CHAPTER 3

LINKED LIST RANKING ALGORITHMS ON EREW PRAM

3.1 Introduction

An asynchronous, CRCW PRAM (or APRAM) algorithm for linked list ranking, proposed by Martel and Subramonian [MS90a, MS90b], performs $\text{EO}(n \log \log n)$ expected work employing $\frac{n}{\log n}$ processors. Motivated by their unique approach, two EREW list ranking algorithms are proposed — one deterministic and the other randomized. The deterministic algorithm performs in $O\left(\frac{n}{p} \times \log(n/p)\right)$ time using $p$ processors, where $n \geq p \log p$. Thus, for $p = O(n/\log n)$, it requires $O(n \log n \log \log n)$ time and a work of $O(n \log \log n)$. 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. With high probability, it requires $O\left(\frac{n}{p} + \log p\right)$ time and, hence, it is an $O(\log n)$-time, work-optimal algorithm employing $p = O(n/\log n)$ processors. Furthermore, it uses less space than the deterministic algorithm.

3.2 APRAM Model and Performance Metrics

A variation of the PRAM model which has recently drawn interest is the APRAM or asynchronous PRAM [CZ89, G89, MS90a, MS90b]. 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.

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. The notation $EO(f(n))$ is used to represent the expected order of various features of a randomized algorithm, including time, space, work and number of elements. The notation $EO(f(n))$ is used to indicate that, with a high probability, the results are within the specified order of $f(n)$.

In order to accurately describe the performance of an asynchronous PRAM algorithm, a different performance metric, also called work, is used in [MPS89, MS90a, MS90b]. The work of a single execution of an 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 incorporated into the asynchronous algorithms, the performance is expected work, which is comparable to the work of a synchronous algorithm.

3.3 APRAM List Ranking Algorithm

The APRAM algorithm which was proposed by Martel and Subramonian requires $EO(n \log \log n)$ expected work using $p = O(n/ \log n)$ processors [MS90a,MS90b]. This result is dependent upon the distance between elements which are selected for processing. The following lemma places a probabilistic bound on this distance and computes an expected value.

**Lemma 1** [MS90a, MS90b]: Consider $n/ \log n$ random selections with replacement from an ordered list of $n$ cells. Let $X$ be the maximum number of contiguously unse-
lected cells. Then, $P[X > 4\log^2 n] < 1/n$ and $E[X] = O(\log^2 n)$.

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, have a high probability of being evenly distributed within the list. Knowing that the selected elements will have their ranks computed first, pointer jumping is used to cause each list element, including the selected elements, to point to its nearest selected successor or to have a pointer of length $\log n$. In order to be able 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 compute its own rank. An overview of the APRAM algorithm by Martel and Subramonian [81, 82] is provided for the sake of completeness.

Algorithm Asynchronous List Ranking [81, 82]

Step 1. Select $m = EO(\frac{n}{\log n})$ 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 $\frac{n}{\log n}$, inclusive.

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 of the algorithm, 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 may be the case that more than one processor will read or set (write) the same flag concurrently. This requires both the concurrent read and concurrent write (CRCW) capabilities of the model, which is independent of any particular algorithm. In addition, the APRAM list ranking algorithm itself may require concurrent read of other elements in steps 2, 4 and 6, while the processors follow pointers asynchronously through the list.

3.4 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 in section 3 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 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 the 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, the
EREW algorithms cause each processor to scan the list from a selected element,
computing the ranks as it proceeds.

The details of the two EREW algorithms are given in the following 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].

Assume a linked list of \( n \) elements is stored in a contiguous array. The determin-
istic (randomized) algorithm assumes a doubly-linked (singly-linked) list. In addition
to the list fields, an array, \( \text{RANK} \) (initialized to 1's), is used to store the computed
rank of each element, and the array, \( \text{STATUS} \), stores the flag for selected elements.

3.4.1 Randomized Algorithm

The first step of the randomized algorithm follows very closely the APRAM algo-
rithm by randomly selecting \( EO(p) \) 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 \( \text{STATUS} = \text{selected} \). 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 the pointer 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.

