SIMPLE DETERMINISTIC AND RANDOMIZED ALGORITHMS
FOR LINKED LIST RANKING ON THE EREW PRAM MODEL∗

SAJAL K. DAS†
Department of Computer Science, University of North Texas
Denton, TX 76203-8846

and

RANETTE H. HALVERSON
Department of Computer Science, Midwestern State University
Wichita Falls, TX 76308-8099

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

An asynchronous, CRCW PRAM (or APRAM) algorithm for linked list ranking, proposed by Martel and Subramonian, performs $O(n \log \log n)$ expected work employing $\frac{n}{\log n}$ processors. Motivated by their unique approach, this paper proposes two EREW list ranking algorithms – one deterministic and the other randomized. The deterministic algorithm performs in $O(\frac{n}{p} + \log(n/p))$ time using $p$ processors, where $n \geq p \log p$. Thus, for $p = O(n/\log n)$, it requires $O(\log n \log \log n)$ time and $O(n \log \log n)$ work. Although not work-optimal, this algorithm is very simple compared to the known work-optimal (deterministic) EREW algorithms for list ranking and has the added advantage of small constant factors in the time and space requirements. The randomized algorithm follows the same line of approach, but uses randomization in one step to decrease the time complexity, thus improving on the time complexity of the original algorithm. It requires $O(\frac{n}{p} + \log p)$ expected time, and hence it is an $EO(\log n)$ expected time, work-optimal algorithm employing $p = O(n/\log n)$ processors. Furthermore, the randomized algorithm uses less space than the deterministic algorithm.

Keywords: Linked list ranking, parallel algorithm, randomized algorithm, work-optimality.

1. Introduction

The linked list ranking problem (also referred to as list ranking) is defined as

∗The authors can be reached by email at das@cs.unt.edu or rhalver@nexua.mwusu.edu.
†This work is supported by Texas Research Grants TARP-003594003 and TATP-003594031.
follows. Given a linked list of length \( n \) stored in an array in which each element (except the last) has a pointer to its successor element in the list, the problem is to compute the rank of each element from the end of the list, i.e., the number of elements succeeding each element (including itself) in the list. Analogously, the rank of an element can be defined in terms of its predecessors in the list.

Because of its importance in solving many problems, various parallel algorithms have been proposed for linked list ranking. They include deterministic [AM88, CV86b, CV88, KRS86, WH86] and randomized algorithms [AM90, V87] on synchronous PRAM models, as well as on fixed connection models such as hypercubes [RJ90], direct connect machines (DCM) [H89, KMR86] and reconfigurable meshes [OSZ91]. For a comprehensive survey of parallel list ranking algorithms, refer to Halverson and Das [HD93].

Most of the existing list ranking algorithms are based on an approach, referred to as reduce-rank-expand [HD93]. This technique involves deleting non-adjacent list elements in parallel until the remaining list is of size \( O(p) \) — called the reduction step — where \( p \) is the number of processors employed. The pointer jumping algorithm due to Wyllie [W79] is then invoked to rank the reduced list. The deleted elements are reinserted to the list in the reverse order of their deletion, computing the rank of each element as it is replaced — this is the expansion step. Various algorithms differ primarily in the manner in which elements are selected for concurrent deletion.

Of particular interest here is an asynchronous, CRCW PRAM (or APRAM) list ranking algorithm described in [MS90]. Although it requires \( EO(n \log \log n) \) expected work, which is slightly more than the optimal (linear) work required by the best known synchronous algorithm, the APRAM algorithm is quite simple, and the underlying approach is not the commonly used reduce-rank-expand method.

