Deterministic Parallel List Ranking

Richard J. Anderson* Gary L. Miller†

Abstract

In this paper we describe a simple parallel algorithm for list ranking. The algorithm is deterministic and runs in $O(\log n)$ time on EREW P-RAM with $n/\log n$ processor. The algorithm matches the performance of the Cole–Vishkin [CV86a] algorithm but is simple and has reasonable constant factors.

1 Introduction

List ranking is a fundamental operation on lists. The problem is: given a linked list, compute the distance each cell is from the end of the list. The problem can be solved by a straightforward sequential algorithm that traverses the list. However, the problem is much more difficult to solve efficiently in parallel. The problem was first proposed by Wyllie [Wyl79] who gave an algorithm that ran in $O(\log n)$ time using n processors. A substantial amount of effort has been put in to finding a deterministic parallel algorithm that achieves the same time bound with $n/\log n$ processors. Such an algorithm would be optimal in the sense that it would have a time processor product equal to that of the sequential algorithm. The result of Wyllie has been gradually improved in a series of papers [Vis84] [WH86] [KRS85] [MR85] [CV85], until Cole and Vishkin [CV86a] succeeded in giving an optimal algorithm with $O(\log n)$ runtime on an EREW P-RAM. The draw back to their algorithm is that it is complicated and has very large constant factors. They rely on an expander graph construction to solve a scheduling problem that arises. In this paper we give an algorithm that runs in $O(\log n)$ time and uses $n/\log n$ processors. Our algorithm is much simpler, and does not rely on an expander graph construction.

There are two main reasons why the parallel list ranking problem has attracted so much attention. First of all, list ranking is a fundamental operation that has many applications. List ranking can be used to compute a prefix sum for any associative operation over a list; this can either be done by using the ranks to reorganize the list into an array and using an efficient algorithm for data independent prefix sum, or by

*Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195, USA. Supported by an NSF Presidential Young Investigator Award.

†Department of Computer Science, University of Southern California, Los Angeles, California 90089, USA. Supported by NSF Grant DCR-8514961

embedding the associative operation into the list ranking algorithm. Another application of list ranking is to perform traversal operations, such as preorder numbering, on trees [TV85]. The problem of expression evalution on trees can be reduced to list ranking [GMT86] which leads to optimal algorithms for evaluating certain types of expressions. The second reason for looking at parallel list ranking is that the problem has been a rich source for ideas about parallel algorithms in general. For example certain arbitration techniques have been developed for list ranking that have turned out to have much wider an application [GPS87]. Many ideas about methodologies for parallel algorithms and schelduling have come out of this work.

Our list ranking algorithm follows the same scheme as other efficient list ranking algorithms. We first describe the general scheme and then give the basic version of our algorithm. Although the first algorithm is correct, it does not achieve the $O(\log n)$ running time because the processor workload may be unbalanced. We show how to fix the problem by attending to certain details. The final subsection gives the analysis of the algorithm which establishes it as an optimal algorithm.

2 List ranking Algorithms

The standard parallel algorithm for list ranking is due to Wyllie and runs in $O(\log n)$ time using n processors. The algorithm uses a path-doubling strategy. Each cell v contains a pointer $D(v)$ and at each time step the assignmemt $D(v) := D(D(v))$ is made. The distance covered by a pointer doubles at each time step, so in $\log n$ steps, all pointers are at the end of the list. It is straight forward to embed the computation of the distance to the end of the list in this process. The drawback of this algorithm is that the *work* (time–processor product) performed is $O(n \log n)$ as opposed to $O(n)$ work for the natural sequential algorithm. The goal of subsequent work on the list ranking problem has been to reduce the processor requirement while still maintaining a run time of approximately $O(\log n)$.

