v)$ and additional linkages to aid us in Step 2. In Step 2 we construct $A_{t+1}(v)$, in Step 3 we construct $\hat{L}_t(v)$, and in Step 4 we construct $L_{t+1}(v)$.*

Step 1: If v is not full, then we construct a temporary list $A'_t(v) = \hat{L}_{t-1}(u) \cup \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$ and predecessor-link $A'_t(v)$ into $\hat{L}_{t-1}(u)$, $\hat{L}_{t-1}(x)$, and $\hat{L}_{t-1}(y)$. If v is full, then we simply take $A'_t(v) = A_t(v)$. In either case, we let $L'_t(v)$ be the sample of $A'_t(v)$ parallel to $L_t(v)$ in $A_t(v)$ (note that $|A'_t(v)| = |A_t(v)|$), and we compute $L_t(v) \xrightarrow{\text{rank}} L'_t(v)$. We can implement this step in $O(1)$ time by the EREW Conforming-Merge Lemma.

Comment: Since $L'_t(v)$ and $L_t(v)$ are parallel samples, $L'_t(v)$ is a (c_1, c_2)-sample of $A'_t(v)$, for $L_t(v)$ is a (c_1, c_2)-sample of $A_t(v)$ (by List Invariant 4).

Step 2: If v is not full, then we construct $A_{t+1}(v) = L_t(u) \cup L_t(x) \cup L_t(y)$, by the EREW c-Cover Merge Lemma. We satisfy the preconditions for applying this lemma by the computations of the previous step and by List Invariant 3 at nodes u, x, and y. If v is full, then we simply take $A_{t+1}(v) = A'_t(v)$.

Comment: This gives us List Invariant 1 at v for after stage $t+1$. In addition, from List Invariant 2 at v's children, it is easy to see that $L'_t(v)$ is a $(c_1, 2c_2 + 3)$-sample of $A_{t+1}(v)$ (if v is full, then it is a (c_1, c_2)-sample).

Step 3: If v is not full, then we locally shift $L'_t(v)$ in $A_{t+1}(v)$ to construct a $(c_1, 2c_2 + 1)$-sample, $\hat{L}_t(v)$, of $A_{t+1}(v)$. We also rank-link $L'_t(v)$ into $\hat{L}_t(v)$, which implicitly ranks $L_t(v)$ into $\hat{L}_t(v)$. If v is full, then we take $\hat{L}_t(v) = L'_t(v)$.

Implementation: Let $b \in A_{t+1}(v)$ be an element in $L'_t(v)$. Since b is in $L'_t(v)$, it is also in $A'_t(v)$. Thus, since $A'_t(v) = \hat{L}_{t-1}(u) \cup \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$ and $A'_t(v)$ is predecessor-linked into $\hat{L}_{t-1}(u)$, $\hat{L}_{t-1}(x)$, and $\hat{L}_{t-1}(y)$, we can easily determine b's predecessor in $\hat{L}_{t-1}(u)$, $\hat{L}_{t-1}(x)$, and $\hat{L}_{t-1}(y)$, respectively, one of which must be b itself. Without loss of generality, assume that $b \in \hat{L}_{t-1}(y)$. Let e and g be the two consecutive elements of $\hat{L}_{t-1}(x)$ such that b is in the interval (e, g), and let h and k be the two consecutive elements of $\hat{L}_{t-1}(u)$ such that b is in the interval (h, k). By List Invariant 3 at u and x, we can easily determine the elements, f and j, if they exist, in $L_t(x)$ and $L_t(u)$, respectively such that f is in the interval (e, g) and j is in the interval (h, k). These are the values that determine how we perform our local shift relative to b. In particular, if $f < b$ and $j < b$, then we make the immediate predecessor of b in $A_{t+1}(v)$ take b's place in $\hat{L}_t(v)$. If $b < f$ and $b < j$, then we make the immediate successor of b in $A_{t+1}(v)$ take b's place in $\hat{L}_t(v)$. If either f or j do not exist, then we use the convention that the missing element is less than b. If none of these conditions are satisfied for b, then we let the copy of b in $A_{t+1}(v)$ also be in $\hat{L}_t(v)$—so that b is its own twin in $L_t(v)$. We show below, in Lemma 5.8, that this simple rule forces $\hat{L}_t(v)$ to be a $(c_1, 2c_2+1)$-sample of $A_{t+1}(v)$. Note that, as in the previous two steps, by Lemma 5.7, we can avoid concurrent reads in this step by an $O(1)$-time broadcasting procedure.