The EREW PRAM algorithms (deterministic and randomized) presented in this paper are motivated by this new technique. By way of comparison, we show that the deterministic algorithm performs in \( O(\frac{n}{p} \cdot \log(n/p)) \) time using \( p \) processors, where \( n \geq p \log p \). Thus, for \( p = \Theta(n/\log n) \), it requires \( O(\log \log n) \) time and \( O(n \log \log n) \) work, matching the performance of the original APRAM algorithm. Although not work-optimal, this algorithm is very simple compared to the known work-optimal, \( O(\log n) \) time (deterministic) EREW algorithms for list ranking and has the added advantage of small constant factors in the time and space requirements. The randomized algorithm also follows the same line of approach, but uses randomization in one step to decrease the time complexity, thus improving on the performance of the original algorithm. It requires \( EO(\frac{n}{p} \log p) \) expected time, achieving work-optimality for \( p \leq O(\frac{n}{\log n}) \). Thus this algorithm improves on the expected time complexity of the original APRAM algorithm for list ranking. Furthermore, our randomized algorithm uses less space than the proposed deterministic algorithm.

In Section 2, the computation models and pertinent terms are defined. This is followed by an overview of the existing APRAM algorithm in Section 3. A detailed description is then provided in Section 4 for each of the new algorithms, including an example and the complexity analysis. Finally, Section 5 outlines the advantages
of our algorithms compared to several known list ranking techniques.

2. Computation Model and Performance Metrics

The parallel random access machine (PRAM) is a widely accepted shared-memory model for parallel computation [J92]. It consists of \( p \) general-purpose processors (each having a small local memory) and a common, shared memory. The processors work in a synchronous fashion, and all synchronization and communication take place via the global memory. Based upon how a shared memory cell is accessed, several variants of the PRAM model are distinguished. The ability for two or more processors to access a single cell simultaneously for a read (write) operation is referred to as concurrent read (write), otherwise it is exclusive read (write). Thus, we distinguish CRCW, CREW and EREW models. Naturally, the EREW (exclusive-read and exclusive-write) PRAM is the weakest, albeit the most feasible model.

Another variation of the PRAM model which has recently drawn interest is the APRAM or asynchronous PRAM [CZ89, G89, MS90]. This model has the same features as the CRCW PRAM with the exception that the processors may all have different clock speeds, causing them to work asynchronously. In addition, each processor has access to an independent random-number generator, which is used to allow processors to randomly select the work to be performed.

For a given problem, let \( T_1 \) and \( T_p \), respectively, represent the running times of the best known sequential algorithm and a (synchronous) parallel algorithm using \( p \) processors. Then the speedup of the parallel algorithm is given by the ratio \( T_1/T_p \), and the work (also referred to as cost) is \( W = p * T_p \). If \( W = O(T_1) \), the parallel algorithm is said to be work-optimal (cost-optimal), and it achieves linear speedup within a constant factor.

An algorithm is said to be randomized if some portion of its outcome is nondeterministic, with its performance stated in terms of the expected time complexity. We shall use the notation \( EO(f(n)) \) to represent the expected order of various parameters of a randomized algorithm, including time, space, work and number of elements.

In order to measure the performance of an asynchronous PRAM algorithm, a different performance metric, but also called work, is used in [MPS92, MS90]. The work of a single execution of an asynchronous algorithm is the total number of single processor instructions performed by the set of asynchronous parallel processors, including busy-wait instructions. Because of the randomization in the asynchronous algorithms, the performance is expected work, which is comparable to the work of a synchronous algorithm.

3. APRAM List Ranking Algorithm

The APRAM algorithm proposed by Martel and Subramonian [MS90], requires \( EO(n \log \log n) \) expected work using \( p = O(n/\log n) \) processors. This result depends on the distance between elements which are selected for processing. The
following lemma places a probabilistic bound on this distance and computes the expected value.

**Lemma 1 [MS90]:** Consider \( n / \log n \) random selections with replacement from an ordered list of \( n \) cells. Let \( X \) be the maximum number of contiguously unselected cells. Then, the probability \( \Pr[X > 4 \log^2 n] < 1/n \) and the expected value \( E[X] = O(\log^2 n) \).

The APRAM algorithm assumes that a singly linked list is stored in an array. The processors randomly select a set of elements which, based on Lemma 1, are expected to be evenly distributed within the list. Knowing that the selected elements have their ranks computed first, pointer jumping is used to cause each list element including the selected ones, to point to its nearest selected successor or to have a pointer of length \( \log n \). In order to compute the ranks, the number of pointers jumped is retained in each element.