The basic step of the list ranking algorithm can be viewed as splicing out an element from the list. When the pointer $D(v)$ is replaced by $D(D(v))$ the cell $D(v)$ is removed from the list starting at v. The source of the inefficiency in Wyllie's algorithm is that the same cell is spliced out of a number of different lists instead of being left alone once it is spliced out. The basic approach to get improved list ranking algorithms has been to splice out a large number of the list cells and then solve the list ranking problem on the smaller list. Most of the list ranking algorithms subsequent to Wyllie contain the following steps:

1. Splice out elements from the list so that $O(n/\log n)$ elements remain.
2. Solve the list ranking problem on the reduced list with Wyllie's algorithm.
3. Reconstruct the entire list by processing the elements that were spliced out.

Step 2 can be done in $O(\log n)$ time and $n/\log n$ processors since the list has been reduced in length. Step 3 can also be done in $O(\log n)$ time with $n/\log n$ processors.

The details of this step depend on the data structure used to represent the spliced out elements. However, standard parallel techniques suffice for this step. The key step in order to have an optimal parallel algorithm is step 1 which must also be solved in $O(\log n)$ time with $n/\log n$ processors. It is this step that has been gradually improved in the series of papers on list ranking and is the step that we address in our algorithm.

There are two major issues that arise in the algorithms to splice elements out of the list. The issues are identifying elements to remove and resolving contention. In order to perform the splice out phase efficiently, it is necessary to have fewer processors than list cells, so processors must identify cells to work on. As cells are removed, processors must be reallocated and find cells still in the list to work on. Naive strategies run into the problem that as elements get removed from the list, it becomes more difficult to find elements still in the list to work on. Various scheduling and reorganizing schemes have been developed to get around this difficulty. Note that to get an optimal $n/\log n$ processor algorithm, a constant fraction of the processors must succeed in finding an element to remove from the list at each time step. A feature of our algorithm is a rather simple scheduling strategy that allows the processors to keep busy with cells still in the list.

The most frequently used reorganization scheme is to move the remaining cells to the start of the array [Vis84] [WH86] [KRS85] [CV86b] [MR85]. The basic way to do this is to compute a prefix summation of the number of live cells in the array, and then move cells forward. The summation can become a bottleneck since it takes $O(\log n)$ time on an EREW P-RAM. The summation can be performed less than $O(\log n)$ time on the more powerful CRCW P-RAM [Rei85] [CV86b], so this step need be dominant. An alternate approach proposed by [MR85] is to compress with a probabilistic routine that does not give perfect compression, but does concentrate the live cells sufficiently so that they can be easily identified. A different approach to allocating the work is to have processors randomly probe into the array to find cells to work on [Vis84]. To construct their optimal deterministic list ranking algorithm Cole–Vishkin introduced a complicated rescheduling scheme based on expander graphs [CV86a]. All of these schemes can be viewed as dynamic rescheduling, since the processor assignments depend upon the data. In contrast to these algorithms, our algorithm relies upon a static scheduling of the list cells to the processors.

The second issue in splicing out elements is avoiding contention. Two adjacent list cells can not be spliced out at the same time. A natural strategy to resolve contention is with randomization [MR85] [Vis84]. When processors identify adjacent cells to remove, random bits are chosen and then a protocol is used that insures that adjacent cells are not selected. The deterministic case for resolving contention is more difficult. Cole–Vishkin and Han [CV85] [Han87] independently developed deterministic protocols based on the cells' addresses to resolve contention. The Cole-Vishkin method (deterministic coin-tossing) finds a subset of the list cells such that no adjacent cells are in the subset and every cell is within a distance of k of an element of the subset. The subset is referred to as a k-ruling set. They show that a 3-ruling

set can be constructed in $O(\log^* n)$ time and a $\log^{(k)} n$-ruling set can be constructed in $O(k)$ time. Our algorithm will use their subroutine for constructing a $\log \log n$-ruling set.