Step 4: We refine $\hat{L}_t(v)$ into $L_{t+1}(v)$. We place each element of $\hat{L}_t(v)$ in $L_{t+1}(v)$, and, for each sublist $B_v(e)$ of $A_{t+1}(v)$ consisting of all elements that are strictly between two consecutive elements e and f of $\hat{L}_t(v)$, we make the median of $B_v(e)$ also be a member of $L_{t+1}(v)$, provided v is full or $|B_v(e)| > c_2$. This completes the computation for Stage $t+1$.

Comment: After completing Step 4 for stage $t+1$, List Invariant 3 holds, since at most one element of $L_{t+1}(v)$ exists in between any two elements of $\hat{L}_t(v)$. Moreover, if v was previously not full, then this forces $L_{t+1}(v)$ to be a (c_1, c_2)-sample of $A_{t+1}(v)$ (assuming $\hat{L}_t(v)$ is a $(c_1, 2c_2+1)$-sample and $c_1 \geq \lfloor (c_2+1)/2 \rfloor$), which satisfies List Invariant 4. If, on the other hand, v was previously full, then this forces $L_{t+1}(v)$ to be a $(\lfloor c_1/2 \rfloor, \lfloor c_2/2 \rfloor)$-sample of $A_{t+1}(v)$.

End of Procedure.

Thus, when Step 4 completes, it should be apparent that we will have satisfied List Invariants 1 and 3. In the next subsection we show that our sample-refining method of Step 4 will correctly give us List Invariants 2 and 4.

5.3 Analysis of our EREW Sorting Algorithm

We begin our analysis by showing that List Invariant 2 is satisfied after Stage $t+1$ completes.

Lemma 5.8: *$\hat{L}_t(v)$ is a $(c_1, 2c_2+1)$-sample of $A_{t+1}(v)$.*

Proof: Recall that $L'_t(v)$ is a (c_1, c_2)-sample of $A'_t(v)$, that $A'_t(v) = \hat{L}_{t-1}(u) \cup \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$, and that $\hat{L}_{t-1}(u)$ (resp., $\hat{L}_{t-1}(x)$, $\hat{L}_{t-1}(y)$) is a $(0,1)$-sample of $L_t(u)$ (resp., $L_t(x)$, $L_t(y)$). Thus, $L'_t(v)$ is a $(c_1, 2c_2+3)$-sample of $A_{t+1}(v)$. Let $I = (a, b)$ be an interval defined by two consecutive elements, a and b, in $L'_t(v)$. So the number of elements of $A_{t+1}(v)$ in I, which we denote by c, is at most $2c_2+3$. Let $\hat{I}$ denote the interval $(\hat{a}, \hat{b})$ defined by the twins, $\hat{a}$ and $\hat{b}$, of a and b, respectively, in $\hat{L}_t(v)$. Then, if we let $\hat{c}$ denote the number of elements of $A_{t+1}(v)$ in $\hat{I}$, we wish to show that $\hat{c} \leq 2c_2+1$.

Since both a and b are in $L'_t(v)$, they both are also in $A'_t(v) = \hat{L}_{t-1}(u) \cup \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$. Let α_1 and α_2 denote the two nodes adjacent to v (i.e., two of u, x, or y) such that a is not in $\hat{L}_{t-1}(\alpha_1)$ or $\hat{L}_{t-1}(\alpha_2)$. Also let e_i and g_i be the two consecutive elements of $L_{t-1}(\alpha_i)$ such that a is in the interval (e_i, g_i), $i = 1, 2$, and let f_i denote the element (if it exists) of $L_t(\alpha_i)$ in this same interval. Similarly, let β_1 and β_2 denote the nodes adjacent to v such that b is not in $\hat{L}_{t-1}(\beta_1)$ or $\hat{L}_{t-1}(\beta_2)$, and let p_i and r_i be the two consecutive elements of $L_{t-1}(\beta_i)$ such that b is in the interval (p_i, r_i), and let q_i denote the element (if it exists) of $L_t(\beta_i)$ in this same interval, $i = 1, 2$. We distinguish a number of cases for the relationships between the elements f_1, f_2, q_1, and q_2 and the elements a and b. To simplify the discussion, we say "a moves right" (resp., "a moves left") to indicate that the successor (resp., predecessor) of a in $A_{t+1}(v)$ is the twin of a in $\hat{L}_{t+1}(v)$, and we say "a stays in place" to indicate that a is its own twin in $\hat{L}_{t+1}(v)$. We also use similar expressions for b. Without loss of generality, we assume that f_1, f_2, q_1, and q_2 all exist, since we use the convention that a missing element is less than a or b, respectively.