The selected elements are then compacted into a smaller array where they are ranked using a pointer jumping algorithm [MPS89, W79]. The ranks are written back into the original array. Each unranked element then follows its pointers to read the rank of its nearest selected successor and computes its own rank. For the sake of completeness, let us summarize the APRAM algorithm from [MS90].

**Algorithm Asynchronous List Ranking**

**Step 1.** Select \( m = EO(n / \log n) \) elements at random by generating a random number between 1 and \( n \) for each element in the list and selecting those elements having values between 1 and \( n / \log n \).

**Step 2.** Perform \( \log \log n \) iterations of pointer jumping on the list elements, so that each element has a pointer to the end of the list, to a selected element, or has a pointer of length \( \log n \). (That is, perform each iteration on all elements before proceeding to the next iteration.)

**Step 3.** Compact the \( m \) selected elements into a smaller array.

**Step 4.** Follow the pointer of each selected element to guarantee that it points to the next selected element or the end of the list.

**Step 5.** Compute the ranks of the elements of the reduced list.

**Step 6.** Copy these ranks into the complete list resulting from Step 2. Follow the pointer of each non-selected element \( x \) to a successor element \( y \), which is ranked, and compute \( RANK(x) := RANK(x) + RANK(y) \).

In general, APRAM algorithms require a synchronization mechanism to guarantee that in each step, all data elements are processed before proceeding to the next step. As a processor completes the processing of a data element, a group of flags is set. It is possible to have more than one processor concurrently read or set (i.e. write) a single flag. Thus, both the concurrent read and concurrent write (CRCW) capabilities are required in the model, which is independent of any particular algorithm. Additionally, the APRAM list ranking algorithm itself may
require concurrent reads in Steps 2, 4 and 6, as the processors follow pointers asynchronously through the list.

4. EREW List Ranking Algorithms

In order to devise an EREW PRAM algorithm, two modifications must be made to the APRAM algorithm, which inherently uses concurrent read and write capabilities. First is the elimination of the synchronization mechanism. Since the processors of the PRAM model operate synchronously, there is no need for the software synchronization program used by the APRAM.

A modification of Steps 2, 4 and 6 (of Algorithm Asynchronous List Ranking) employs a partitioning of the data to eliminate concurrent reads, as well. This approach requires that a linked list of size $O(p)$, where $p$ is the number of processors, be constructed from the original list so that the elements in the shorter list (referred to as selected elements) are nearly equally spaced in the original list with respect to their pointer distances. The APRAM algorithm accomplishes this task while at the same time causing the pointer of every list element to point to its nearest selected successor. The proposed EREW algorithms construct the shorter linked list of selected elements while leaving the pointers of all the unselected elements intact. As a result, the last step of our algorithms also vary from that of the APRAM algorithm. To be more precise, while the APRAM algorithm causes each element to concurrently read the rank of its successor, our algorithms cause each processor to scan the list from a selected element, computing the ranks as it proceeds.

The details of the proposed EREW algorithms are given in the following two subsections. They vary primarily in the manner in which elements are selected for the shorter list. The randomized algorithm uses Las Vegas style randomization and closely follows the APRAM method. The deterministic algorithm uses a totally different approach, referred to as deterministic coin tossing due to Cole and Vishkin [CV86b].

As usual, the input linked list of $n$ elements is stored in a contiguous array. The deterministic (randomized) algorithm assumes a doubly-linked (singly-linked) list. In addition to the list fields, an array RANK (initialized to 1's) is used to store the computed rank of each element, and an array STATUS stores the flag for selected elements.

4.1. Randomized Algorithm

The first step of this algorithm follows the APRAM algorithm very closely by randomly selecting $O(p)$ elements from the input list. These elements are then compacted into a smaller array, and each processor is assigned to a partition of size $O(1)$. Each processor scans the original list starting at its assigned elements to find the first successor element which has ‘STATUS = selected’. The pointers and the pointer distances between the selected elements are stored in the compacted array, thus constructing a linked list of the selected elements and the (pointer) distance between them. The compacted list is ranked by such distances using a
pointer jumping algorithm [W79], and the ranks are written back into the original list array. Processors again scan the linked list from their assigned elements to compute the ranks of the remaining elements.