3 Basic Algorithm

Our list ranking algorithm splices items out of the list until $n/\log n$ items remain. When the algorithm is reduced to $n/\log n$ items, Wyllies algorithm can be applied with one processor per list cell. The ranks of the items can then be computed by adding them back into the list in the reverse of the order that they were removed.

One way to view our algorithm is that the processors make probes into the list to find cells to work on. The probing is done in a manner that assures that the processors making probes find distinct cells that have not been removed. If a processor identifies a cell with neither of its neighbors identified by other processors, then the cell can be spliced out immediately. The difficult case is when the cell identified by a processor is adjacent to a cell identified by another processor. The processors cannot splice out their cells simultaneously. One of the key components of our algorithm is the manner in which contention is resolved.

We refer to a set of adjacent cells that are selected by processors as a *chain*. The first cell is the *ruler* and the remaining cells are *subjects*. It is easy to determine which cells are rulers and which cells are subjects in constant time. All of the work of removing the subject cells is assigned to the processors of the ruler. The processors of the subject cells are released and go on to work on other cells. A ruler removes its subjects by splicing them out one at a time. This removal takes place while other cells are probing the list and determining whether they are rulers or subjects.

The algorithm is divided into *stages*. During a stage, a processor may remove a cell from the list or find another cell to work on. A stage can be performed in constant time. Each ruler removes one of its subjects in a stage. If a ruler removes its last subject, it becomes *active* and will participate in the reallocation phase, otherwise it waits for the next stage to remvoe its next subject. After the rulers have removed subjects, all active processors pick cells and arbitrate to find rulers and subjects. At this time, the selected cells not adjacent to other selected cells are spliced out. The processors associated with subject cells are set to being active at the next stage so that they are reallocated.

The allocation of processors to list cells is done in a static manner. Initially, the memory is divided up into $n/\log n$ blocks of $\log n$ items each. Each processor is assigned one of these blocks which we refer to as its *queue*. The items in a queue are not necessarily adjacent in the list. A processor acts upon the items in its queue in order, starting with the first item. We refer to the number of items left in a queue as its *height*, and the item currently under consideration as its *head*. There are two ways that an item can cease to be the head of a queue, it can be spliced out, or it can become a subject. In either case, the processor takes the next item in the queue as its new head. If a queue becomes empty, then the processor ceases to work.

We give a more formal description of this algorithm by giving the code for a single stage. Each processor runs the following code. The queue for the processor is represented by an array, with *head* the location of the queue head. The list is assumed to be doubly linked. The cell p has pointers *p.next* and *p.prev* to its successor and predecessor respectively. A cell also has a status field, which can have the value *ruler, subject, active, inactive* or *removed.* Initially, all cells have *inactive* status, active status indicates that a processor is assigned to them.

The code breaks into the three indicated parts. The first part is where the rulers can remove cells, the second part is to identify new rulers and subjects, and the last part is to advance the queue where necessary.

```
Stage
  p := Q[head];
                        -- Rulers splice out cells
  if p.status = ruler then
    SpliceOut(p.next);
    if p.next.status ≠ subject then
      p.status := active;
    end
  end
                        -- Identify new rulers
  if p.status = active then
    if p.prev.status ≠ active ∧ p.next.status ≠ active then
      SpliceOut(p); p.status := removed;
    else if p.prev.status = active then
      p.status := ruler;
    else
      p.status := subject;
    end
  end
                        -- Advance queue heads
  if p.status = removed ∨ p.status = subject then
    head := head − 1;
    if head < 0 then
      stop
    else
      Q[head].status := active;
    end
  end
```