1. All of f_1, f_2, q_1, and q_2 lie in I. Then $c \leq 2c_2 + 3$. In this case a moves right and b moves left. Thus, $\hat{c} \leq 2c_2 + 1$.

2. Three of f_1, f_2, q_1, or q_2 lie in I and one lies outside I. Then $c \leq 2c_2 + 2$. In this case either a stays in place and b moves left or a moves right and b stays in place. In either case, $\hat{c} = 2c_2 + 1$.

3. Two of f_1, f_2, q_1, or q_2 lie in I and two lie outside I. We distinguish several subcases:

   (a) f_1 and f_2 are outside. Then $c \leq 2c_2 + 1$, and a and b both move left. Thus, $\hat{c} \leq 2c_2 + 1$.

   (b) f_i and q_j are outside, where i and j are either 1 or 2. Let us first address the degenerate case where $i = j$ and $f_i = q_j$. In this degenerate case, $c \leq 2c_2 + 2$, but either a stays in place and b moves left or a moves right and b stays in place; hence, $\hat{c} \leq 2c_2 + 1$. In the general case, when $f_i \neq q_j$, then $c \leq 2c_2 + 1$ and both a and b stay in place. Thus, $\hat{c} \leq 2c_2 + 1$.

   (c) q_1 and q_2 are outside. Then $c \leq 2c_2 + 1$, and a and b both move right. Thus, $\hat{c} \leq 2c_2 + 1$.

4. Three of f_1, f_2, q_1, or q_2 lie outside of I and one lies inside I. Then $c \le 2c_2$. In this case either a stays in place and b moves right or a moves left and b stays in place. In either case, $\hat{c} = 2c_2 + 1$.

5. f_1, f_2, q_1, or q_2 all lie outside of I. Because of possible degeneracies, we distinguish several subcases:

   (a) $f_1 = q_1$ and $f_2 = q_2$. Then $c \le 2c_2 + 1$. If f_1 and f_2 are both greater than b, then a and b both move right. If f_1 and f_2 are both less than a, then a and b both move left. Otherwise, a and b both stay in place. In any case, $\hat{c} \le 2c_2 + 1$.

   (b) $f_1 \ne q_1$ and $f_2 = q_2$. Then $c \le 2c_2$. If f_2 is less than a, then a moves left and b stays in place, and if f_2 is greater than b, then a stays in place and b moves right. In either case, $\hat{c} \le 2c_2 + 1$.

   (c) $f_1 = q_1$ and $f_2 \ne q_2$. Similar to previous case.

   (d) $f_1 \ne q_1$ and $f_1 \ne q_2$. Then $c \le 2c_2 - 1$. In this case a moves left and b moves right. Thus, $\hat{c} \le 2c_2 + 1$.

This completes the proof. □

Thus, after Step 4 completes, List Invariant 4 is satisfied for each node in the tree. So we have only to prove that List Invariant 2 is satisfied after stage $t + 1$. The next lemma is essential to proving this.

Lemma 5.9: *$A_t(v)$ 3-conforms to $A'_t(v)$.*

Proof: If v is full, then $A_t(v) = A'_t(v)$; hence, the lemma is trivially true in this case. So suppose v is not full. Then $A_t(v) = L_{t-1}(u) \cup L_{t-1}(x) \cup L_{t-1}(y)$ and $A'_t(v) = \hat{L}_{t-1}(u) \cup \hat{L}_{t-1}(x) \cup \hat{L}_{t-1}(y)$. Let a_i denote the i-th element of $A_t(v)$ and let a'_i denote the i-th element of $A'_t(v)$ (i.e., a_i's twin in $A'_t(v)$), for $i \in \{1, 2, ..., |A_t(v)|\}$. Also, let $\hat{a}_i$ denote the twin in $\hat{L}_t(\alpha)$ of the copy of a_i in $L_{t-1}(\alpha)$, where α is x if a_i came from $L_{t-1}(x)$, α is y if a_i came from $L_{t-1}(y)$, and α is u otherwise. Suppose, for the sake of contradiction, there is an $i \in \{4, 5, ..., |A_t(v)|\}$ such that $a_i < a'_{i-3}$. Without loss of generality, suppose a_i came from $L_{t-1}(x)$. Note that each a'_j is an $\hat{a}_k$ for some k. By a pigeonhole argument similar to that used in the proof of Lemma 3.2, since a'_{i-3} is the $(i-3)$-rd element in $A'_t(v)$, there must be at least 3 elements a_f, a_g, and a_h with $a_f < a_g < a_h < a_i$ such that