Illustrative Example

Assume that \( p = 4 \). The 16-element list is given in table 3.1 where \( L(0) \) is the head of the list and points to \( L(3) \). The array is divided into \( p \) partitions with a processor assigned to each one. A processor processes its assigned 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 ‘*’ in step 1 of table 3.1.

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 computed. (The actual compaction is not shown for brevity.) For example, suppose that processor \( P_0 \) is assigned to \( L(0) \) 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, \( P_0 \) writing the rank of 16 into \( L(0) \) again follows the pointers to \( L(3) \) and \( L(6) \) computing their ranks as 15 and 14, respectively.
Table 3.1: Demonstration of the EREW Randomized Algorithm

<table>
<thead>
<tr>
<th>List L</th>
<th>RANK</th>
<th>[0, 3, 6, 8, 5, 1, 2, 4, 11, 9, 7, 10, 14, 12, 15, 13]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]</td>
</tr>
<tr>
<td><strong>Step 1</strong></td>
<td><strong>Random #</strong></td>
<td>[3, 8, 10, 3, 9, 13, 16, 11, 2, 15, 7, 1, 8, 6, 4, 12]</td>
</tr>
<tr>
<td></td>
<td><strong>Select</strong></td>
<td>[*]</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td><strong>New list</strong></td>
<td>[0, 8, 11, 10, 15]</td>
</tr>
<tr>
<td></td>
<td><strong>RANK</strong></td>
<td>[3, 5, 3, 3]</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td><strong>RANK</strong></td>
<td>[6, 13, 8, 5]</td>
</tr>
<tr>
<td></td>
<td><strong>RANK</strong></td>
<td>[16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]</td>
</tr>
</tbody>
</table>

Complexity Analysis

Before analyzing the performance of the randomized algorithm, a generalization of Lemma 1 (from [81]) is given to bound the distance between selected elements.

**Lemma 2:** Consider \( p \) random selections with replacement from an ordered list of \( n \) elements. Let \( X \) be the maximum number of contiguous unselected elements, and \( k < p \) be a function of \( n \). Then \( P[X \geq 2n/k] \leq ke^{-p/k} \) and \( E[X] \leq 2n/k - n(k - 1)e^{-p/k} \).

**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

\[
P[Y_i = 0] = (1 - \frac{p}{n})^{n/k} \rightarrow e^{-p/k}
\]

\[
P[X \geq 2n/k] \leq P[\text{at least one } Y_i = 0] \leq \sum_{i=1}^{k} P[Y_i = 0] = ke^{-p/k}
\]

\[
E[X] \leq nke^{-p/k} + (1 - ke^{-p/k}) \cdot \frac{2n}{k} = \frac{2n}{k} - n(k - 1)e^{-p/k}
\]

By selecting the partition size of \( k = n/(2\log^2 n) \) and \( p = n/\log n \), the results obtained are the same as in [MS90a, MS90b]. That is, \( P[X \leq 4\log^2 n] < 1/n \) and \( E[X] = O(\log^2 n) \). \(\square\)
The random selection of elements requires \( O(n/p) \) time. With high probability, \( O(p) \) elements are selected which are \( O(n/p) \) 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 [W79], 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 requires \( EO(1) \) time and the sequential scan of the list requires \( EO(n/p) \) time. Therefore, the overall time requirement for the randomized EREW algorithm is \( EO(n/p + \log p) \), which provides work-optimal speedup for \( p \leq O(n/\log n) \) processors, achieving \( EO(\log n) \) time.

3.4.2 Deterministic Algorithm

In order to deterministically accomplish the results of the randomized Step 1 of the previous algorithms, the technique of constructing a 2-ruling set is used. 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 each ruling set element and its nearest ruling set successor. Using the algorithm by Cole and Vishkin [25], the 2-ruling set of a linked list is constructed in \( O(n/p) \) time using \( p \) processors. 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. By repeatedly applying this algorithm \( O(\log n/p) \) times, 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 pointer jumping [W79]. 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 $O\left(\frac{n}{p}\right)$ 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 $\left(\frac{n}{p}\right)$. Add the head of the list to the ruling set if not already selected.

Step 2. Compact the $O\left(\frac{n}{p}\right)$ 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.

