A SIMPLE OPTIMAL RANDOMIZED PARALLEL LIST RANKING ALGORITHM

by

Karl Abrahamson
Norm Dadoun
David Kirkpatrick
Teresa Przytycka

Technical Report 87-14

May, 1987

A Simple Optimal Randomized Parallel List Ranking Algorithm *

Karl Abrahamson Norm Dadoun
David Kirkpatrick Teresa Przytycka

Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada

Abstract

We describe a randomized parallel algorithm to solve list ranking in $O(\log n)$ expected time using $n/\log n$ processors, where n is the length of the list. The algorithm requires considerably less load rebalancing than previous algorithms.

Keywords: Parallel algorithms, list ranking, randomization.

1 Introduction

This paper is concerned with a parallel algorithm for the parallel random access machine (PRAM) model. Several versions of such machines are described in the literature, the major difference between them being whether they permit concurrent reading or concurrent writing (or both) of a memory cell by two or more processors. The version presumed here is the most restrictive one, the exclusive-read, exclusive-write (EREW) PRAM.

The list ranking problem is stated as follows. The input is a linear linked list of n cells contained in an array of n cells. The list cells can be in any order in the array. The problem is to assign to each list cell its distance from the end of the list, measured along the linked list.

List ranking is encountered in parallel algorithms for a number of problems. It is a fundamental part of the Euler tour technique which has been used to compute biconnected components [7] and strong orientation of a graph [8], and to evaluate expressions [1,2].

The "standard" parallel list ranking algorithm employs the fundamental recursive doubling technique. Each list node is assigned a processor. Each node v also has a variable $d(v)$ which is initially set to 1, and which represents the distance ahead in the list that its link $l(v)$ currently points. The last node in the list points to itself, and its distance is 0. The basic step

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada

is for each node v, synchronously in parallel, to set $d(v) \leftarrow d(v) + d(l(v))$ and $l(v) \leftarrow l(l(v))$. That step is repeated until the link of each node points to the end of the list. Since each distance is doubled at each step (until the link points to the end of the list), this algorithm takes $O(\log n)$ time using $O(n)$ processors.

The standard algorithm is suboptimal in the sense that the product of the number of processors and the time is $O(n \log n)$, although the problem is solvable sequentially in linear time. The challenge is to find a fast algorithm which achieves optimal speedup, i.e. a linear number of total operations. An ideal algorithm would take $O(\log n)$ time using only $n/\log n$ processors.

A step in that direction is taken by Kruskal, Rudolf and Snir [5], who show how to solve list ranking in $O(n^\epsilon)$ time using $n^{1-\epsilon}$ processors, for any $\epsilon > 0$, thus achieving optimal speedup.

Vishkin [9] suggests viewing list ranking as a parallel prefix computation problem. Each node has a value of 1, and each prefix sum in the reversal of the list is to be computed. There is a well known parallel prefix algorithm which takes $O(\log n)$ time using $n/\log n$ processors (see [9]). Unfortunately, that algorithm requires that the numbers to be summed be consecutive in an array. But when the list nodes are in the same order in the array as in the list, the list ranking problem is trivial.

So, when the list nodes are not consecutive in the array, the parallel prefix algorithm cannot be applied directly. Nevertheless, Vishkin [9] describes randomized list ranking algorithms similar in spirit to the parallel prefix algorithm, including one which achieves $O(\log n \log^* n)$ time using $n/(\log n \log^* n)$ processors. A crucial feature of this and subsequent efficient list ranking algorithms that distinguish them from parallel prefix algorithms is the necessity of spending time balancing the work load among the processors. Indeed, if it were not for the need for rebalancing, Vishkin's algorithm would be considerably simpler and would achieve $O(\log n)$ time with $n/\log n$ processors.

Cole and Vishkin [4] achieve the same time and processor bounds for list ranking as Vishkin [9], using a deterministic algorithm. Again, load rebalancing is the costliest part of the algorithm. Miller and Reif [6] describe a randomized algorithm which solves list ranking in $O(\log n)$ time using $n/\log n$ processors, involving a substantial and difficult rebalancing operation. Finally, Cole and Vishkin [3] describe a deterministic algorithm which achieves $O(\log n)$ time using $n/\log n$ processors. This algorithm is based heavily on a general load balancing scheme, which is sufficiently costly that Cole and Vishkin admit that this algorithm, although good for very large n, is probably not practical.

This paper shows that the extensive global rebalancing inherent in the algorithms above is not really needed for list ranking, at least for randomized algorithms. An algorithm is described which achieves $O(\log n)$ expected time using $n/\log n$ processors, but performs only one global rebalancing, and many simple, local rebalancing operations. The algorithm is conceptually simpler than previous list ranking algorithms with competitive time and processor bounds.