$$a_i < \hat{a}_f, \ a_i < \hat{a}_g, \text{ and } a_i < \hat{a}_h. \tag{1}$$

Note that for any $\alpha \in \{u, x, y\}$, by List Invariant 2 at α (for stage $t-1$), $L_{t-1}(\alpha)$ 1-conforms to $\hat{L}_{t-1}(\alpha)$. This implies that none of a_f, a_g, and a_h came from $L_{t-1}(x)$, for, otherwise, this would contradict List Invariant 2 at x (for it would imply that the copy of a_i in $L_{t-1}(x)$ is not in the 1-neighborhood of $\hat{a}_i$ in $\hat{L}_{t-1}(x)$). Without loss of generality, suppose a_h came from $L_{t-1}(y)$. Note that (1) also implies that $a_h < \hat{a}_f$ and $a_h < \hat{a}_g$, since $a_h < a_i$. Thus, a_f and a_g must both come from u, or we would contradict List Invariant 2 at y (for, otherwise, the copy of a_h in $L_{t-1}(y)$ would not be in the 1-neighborhood of $\hat{a}_h$ in $\hat{L}_{t-1}(y)$). Finally, note that (1) also implies that $a_g < \hat{a}_f$, since $a_g < a_i$. But this contradicts List Invariant 2 at u. Thus, $\hat{a}_{i-3} \leq a_i$, for $i \in \{4, 5, ..., |A_t(v)|\}$. By a similar argument we also have that $a_i \leq \hat{a}_{i+3}$, for $i \in \{1, 2, ..., |A_t(v)| - 3\}$. Therefore, $A_t(v)$ 3-conforms to $A'_t(v)$. $\Box$

As a simple corollary to this lemma, we have that List Invariant 2 is satisfied, provided c_1 is sufficiently large:

Corollary 5.10: *If $c_1 \geq 3$, then $L_t(v)$ 1-conforms to $\hat{L}_t(v)$.*

Proof: As an immediate consequence of Lemma 5.9, we have that $a'_{i-4} < a_i < a'_{i+4}$, where a_i (resp., a'_i) denotes the i-th element of $A_t(v)$ (resp., $A'_t(v)$). Suppose a_i is in $L_t(v)$, and let a_{i+j} be the next element of $A_t(v)$ in $L_t(v)$ (i.e., the smallest element of $L_t(v)$ greater than a_i). Then a'_i and a'_{i+j} are consecutive elements in $\hat{L}_t(v)$. By List Invariant 4, if $c_1 \geq 3$, then $j \geq 4$. Thus, $L_t(v)$ 1-conforms to $\hat{L}_t(v)$. $\Box$

So, after Step 4 completes, we are ready to begin the next stage, as all the invariants have been satisfied. Also recall that we have placed some restrictions on the values of c_1 and c_2, namely $c_1 \geq \max\{3, \lfloor (c_2+1)/2 \rfloor\}$ and $c_2 \geq 2c_1$. These are not the only constraints we place on these values, however. In the next section we describe how we can generalize our token-passing strategy for implementing processor and space allocation. As it turns out, our new strategy will place some additional constraints on c_1. Nevertheless, c_1 will still be a small constant and $c_2 = 2c_1$. Thus, after $O(\log c_1 * \log n) = O(\log n)$ stages the root node becomes full, implying a total running time of $O(\log n)$.

5.4 Space and Processor Allocation

As in our CREW algorithm, our method for allocating space and processors is based on a token-passing scheme. This token-passing scheme is considerably more involved than in the CREW case, however. One of the complicating factors is that in our

EREW algorithm elements are not only sent up the tree, but some are also sent down the tree. To distinguish between these two situations we say that an element in $A_t(v)$ is *rising* if it came from $L_{t-1}(x)$ or $L_{t-1}(y)$ and is *falling* if it came from $L_{t-1}(u)$ (and we maintain similar labels in $A'_t(v)$). In fact, we further refine the rising elements into two subsets. We say that a rising element e in $A_t(v)$ is *genuine* if the copy of e in $L_{t-1}(\alpha)$ is also a rising element, where α is a child of v. A rising element e in $A_t(v)$ is *fake*, otherwise, i.e., if the copy of e in $L_{t-1}(\alpha)$ is a falling element.