4.1.1. Illustrative Example

Assume that \( p = 4 \). The 16 element list \( L \) is given in Table 1 such that \( L(0) \) is the head of the list pointing to element \( L(3) \). Each processor is assigned to one of four partitions. It processes its partition by generating a random number between 1 and \( n = 16 \) and marking as 'selected' those elements having values between 1 and 4, inclusive. The selected elements are marked with 's' in Step 1 of Table 1.

<table>
<thead>
<tr>
<th>Table 1. Tracing EREW Randomized Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>List L</td>
</tr>
<tr>
<td>BAKN</td>
</tr>
<tr>
<td>Step 1</td>
</tr>
<tr>
<td>Random #</td>
</tr>
<tr>
<td>Select</td>
</tr>
<tr>
<td>Step 2</td>
</tr>
<tr>
<td>New list</td>
</tr>
<tr>
<td>Step 3</td>
</tr>
<tr>
<td>RANK</td>
</tr>
<tr>
<td>Step 4</td>
</tr>
<tr>
<td>Final RANK</td>
</tr>
</tbody>
</table>

In Step 2 the elements are compacted into a smaller array and linked together based on their locations in the original list, and the pointer distances between successive elements are computed. (For brevity, the details of compaction are not shown.) For example, suppose that processor \( P_0 \) is assigned to the element \( L(0) \) in the list. It follows the links from \( L(0) \) to \( L(3) \), then to \( L(6) \), and finally to \( L(8) \), which is also selected. Thus, \( P_0 \) sets \( \text{RANK}(0) := 3 \) and changes its pointer to the location of \( L(8) \) in the compacted array. In Step 3, the compacted array is partitioned, each processor is assigned to a constant number of elements and the ranks are computed. In Step 4, the processors write the ranks of their assigned elements back into the original array. Each processor then scans forward from its assigned element, computing the ranks of the unselected elements until it encounters another selected element. For example, \( P_0 \) writes the rank of 16 into \( L(0) \) and again follows the pointers to \( L(3) \) and \( L(6) \) computing their ranks as 15 and 14, respectively.

4.1.2. Complexity Analysis

Before analyzing the performance of our randomized algorithm, let us generalize Lemma 1, proposed in [MS90], to bound the distance between selected elements.

**Lemma 2:** Consider an \( n \)-element list given in some order. Each element is randomly assigned a value between 1 and \( n \), inclusive. Those elements assigned values less than or equal to \( p \) are selected. Let \( X \) be the maximum number of contiguous, unselected elements in the list. Then for some \( k < p \), where \( k \) is a function of \( n \), the probability \( \Pr[X \geq 2n/k] \leq ke^{-p/k} \) and the expected value \( E[X] \leq 2n/k + nk e^{-p/k} \).
Proof: Assume the list is divided into \( k \) partitions each containing \( n/k \) elements. The probability that any given cell is selected is \( p/n \). Define \( Y_i = 0 \) if no cell in the \( i \)th partition is selected, and \( Y_i = 1 \) otherwise. Then

\[
\Pr[Y_i = 0] := (1 - \frac{p}{n})^{n/k} \rightarrow e^{-p/k} \\
\Pr[X > 2n/k] \leq \Pr[\text{at least one } Y_i = 0] \leq \sum_{i=1}^{k} \Pr[Y_i = 0] = ke^{-p/k} \\
\text{Thus, } \Pr[X \leq 2n/k] \geq 1 - ke^{-p/k}
\]

The expected value of \( X \) is therefore computed as follows:

\[
\mathbb{E}[X] = \sum_{x=1}^{n} x \Pr[x] = \sum_{x=1}^{2n/k} x \Pr[x] + \sum_{x=2n/k}^{n} x \Pr[x]
\]