Illustrative Example

In the deterministic algorithm, the only significant change from the randomized algorithm 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 3.2.
Table 3.2: Demonstration of the EREW Deterministic Algorithm

| List L | 0 3 6 8 5 1 2 4 11 9 7 10 14 12 15 13 |
| RANK   | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
| Step 1 (iteration 1) | (a) Ruling | * * * * * * * * * |
|        | (b) List   | 0 3 8 1 4 7 12 13 |
|        | RANK       | 1 2 2 2 3 3 2 1 |
| (iteration 2) | (a) Ruling | * * * * * * * |
|        | (b) List   | 0 3 1 7 13 |
|        | RANK       | 1 4 5 5 1 |
| Step 3 RANK | 16 15 11 6 1 |
| Step 4 Final RANK | 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 |

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.

Complexity Analysis

In step 1 of the algorithm, the selection of the set of $O(p)$ elements that are $O(\frac{n}{p})$ distance apart requires $O(\frac{n}{p} \log \frac{n}{p})$ time, as detailed below. Steps 2 and 3 require $O(\frac{n}{p} + \log p)$ and $O(\log p)$ 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 $O(\frac{n}{p})$ 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 [25] in \( O\left(\frac{n}{p}\right) \) time. At most, \( \frac{n}{2} \) elements are in the ruling set and as few as \( \frac{n}{3} \) 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 3 links for each element) by adding the contents of the RANK field of the element 'jumped' to the ruling set element being processed. Since each processor is assigned at most \( \frac{n}{p} \) elements, this step requires \( O\left(\frac{n}{p}\right) \) time.

The operations described in (a) and (b) above are repeated to reduce the number of elements in the set to \( O(p) \). Let \( m = \frac{n}{p} \) for the original values of \( n \) and \( p \). Since no compaction is used between iterations of step 1, each iteration requires \( m = O\left(\frac{n}{p}\right) \) 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.\(^2\) The following recurrence equation describes the time complexity of step 1 of the algorithm.

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

The recurrence equation has the solution \( T(n) = O\left(\frac{n}{p} \log\frac{n}{p}\right) \), making the overall time complexity for the deterministic algorithm \( O(\log p + \frac{n}{p} \log\frac{n}{p}) = O\left(\frac{n}{p} \log\frac{n}{p}\right) \) for \( n \geq p \log p \). Thus, for \( p = O\left(\frac{n}{\log n}\right) \), it attains time complexity \( O(\log n \log \log n) \) and work \( O(n \log \log n) \).

\(^2\) Analysis of a version which does compact between steps shows that the overall time performance of the algorithm is not affected. However, the incorporation of compaction incurs the penalty of a larger space requirement and causes the algorithm to be less simple.
3.5 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 [7, 24, 25, 28, 69, 110]. This approach 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 [25, 28, 69, 110] or generating expander graphs for balancing the work load among processors [24, 28]. Since these techniques are not used in the algorithms proposed here, the constant factors involved in the time complexity analysis are reduced.

2. **Small Space Requirements**: Table 3.3 provides a comparison of the required work space of six work-optimal EREW list-ranking algorithms (five deterministic and one randomized) to the proposed algorithms. The space requirements of these algorithms may be 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 – #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
Table 3.3: Space Requirements of List Ranking Algorithms

<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, CV88a]</td>
<td>8n</td>
<td>expander graph (3n)</td>
<td>26n</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 [DH92]</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>

is $O(n)$ for all the algorithms, but the variation in the constant factor is shown in the first column of table 3.3.

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.
The column labeled OTHERS gives the space required by other data structures. The numbers in parentheses there indicate the minimum amount of space required. The last column indicates the minimum total space required for the implementation of the algorithm.

As shown table 3.3, the proposed deterministic algorithm requires significantly less space than all but one algorithm. Also, the randomized implementation requires less space than the well-known randomized algorithm [7].

3.6 Conclusion

Although several work-optimal linked list ranking algorithms are known for the EREW PRAM model, the deterministic and randomized algorithms proposed here provide a simple method of list ranking which is efficient in both time and space requirements. It still remains to be determined if the deterministic algorithm can be modified to achieve work-optimal time.