Notation: *Given an element e in $A_{t-1}(v)$, we let $r_t(e)$ denote the copy in $A_t(v)$ of the twin in $A'_{t-1}(v)$ of e in $A_{t-1}(v)$.*

Intuitively, $r_t(e)$ is the "replacement" for e in $A_t(v)$. Our EREW token invariant borrows ideas from the token passing strategies we used in our CREW sorting scheme as well as in our method for cascade merging in a dag with a constant-width tree partition. In particular, we pass a token with each element e, as in our CREW sorting, and we deal with the deficits that this scheme creates by "stealing" tokens from where they are not needed, as in our method for cascade merging in a dag. To be precise, our token invariant is the following:

EREW Token Invariants (for after stage t): *Let v be a node in T.*

1. *If the children of v are not full, then each element e in $A_t(v)$ has a token associated with it, provided e is not in $\hat{L}_{t-1}(v)$ and e is not adjacent (in $A_t(v)$) to an element f such that $f \in \hat{L}_{t-1}(v)$.*

2. *If the children of v are full, but v is not full, then each element e in $A_t(v)$ has a token associated with it, provided e is not in $\hat{L}_{t-1}(v)$ and e is not adjacent to an element f such that $f = r_t(r_{t-1}(\cdots r_{t^*}(g)))$ for some $g \in \hat{L}_{t^*-1}(v)$, where t^* is the stage in which v's children became full.*

3. *If v is full, then each element e in $A_t(v)$ has a token associated with it, provided e is not in $\hat{L}_{t-1}(v)$ and e is either a falling element a genuine rising element.*

To avoid there being too few elements at the two ends of a list, we make the convention that $+\infty$ and $-\infty$ are present in each list, and are always placed (implicitly) in each sample.

Initially, the element in each leaf node has a token associated with it. The token assigned with an element e in $A_t(v)$ is passed to e's twin in $A'_t(v)$ (if e has a token to pass) in Step 1. This slightly modifies the token invariant at v, of course. For example, if v's children are not full, then an e in $A'_t(v)$ will have a token assigned to it provided e is not the twin in $L'_t(v)$ of an element of $L_t(v)$ that is also in $\hat{L}_{t-1}(v)$ nor is e adjacent to any such element. In Step 2, when v receives new elements from u, x, and y, we pass a token with each element v receives. That is, we send a token with each element in $L_t(u) - \hat{L}_{t-1}(u)$ (resp., $L_t(x) - \hat{L}_{t-1}(x)$, $L_t(y) - \hat{L}_{t-1}(y)$). But, in order to implement this, each node v with non-full children must send 3 tokens for each element in $L_t(v) - \hat{L}_{t-1}(v)$ (not just 1 token as in our CREW algorithm). To deal with this, we have the element e, which we wish to send to v's neighbors, "steal" the tokens from his predecessor and successor elements in $A_{t+1}(v)$—these are the tokens that we pass along with e to v's children x and y (we send e's token to u). This accounts for the parts of the token invariant mentioning elements adjacent to elements in $\hat{L}_{t-1}(v)$. If new elements are inserted (in Step 2) between a tokenless e in $\hat{L}_{t-1}(v)$ and its tokenless neighbors, then we pass the tokens for these new elements "down the row" so that the immediate neighbors of e are always tokenless. This maintains Token Invariant 1, provided $c_1 \geq 4$ (so there are always elements from which to steal tokens). If v's children are full, then we do not send any elements to them, so we need only send a copy of e to u. Thus, the token for e will only go to u in this case. Given that Token Invariant 1 was satisfied at v before v's children became full, then this maintains Token Invariant 2.