It must be shown that this algorithm correctly splices out the list elements and that $O(\log n)$ stages are sufficient to reduce the list to $n/\log n$ cells. The correctness of the algorithm follows from the facts that adjacent cells are never removed at the same time, and every cell is eventually looked at by a processor. The run time is more difficult to establish, (especially since, as currently stated, it is greater than $O(\log n)$.) The following arguement shows that as long as most of the processors have nonempty queues, then the algorithm is performing work at a reasonable rate. We do this by

establishing a simple accounting scheme for the work done. There are two ways that a cell can be spliced out, it can either be removed as an isolated cell, or by becoming a subject and then removed by a ruler. When an isolated cell is spliced out, we assign the processor one unit, when a ruler splices out a cell we assign the processor one half unit and when we make a cell a subject we assign the processor one half unit. Thus, for each cell that is removed, one unit is assigned. At the start of a stage, each processor is associated with a cell that has status *ruler* or status *available*. If there are k *available* cells, at least $k/4$ units are assigned, since the number of subjects created is at least as large as the number of rulers created. If all processors were to remain busy, this argument would show that $4 \log n$ phases would suffice to remove all of the cells from the list. However, some processors could exhaust there queues, while others had made very little progress. One bad case that could occur is for a ruler to have a large number of subjects (as many as $n/\log n$) which would take a very long time to remove. To solve this problem, we alter the selection of rulers, so that a ruler never has too many subjects, and the cells that are made rulers tend to be lower down in their queues.

4 Load Balancing

In the previous section we saw that the problem with our proposed algorithm is that some queues could be delayed working on long chains, while others could finish quickly and become idle. The solution to this problem has two parts, we insure that the chains do not become too long and also insure that the shorter queues are assigned work as opposed to the longer queues. The major tool that we use for this is the ruling set algorithm of Cole and Vishkin [CV85].

Definition 4.1 *In a list L, a k-ruling set is a subset R of L such that there are no adjacent elements in R, and every element of L is within distance k of an element of R.*

Cole and Vishkin give an efficient parallel algorithm for constructing a ruling set. We use a version of their algorithm that finds a $\log \log n$ ruling set in a list of length n in constant time with one processor assigned to each element.

The process of identifying rulers consists of taking the sets of adjacent elements that are picked by processors at that phase. In the original algorithm, we just take the first cell in a chain as a ruler, and the remaining cells as subjects. To balance the work load, we break up the longer chains that we identify. There are a number of things that we can do while making chains and still have a stage run in constant time. If we break up a chain so that we have some non-adjacent pieces of length one, we can splice them out when they are encountered. We can also traverse the list backwards, instead of forwards to splice out elements. This allows us to have as rulers the cells of minimum height in a chain, and then remove cells working uphill. Finally, we can mark the first and last elements of a chain so that rulers know when to stop

removing items. With the flexibility to perform these operations, it is easy to break the chains into increasing height, decreasing height, and constant height chains. The constant height chains can then broken up into subchains of length at most $\log\log n$ time with the ruling set algorithm. The code given above can be modified to perform these additional tasks.

The effect of the modification is to have the following two constraints on rulers and chains: a ruler never has more that $\log\log n$ subjects, and the height of a ruler is no greater than the height of its subjects. The performance analysis given in the next section shows that these conditions are sufficient to make this an $O(\log n)$ algorithm.

5 Performance Analysis

In this section we prove that the algorithm with the enhanced method of choosing rulers is an $O(\log n)$ algorithm. The proof relies on an accounting scheme that divides the work for removing an item amongst the various steps.

Each cell is assigned a weight related to its height. The i-th item from the top is assigned a weight of $(1-\alpha)^i$ where $\alpha = \frac{1}{\log\log n}$. The accounting scheme is the same as the one sketched above: the removal of an isolated item is the full weight of the item, the removal of an item by a ruler is half the weight, the identification of a subject is half the weight, and the identification of a ruler is 0. We show that each stage reduces the total weight by a factor of at least $1-\frac{\alpha}{4}$. This allows us to bound the number of phases that are required to reduce the number of items to $\frac{n}{\log n}$.

Claim 5.1 *A single phase of the algorithm reduces the weight by a factor of at least* $1-\frac{\alpha}{4}$.