In the first term, substitute the maximum possible values of 1 for \( \Pr[X \leq 2n/k] \) and \( \frac{2n}{k} \) for \( x \). In the second term substitute the expression for \( \Pr[X > 2n/k] \) and the maximum of \( n \) for \( x \). This provides the following inequality.

\[
\mathbb{E}[X] \leq \frac{2n}{k} \sum_{x=1}^{2n/k} x \Pr[x] + nke^{-p/k} \leq \frac{2n}{k} + nke^{-p/k}
\]

If a partition size of \( k = n/(2 \log^2 n) \) and \( p = n/\log n \) is selected, then the results obtained are the same as in [MS90]. That is, \( \Pr[X \leq 4 \log^2 n] < 1/n \) and \( \mathbb{E}[X] = O(\log^2 n) \). For a partition size of \( k = n/\log n \) and \( p = n/\log n \), \( \mathbb{E}[X] = O(\log n) \). \( \square \)

Now we are ready to analyze the time complexity of our algorithm. The random selection of elements requires \( O(n/p) \) time. The number of elements selected is \( EO(p) \) each being \( EO(n/p) \) distance apart, according to Lemma 2. The compaction requires \( O(n/p + \log p) \) time using the parallel prefix sums algorithm. Forming the linked list of selected elements requires \( EO(n/p) \) time. Using the pointer jumping algorithm, the reduced list is ranked in \( EO(\log p) \) time. Because there are \( EO(p) \) elements, each processor is assigned to \( EO(1) \) elements. Thus for each processor, writing is an \( EO(1) \) time operation, and the sequential scan of the list requires \( EO(n/p) \) time.

Therefore, the overall time required by the randomized EREW algorithm is \( EO(\frac{n}{p} + \log p) \), which provides work-optimal speedup for \( p \leq O(n/\log n) \) processors, achieving \( EO(\log n) \) time.

4.2. Deterministic Algorithm

In order to deterministically accomplish the results of the randomized Step 1 of the previous algorithms, we use the technique of constructing a 2-ruling set. Given an \( n \)-element linked list, a 2-ruling set is a subset \( U \) of elements such that no two elements of \( U \) are adjacent and for every element in the list, there is a directed path from it to some element in \( U \), having path length at most two. In more practical terms, there are exactly one or two non-ruling set elements between two successive ruling set elements. Using an algorithm by Cole and Vishkin [CV86b], a 2-ruling set can be constructed in \( O(n/p) \) time using \( p \) processors on the EREW PRAM model. The 2-ruling set algorithm is first applied to the original list and then to the newly constructed 2-ruling set to increase the distance between the
selected elements. Performing $O(\log(n/p))$ iterations of this algorithm, elements are eliminated from the set and the distance between ruling set elements is doubled each time, thus ensuring that $O(p)$ elements are selected and the distance between successive elements is $O(n/p)$.

As in the randomized algorithm, the selected list is then compacted into a smaller array, where the ranking is computed via the pointer jumping technique. The processors write the ranks back into the original array and scan the list to compute the remaining ranks. The steps of the deterministic EREW PRAM algorithm are formally stated as follows.

Algorithm Deterministic EREW List Ranking

Step 1. Select a ruling set of $O(p)$ elements from the list so that the elements are $O(n/p)$ distance apart, forming a linked list of the selected elements and computing the actual distance between each pair of elements. This is accomplished by repeatedly constructing a 2-ruling set of the previous ruling set, increasing the distance between selected elements on each iteration until the distance between the elements is $O(n/p)$. Add the head of the list to the ruling set if not already selected. (See implementation details in subsection 4.2.2.)

Step 2. Compact the $O(n/p)$ selected elements into a separate array.

Step 3. Rank the elements of the selected list.

Step 4. Write the rankings back to the original array. Each processor scans forward in the original list, computing the ranks of its successor elements until another ruling set element is encountered.

4.2.1. Illustrative Example

In the deterministic algorithm, the only significant change is in Step 1. Each processor is assigned to a partition of the original array for which a 2-ruling set is computed as the first iteration. These selected elements are linked together to form a linked list with the pointer distances between successive selected elements stored in the RANK field as shown in the first iteration of Table 2. On the second iteration, each processor scans its entire partition but computes the ruling set based only on the elements selected in the previous iteration.