The two immediate neighbors of an element f in $A_t(v)$, such that $f = r_t(r_{t-1}(\cdots r_{t^*}(g)))$ for some $g \in \hat{L}_{t^*-1}(v)$, remain tokenless until v becomes full. Once v becomes full, the only real computation we are performing is to take the middle element, e, of each interval of $A_t(v)$ determined by two consecutive elements of $L_t(v)$ and put e into $L_{t+1}(v)$. In order to facilitate the implementation of our token-passing scheme, we modify our stage computation procedure so that right after v becomes full (i.e., after Step 2) we remove each fake rising element e from $A_t(v)$, which originally came from $L_{t-2}(v)$, and give its token to $r_t(r_{t-1}(e))$ in $L_t(v)$. We show below that this gives us Token Invariant 3 at v. A property that is crucial to our being able implement this is that there can be no fake rising elements in $L_t(v)$. For the token for such a fake rising element e would no longer be associated with e—it would have been sent along with copies of e sent to the nodes adjacent to v. Fortunately, as we will now show, we can guarantee that no fake rising element is ever put into $L_t(v)$. In the next lemma we show that, for each element e in $A_{t-1}(v)$, e "closely approximates" $r_t(e)$.

Lemma 5.11: *If e is an element in $A_{t-1}(v)$, then e is in the 9-neighborhood of $r_t(e)$ in $A_t(v)$.*

Proof: Let $\hat{e}$ denote the twin of e in $A'_{t-1}(v)$. Since $A_{t-1}(v)$ 3-conforms to $A'_{t-1}(v)$, by Lemma 5.9, e is in the 3-neighborhood of $\hat{e}$ in $A'_{t-1}(v)$. Note that by List Invariant 3 at u, x, and y for stage $t-1$, and because $A_t(v) = L_{t-1}(u) \cup L_{t-1}(x) \cup L_{t-1}(y)$, $A'_{t-1}(v)$ is a $(0,3)$-sample of $A_t(v)$. Thus, e is in the 9-neighborhood of the copy of $\hat{e}$ in $A_{t-1}(v)$, which is $r_t(e)$. $\square$

Thus, we immediately have the following corollary:

Corollary 5.12: *If e is an element in $A_{t-2}(v)$, then e is in the 27-neighborhood of $r_t(r_{t-1}(e))$ in $A_t(v)$.*

Proof: By the previous lemma, e is in the 9-neighborhood of $r_{t-1}(e)$ in $A_{t-1}(v)$. Consider the 9th element, d, less than $r_{t-1}(e)$ in $A_{t-1}(v)$. Again by the previous lemma, d is in the 9-neighborhood of $r_t(d)$ in $A_t(v)$. Similarly, the 9th element, f, greater than $r_{t-1}(e)$ is in the 9-neighborhood of $r_t(f)$ in $A_t(v)$. Therefore, e is in the 27-neighborhood of $r_t(r_{t-1}(e))$. $\square$

We use this corollary in the next lemma, which states that if c_1 is sufficiently large, then we can guarantee that there is no fake rising element in $L_t(v)$.

Lemma 5.13: *If $c_1 \geq 27$ and v was not full before stage t, then there can be no fake rising elements in $L_t(v)$.*

Proof: Suppose e is a fake rising element in $A_t(v)$. We must show that e is not in $L_t(v)$. Since $A_t(v) = L_{t-1}(u) \cup L_{t-1}(x) \cup L_{t-1}(y)$, without loss of generality, there is a falling copy of e in $L_{t-1}(x) \subseteq A_{t-1}(x)$. Thus, there is a copy of e in $L_{t-2}(v) \subseteq A_{t-2}(v)$ (by the definition of a falling element in $A_{t-1}(x)$). By the previous corollary, e is in the 27-neighborhood of $r_t(r_{t-1}(e))$ in $A_t(v)$. In addition, since e is in $L_{t-2}(v)$, $r_{t-1}(e)$ is in $L_{t-1}(v)$ and $r_t(r_{t-1}(e))$ is in $L_t(v)$. By List Invariant 4, the next element of $L_t(v)$ greater (resp., smaller) than $r_t(r_{t-1}(e))$ must be at least $c_1 + 1 \geq 28$ elements away from $r_t(r_{t-1}(e))$ in $A_t(v)$. Therefore, e cannot be in $L_t(v)$. $\square$

So, by taking $c_1 = 27$ and $c_2 = 54$, we can guarantee that we will never send a fake rising element from v to any of the nodes adjacent to v so long as v is not full. Thus, our token invariant can be maintained by the modification mentioned above (namely, by removing the fake rising elements from $A_t(v)$ once v becomes full). Moreover, we conserve the n original tokens. This gives us the following theorem:

Theorem 5.14: *Given a set S of n elements, one can sort S in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the EREW PPM model.* $\square$

6 Conclusion

We have designed general techniques for performing cascade merging in CREW and EREW parallel versions of the pointer machine model. In particular, we have shown that one can sort in $O(\log n)$ time using $O(n)$ processors in the CREW and EREW PPM models. Thus, we have established the existence of simple sorting algorithms for parallel models weaker than those used by Cole [10]. Some of the interesting aspects of our methods include the use of rank-linked and predecessor-linked samples, and the use of simple token-passing schemes to implement space and processor allocation.