To facilitate the argument, we use the following bookkeeping trick: the weight of a queue is considered to be the weight of the remaining elements, plus the remaining weight of the subjects of the queue head. In Figure 1, queue 1 has weight $(1-\alpha)^2 + (1-\alpha)^3 + (1-\alpha)^4 + \frac{1}{2}(1-\alpha)^2 + \frac{1}{2}(1-\alpha)^1$, with the first three terms coming from queue 1, the fourth term coming from queue 2 and the fifth term coming from queue 3.

We cover the three major cases separately. First we look at what happens when an isolated cell is removed. Suppose that we remove a cell of wieght $(1-\alpha)^i$. The queue has weight at most $\sum_{i\le j<\log n}(1-\alpha)^j < \sum_{i\le j}(1-\alpha)^j = (1-\alpha)^i\frac{1}{\alpha}$. Thus, the factor of reduction is

$$\frac{\sum_{i+1\le j<\log n}(1-\alpha)^j}{\sum_{i\le j<\log n}(1-\alpha)^j} = 1 - \frac{(1-\alpha)^i}{\sum_{i\le j<\log n}(1-\alpha)^j} < 1-\alpha$$

The second case is when a vertex is identified as a subject. We account for the weight of all of the queues associated with the chain. The queues that have cells that become subjects lose one cell each and the queue that has the ruler picks up the cells. Each cell that becomes a subject has half of its weight removed by the accounting.

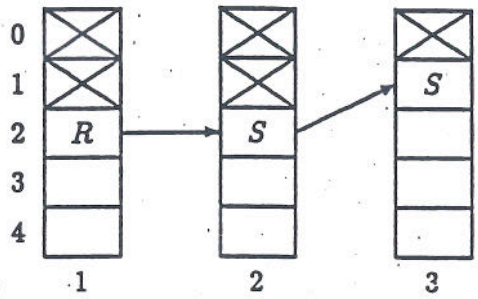

Figure 1: Figure 1. Computing the weight of a chain

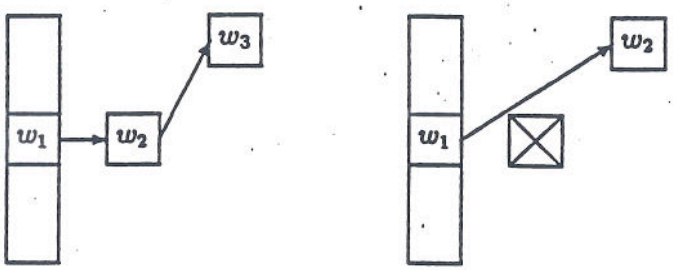

Figure 2: Figure 2. Rearranging weights

Suppose the weight of the ruler is $(1-\alpha)^{i_1}$, and the weight of the subjects are $(1-\alpha)^{i_j}$ for $2 \leq j \leq k$. Since the ruler is chosen to be the element of minimum height, we have $i_1 \leq i_j$. Let $Q_j = \sum_{i_j \leq l < \log n}(1-\alpha)^l$. The factor of reduction is:

$$1 - \frac{\frac{1}{2}\sum_{2\leq j\leq k}(1-\alpha)^{i_j}}{\sum_{1\leq j\leq k} Q_j} < 1 - \frac{\alpha}{2}\frac{\sum_{2\leq j\leq k}(1-\alpha)^j}{\sum_{1\leq j\leq k}(1-\alpha)^j} \leq 1 - \frac{\alpha}{4}.$$

The final case to consider is when a ruler removes a cell. In this case we have to account for the weight of the queue as well as the weight of the chain of subjects. We shall perform the accounting as if the heaviest element was removed (even though it is not necessarily the case). It can be simulated by rearranging the weights as indicated in the Figure 2.