2 The Algorithm

The basic idea, like that employed in previous algorithms, is to reduce list ranking to a smaller instance of the same problem. Imagine, for the moment, that each list node has a dedicated processor. A *round* consists of the following operation. Each node tosses an unbiased coin. Any node v which tosses tails, and whose successor in the list tosses heads, is a non-survivor. All other nodes are survivors. The last node in the list is always a survivor.

Notice that no two consecutive list nodes can be survivors. Each survivor v checks whether its successor is a survivor. If not, then v executes a *bypass* operation: v sets $d(v) \leftarrow d(v) + d(l(v))$ and $l(v) \leftarrow l(l(v))$, effectively deleting the non-survivor from the list. Now the reduced list is ranked recursively. When the reduced list is completely ranked, non-survivor u can set $d(u) \leftarrow d(l(u)) + 1$.

This reduction strategy is carried out for enough rounds until the size s of the reduced list has an expected value well below $n/\log n$. The reduced list is collapsed into an array of size s (thereby facilitating global rebalancing). If the reduced list has more than $n/\log n$ nodes (a highly improbable event) then more rounds are employed to reduce the size to at most $n/\log n$. At this point the standard algorithm is invoked on the reduced list.

If there are actually only $n/\log n$ processors available for the above algorithm, then each processor must be responsible for about $(N/n)\log n$ nodes, where N is the current size of the list. It is crucial that the load be fairly well balanced among the $n/\log n$ processors. Our basic strategy is to maintain that balance locally as follows. The original array (of size n) is divided into blocks of size $m = \lceil 2\log^2 n \rceil$, each block consisting of m contiguous locations *in the array*. Approximately $h = (\log n)/2$ processors are assigned to each block. The processors assigned to each block remain the same as the algorithm progresses.

There is no redistribution of processors between blocks. But the processors within each block carefully share the load of that block among themselves. It will be convenient to collect the rounds into groups of 10, and to

call each such group of rounds a *phase*. After each phase the k_x survivors within block x are compacted into the first k_x locations of the block, and the h processors associated with block x are assigned to equal chunks of the compacted block. Compaction is easily achieved as follows. Using a parallel prefix algorithm, number the survivors from left to right. Number the non-survivors from left to right, starting with the largest survivor number plus one. Compute the new address of each node, using the computed numbers as offsets from the start of the block. This will be done in all blocks in parallel. Then it is easy to update the links, and move each node to its new address.

When the reduction part of the algorithm is finished, all of the survivors in the entire array are compacted to the beginning of the array, and the standard algorithm is run.

The above description is summarized in the pseudo-code below. It is important to maintain synchrony in the algorithm. Some steps take less time for some processors than for others, because the load is not exactly balanced. But each processor knows the maximum length of time that any other processor can take to complete a given step. The notation "[instructions]" means to execute the given instructions, and then to wait until the maximum time for those instructions has expired. The program is easily implemented on a single-instruction, multiple data machine, and it may be helpful to imagine such an implementation.

It is important that the blocks decrease in size, so that the time to process a block decreases exponentially with the number of phases executed. With low but positive probability, a given block does not decrease fast enough, and the processors in that block will not have enough time to deal with all of the survivors. In that case, some of the survivors become *passive*; they retain their current coin toss values and they continue to survive. In short, they are ignored. Initially, all survivors are *active*.

For brevity, the backup part of the algorithm, in which non-survivors are assigned ranks as the recursion backs up, is omitted from the following description.

List Ranking Algorithm

$m \leftarrow \lceil 2 \log^2 n \rceil$ {block size}
$b \leftarrow \lceil n/m \rceil$ {number of blocks}
$h \leftarrow \lfloor n/(b \log n) \rfloor$ {processors per block}
1: $n \leftarrow$ number of survivors. (All survivors are active.)
for each block in parallel **do** (using h processors)
  for $\phi \leftarrow 1$ **to** $\lceil \log \log n \rceil$ **do**
    for $r \leftarrow 1$ **to** 10 **do**
      [Assign a coin toss to each active survivor in the block.]

[Mark each active survivor which tossed tails, and whose successor tossed heads, as a non-survivor.]
[Bypass each non-survivor.]
od
od
[Compact the first $m2^{-\phi}$ survivors to the front of the block.]
(Remaining survivors become passive.)
od
Compact the survivors in the entire array to the front.
if $> n/\log n$ survivors then goto 1.
Run the standard algorithm on the remaining survivors.