In Step 2, the elements that have been selected are compacted to a new array. (This is not explicitly shown in the example.) Step 3 causes the ruling set elements to be ranked. In Step 4, the ranks are written into the original array. The processors then scan the original pointers to compute the ranks as in the previous example.

4.2.2. Complexity Analysis

In Step 1, the selection of the set of $O(p)$ elements that are $O(n/p)$ distance apart requires $O(\frac{p}{p} \cdot \log \frac{n}{p})$ time, as detailed below. Steps 2 and 3 require $O(\frac{n}{p} + \log p)$ and
Table 2. Tracing EREW Deterministic Algorithm

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Iteration 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Ruling</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) List</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANK</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Iteration 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Ruling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) List</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANK</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANK</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final RANK</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

O(log p) time, respectively. In Step 4, because there are O(p) elements and each processor is assigned O(1) elements, writing takes O(1) time while the sequential scan of the list requires O(n/p) time.

The implementation and analysis of Step 1 are now detailed. The list is divided into p partitions, each being assigned to a processor which repeatedly performs the following steps (a) and (b) until the size of the selected ruling set is O(p).

(a) Construct a 2-ruling set using deterministic coin tossing [CV86b] in O(n/p) time. At most, n/2 elements are in the ruling set and as few as n/3 elements may be selected. (On successive iterations, construct a ruling set from that obtained in the previous iteration.)

(b) Follow the links from each ruling set element to the next such element to create a linked list consisting of the ruling set elements only. Count the number of links traversed (which is at most three links for each element) by adding the contents of the RANK field of the element 'jumped' Since to the ruling set element being processed. each processor is assigned at most n/p elements, this step requires O(n/p) time.

The operations described in (a) and (b) above are repeated to reduce the number of elements in the set to O(p). Let m = n/p for the original values of n and p. If no compaction is used between iterations of Step 1, each iteration requires m = O(n/p) time. Although the number of elements which are actually being used in the computation of the ruling set is reduced by one-half on each iteration, each processor must scan its entire partition to ‘find’ these elements. The following recurrence equation describes the time complexity of Step 1 of the algorithm.

\[
T(n) := \begin{cases} 
T(n/2) + O(m) & \text{for } n > p \\
O(1) & \text{for } n \leq p 
\end{cases}
\]

It has the solution \( T(n) = O(\frac{n}{p} \ast \log \frac{n}{p} ). \)

Thus, the overall time complexity for the deterministic algorithm is \( O(\log p + (n/p) \ast \log (n/p)) = O(n/p \ast \log (n/p)) \) for \( n \geq p \log p \). Thus, for \( p = O(n/\log n) \), it attains time complexity \( O(n/\log \log n) \) and work \( O(n/\log \log n) \).

Consider a variation of Step 1 in which compaction is used between iterations. Each iteration requires \( O(\frac{n}{p^2}) \) time for the parts (a) and (b), computing the ruling set and linking the elements. The compaction of the new list into a smaller array
requires $O\left(\frac{n}{p} + \log p\right)$ time, leading to the following recurrence relation.

$$T(n) := \begin{cases} T(n/2) + O\left(\frac{n}{p} + \log p\right) & \text{for } n > p \\ O(1) & \text{for } n \leq p \end{cases}$$

Solving this recurrence provides the same results as without compaction.

5. Advantages of New Algorithms

The proposed EREW algorithms are simple and use very basic parallel techniques. Therefore, they are easy to implement. They also have advantages over other list ranking algorithms, as outlined below.

1. Small Constants: All of the previously known, work-optimal EREW algorithms use the reduce-rank-expand method [AM90, CV86a, CV86b, CV88, KRS86, WH86], which requires the additional $O(\log n)$ phase of rebuilding the reduced list after it has been ranked. Several of these algorithms use time consuming techniques such as recursively compacting the array as the list is reduced [CV86b, CV88, KRS86, WH86] or generating expander graphs for balancing the work load among processors [CV86a, CV88]. Since these techniques are not used in our algorithms, the constant factors involved in the time complexity analysis are reduced.