We also showed how to generalize our approach to cascade merging in dags with bounded-width tree partitions, and we showed how to use this to achieve an asymptotic improvement over what seems possible making use of previous (array-based) techniques for solving the set-expression evaluation problem. Our method for this problem also runs in $O(\log n)$ time using an optimal $O(n)$ number of processors.

We leave open the following questions:

- Can one solve the set-expression evaluation problem optimally as a circuit (say, by extending the sorting network of Ajtai, Komlós, and Szemerédi [2])?
- What is the complexity of sorting on a CRCW PPM?

References

[1] Abrahamson, K., Dadoun, N., Kirkpatrick, D. G., and Przytycka, T. A simple parallel tree contraction algorithm. *J. Algorithms 10* (1989), 287–302.

[2] Ajtai, M., Komlós, J., and Szemerédi, E. Sorting in $c \log n$ parallel steps. *Combinatorica 3* (1983), 1–19.

[3] Atallah, M. J., Cole, R., and Goodrich, M. T. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput. 18* (1989), 499–532.

[4] Batcher, K. E. Sorting networks and their applications. In *Proc. 1968 Spring Joint Computer Conf.* (Reston, VA, 1968), AFIPS Press, pp. 307–314.

[5] Ben-Or, M. Lower bounds for algebraic computation trees. In *Proc. 15th Annu. ACM Sympos. Theory Comput.* (1983), pp. 80–86.

[6] Bilardi, G., and Nicolau, A. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *Information and Computation 18* (1989), 216–228.

[7] Bitton, D., DeWitt, D. J., Hsiao, D. K., and Menon, J. A taxonomy of parallel sorting. *ACM Computing Surveys 16*, 3 (1984), 287–318.

[8] Chazelle, B. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput. 17* (1988), 427–462.

[9] Chazelle, B., and Guibas, L. J. Fractional cascading: I. A data structuring technique. *Algorithmica 1* (1986), 133–162.

[10] Cole, R. Parallel merge sort. *SIAM J. Comput. 17*, 4 (1988), 770–785.

[11] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[12] Hagerup, T., and Rüb, C. Optimal merging and sorting on the erew pram. *Information Processing Letters 33* (1989), 181–185.

[13] Hong, J. W., Mehlhorn, K., and Rosenberg, A. L. Cost trade-offs in graph embedding with applications. *J. ACM 30* (1983), 709–728.

[14] J. H. Reif, e. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[15] JáJá, J. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Mass., 1992.

[16] Karp, R. M., and Ramachandran, V. Parallel algorithms for shared memory machines. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier/The MIT Press, Amsterdam, 1990, pp. 869–941.

[17] KNUTH, D. E. *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1968.

[18] KNUTH, D. E. *Sorting and Searching*, vol. 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[19] KOSARAJU, S. R., AND DELCHER, A. L. Optimal parallel evaluation of tree-structured computations by raking. In *Proc. AWOC 88*, vol. 319 of *Lecture Notes in Computer Science.* Springer-Verlag, 1988, pp. 101–110.

[20] LEIGHTON, F. T. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers C-34*, 4 (1985), 344–354.

[21] MILLER, G. L., AND REIF, J. H. Parallel tree contraction part 1: Fundamentals. *SIAM J. Comput. 5* (1989), 47–72.

[22] MILLER, G. L., AND REIF, J. H. Parallel tree contraction II: Further applications. *SIAM J. Computing 20* (1991), 1128–1147.

[23] PATERSON, M. Improved sorting networks with $o(\log n)$ depth. *Algorithmica 5*, 1 (1990), 75–92.

[24] SEDGEWICK, R. *Algorithms.* Addison-Wesley, Reading, MA, 1983.

[25] SHILOACH, Y., AND VISHKIN, U. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms 2* (1981), 88–102.

[26] TARJAN, R. E. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci. 18* (1979), 110–127.