3 Analysis

We do a quite crude but simple analysis. A *round* is one iteration of the r-loop. A *phase* is one iteration of the ϕ-loop.

In what follows we can afford to ignore the special case of the end of the list, since there is only one such node. Also, we will presume that all of the survivors in each block are active throughout the execution of the ϕ-loop. We will see that the probability of that failing to occur is negligibly small.

Consider a particular block B containing s survivors. Choose k arbitrary survivors from B. Let R be the probability that those k nodes survive one more round, and P be the probability that they survive one more phase.

The k given nodes are spread in some unknown fashion in the current list. Considering the list to be a directed graph, the subgraph induced by the k chosen nodes consists of a collection $C_1, \ldots, C_t$ of chains. Let l_i be the length (number of nodes) of C_i, for $i = 1, \ldots, t$.

Choose an arbitrary i, and let C_i' be chain C_i with one node added to the end. In order for every node in C_i to survive one more round, the sequence of coin tosses assigned to the nodes in C_i' must be in H^*T^*, where H is heads and T is tails. So the probability that all k chosen nodes survive one more round is

$$R = \prod_{i=1}^{t} \frac{l_i + 2}{2^{l_i+1}}.$$

But $\frac{n+2}{2^{n+1}} \leq \left(\frac{3}{4}\right)^n$ for $n \geq 1$, so $R \leq \left(\frac{3}{4}\right)^k$, and $P \leq \left(\frac{3}{4}\right)^{10k}$.

Let Q be the probability that *any* $\lceil s/2 \rceil$ of the s survivors of block B survives one more phase. Then $Q < \binom{s}{\lceil s/2 \rceil}\left(\frac{3}{4}\right)^{10(s/2)} < 2^s\left(\frac{3}{4}\right)^{5s} < 2^{-s}$.

As long as a block has at least $2 \log n$ survivors, its size is cut by a factor of $\leq 1/2$ in the next phase with probability $> 1 - \frac{1}{n^2}$. So the probability that *any* block containing $\geq 2 \log n$ survivors fails to be cut by a factor of $\leq 1/2$ in *any* of the $\lceil \log \log n \rceil$ phases is less than $\lceil \log \log n \rceil / n$, which is negligibly small.

Hence, with high probability, every block has size $\leq m2^{-\phi}$ at the end of phase ϕ. That justifies the presumption that all of the survivors are active. The probability that there are more than $n/\log n$ survivors at the end of the ϕ-loop is sufficiently small that the loop formed by the goto has negligible contribution to the expected time.

The expected time required for phase ϕ is $O(\frac{m2^{-\phi}}{\log n}) = O(2^{-\phi} \log n)$, since the upper bound on the number of active survivors in a given block in phase ϕ is $m2^{1-\phi}$, and there are $\Theta(\log n)$ processors assigned to each block. So the total expected time for the ϕ-loop is $O(\sum_k 2^{-k} \log n) = O(\log n)$, and the total time is $O(\log n)$.

4 Conclusion

The fundamental difficulty in list ranking with few processors seems to be keeping the load balanced among the processors. We have shown that a modest amount of rebalancing suffices. In a sense, the load is balanced automatically. Note that if the blocks had been chosen smaller in our algorithm, the load would not remain balanced, and some other form of rebalancing would be necessary.

We believe that our algorithm compares favorably with other optimal list ranking algorithms, both in terms of simplicity and efficiency. It is, however, a randomized algorithm, and hence cannot be directly compared to deterministic algorithms. Our approach appears to rely heavily on randomization to keep the load approximately balanced. It seems that any optimal deterministic algorithm must do extensive load rebalancing.

References

[1] K. Abrahamson, N. Dadoun, D. Kirkpatrick, and T. Przytycka. A Simple Parallel Tree Contraction Algorithm. 1987. Manuscript, Department of Computer Science, University of British Columbia.

[2] R. Cole and U. Vishkin. *The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time.* Computer Science Department Technical Report 242, Courant Institute, 1986.

[3] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *27th Annual Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[4] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th Annual Symposium on Theory of Computing*, pages 206–219, 1986.

[5] C. P. Kruskal, L. Rudolf, and M. Snir. Efficient parallel algorithms for graph problems. In *International Conference on Parallel Processing*, pages 180–185, 1985.

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

[7] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. on Comput.*, 14(4):862–864, 1985.

[8] U. Vishkin. On efficient parallel strong orientation. *Inf. Process. Lett.*, 20:235–240, 1985.

[9] U. Vishkin. Randomized speedups in parallel computation. In *16th Annual Symposium on Theory of Computing*, pages 230–239, 1984.