2. Small Space Requirements: Table 3 provides a comparison of the required workspace of six work-optimal EREW list-ranking algorithms (five deterministic and one randomized) with our algorithms. The space requirements of these algorithms are due to compaction arrays, deletion stacks and list fields. From this table it is seen that all previous algorithms require stack space for reduction, and all but #5 and #6 use repeated compaction.

It is assumed that the linked list for each algorithm is stored in a contiguous array with the minimal fields of NEXT (a pointer to the successor element), PREV (a pointer to the predecessor element), and RANK. All those using reduction (#1 through #6) also require a STATUS field. All but #1 use deterministic coin tossing or an equivalent technique which requires at least three additional fields, for each element of the list. Algorithm #4 requires a counter field for each element to determine the number of successors that have been deleted. Clearly, the space required for the list is $O(n)$ for all the algorithms, but the variation in the constant factor is shown in the second column of Table 3.

In the reduce-rank-expand technique of list ranking, each iteration of the algorithm causes approximately one-half of the elements to be marked as deleted. The unmarked elements are then compacted into a new array, producing a shorter list. The new list is then processed in a similar manner to reduce it by approximately one-half. This reduction continues until the list is of size $O(p)$. Because the information in each reduced list is necessary for the expansion phase, a new array must be used for each iteration. The total space used for compaction arrays is $O(n)$, with the constants being that used for storage of the original array.
### Table 3. Space Requirements

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>FIELDS</th>
<th>OTHERS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1: Divide &amp; Conquer [KRS86]</td>
<td>4n</td>
<td>recursion stack (n)</td>
<td>9n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction (4n)</td>
<td></td>
</tr>
<tr>
<td>#2: Ruling Set with Compaction [CV86b]</td>
<td>7n</td>
<td>stack (n)</td>
<td>15n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction (7n)</td>
<td></td>
</tr>
<tr>
<td>#3: Reduction Using Maximal Matching [WH86]</td>
<td>7n</td>
<td>stack (n)</td>
<td>15n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction (7n)</td>
<td></td>
</tr>
<tr>
<td>#4: Ranking With Expander Graphs [CV86a, CV88]</td>
<td>8n</td>
<td>expander graph (3n)</td>
<td>24n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary trees (4n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>stack (n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction (8n)</td>
<td></td>
</tr>
<tr>
<td>#5: Ruling Set without Compaction [AM88]</td>
<td>7n</td>
<td>stack (n)</td>
<td>8n</td>
</tr>
<tr>
<td>#6: Coin Tossing without Compaction [AM90]</td>
<td>5n</td>
<td>permutation array (n)</td>
<td>6n</td>
</tr>
<tr>
<td>#7: Pointer Jumping on a Ruling Set [this paper]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a) deterministic</td>
<td>7n</td>
<td>one compaction (n)</td>
<td>8n</td>
</tr>
<tr>
<td>b) randomized</td>
<td>3n</td>
<td>one compaction (n)</td>
<td>4n</td>
</tr>
</tbody>
</table>

*O(\log n)* time complexity

The column labeled OTHERS gives the space required by other data structures. The numbers in parentheses indicate the minimum amount of space required. The last column, TOTAL, indicates the minimum total space required for the implementation of the algorithm. We note that the reason for the larger space requirements of the previously known algorithms are due to the actual techniques inherent in the reduce-rank-expand approach, and are not due to implementation inefficiencies.

As shown in Table 3, the proposed deterministic algorithm requires significantly less space than all but one algorithm. Also, our randomized implementation requires less space than the well-known randomized algorithm [AM90].

### 6. Conclusion

Although several work-optimal linked list ranking algorithms are known for the EREW PRAM model, the proposed deterministic and randomized algorithms provide a simple method of list ranking, efficient in both time and space requirements.

**Acknowledgements:** We are grateful to Dr. William Hinds of Midwestern State University for his help in the formulation of Lemma 2. Thanks are also due to the referees for insightful comments.

### References