Suppose the ruler has weight $(1-\alpha)^i$ and the subjects have weight $(1-\alpha)^{i_1}, \ldots, (1-\alpha)^{i_k}$, where $(1-\alpha)^{i_k}$ is the maximum weight and $k \leq \log\log n$. (Finally the value of α and the length's of the chains come into play!) The fraction of reduction is:

$$1 - \frac{\frac{1}{2}(1-\alpha)^{i_k}}{\sum_{i\leq j<\log n}(1-\alpha)^j + \frac{1}{2}\sum_{1\leq j\leq k}(1-\alpha)^{i_j}} \leq 1 - \frac{(1-\alpha)^{i_k}}{\frac{2}{\alpha}(1-\alpha)^{i_k} + (1-\alpha)^{i_k}\log\log n} = 1 - \frac{\alpha}{3}.$$

We can now state the main theorem concerning the runtime of this algorithm.

Theorem 5.2 *The number of list cells remaining after* $5 \log n$ *stages is at most* $n/\log n$.

Proof: At the start of the algorithm, the total weight of all of the items is:

$$\frac{n}{\log n} \sum_{0 \le i < \log n} (1-\alpha)^i < \frac{n}{\log n} \frac{1}{\alpha}.$$

Since the weight of the smallest item is $(1-\alpha)^{\log n - 1}$, if we show that the total weight is reduced to at most $\frac{n}{\log n}(1-\alpha)^{-\log n+1}$ at time $t = 5 \log n$, we will have established that there are at most $n/\log n$ items. Since the total weight is reduced by a factor of at least $1 - \frac{\alpha}{4}$ each stage, the weight at time t is at most

$$\frac{n}{\log n} \frac{1}{\alpha}(1-\frac{\alpha}{4})^t \le \frac{n}{\log n} \log\log n (1-\frac{\alpha}{4})^{5 \log n} < \frac{n}{\log n}(1-\alpha)^{\log n}.$$

∎

6 Conclusions

We have shown that list ranking can be solved by a relatively simple optimal algorithm. It is important to pursue the idea further and develop truely practical list ranking algorithms. In practice, Wyllie's algorithm is still probably the best deterministic algorithm, although we believe that our algorithm is a contribution towards finding a better practical algorithm. In this paper we have considered the synchronous P-RAM model. An important area of research is to look at the list rank problem in other models such as asynchronous models [LG87].

References

[CV85] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th ACM Symposium on Theory of Computation*, pages 206–219, 1985.

[CV86a] R. Cole and U. Vishkin. Approximate scheduling, exact scheduling, and applications to parallel algorithms. In *27th Symposium on Foundations of Computer Science*, pages 478–491, 1986. Part I to appear in SIAM Journal of Computing.

[CV86b] R. Cole and U. Vishkin. *Faster Optimal Parallel Prefix Sums and List Ranking.* Technical Report 56/86, Tel Aviv University, December 1986.

[GMT86] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in the EREW model. 1986. Extended abstract.

[GPS87] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th ACM Symposium on Theory of Computation*, pages 315–324, 1987.

[Han87] Y. Han. *Designing Fast and Efficient Parallel Algorithms.* PhD thesis, Duke University, 1987.

[KRS85] C. Kruskal, L. Rudolf, and M. Snir. The power of parallel prefix computation. In *International Conference on Parallel Processing*, pages 180–185, 1985.

[LG87] B. D. Lubachevsky and A. G. Greenberg. Simple, efficient asynchronous parallel prefix algorithms. In *International Conference on Parallel Processing*, pages 66–69, 1987.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[Rei85] J. H. Reif. An optimal parallel algorithm for integer sorting. In *26th Symposium on Foundations of Computer Science*, pages 496–503, 1985.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[Vis84] U. Vishkin. Randomized speed-ups in parallel computation. In *Proceedings of the 16th ACM Symposium on Theory of Computation*, pages 230–239. 1984.

[WH86] R. A. Wagner and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *International Conference on Paralle Processing*, pages 924–930, 1986.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Department of Computer Science, Cornell University, 1979.