A Simple Randomized Parallel Algorithm for List-Ranking

Richard J. Anderson* Gary L. Miller†

Abstract

In this paper we give simple parallel algorithm for the list-ranking problem. The algorithm is a randomized $O(\log n)$ time, $n/\log n$ processor algorithm for an EREW P-RAM. The algorithm is substantially simpler than other optimal algorithms for list-ranking. Keywords – parallel processing, analysis of algorithms.

1 Introduction

A major goal in the design of parallel algorithms is to achieve substantial speed up without sacrificing processor efficiency. The most desirable parallel algorithms are the ones with processor–time product equal to the sequential complexity of the problem. These algorithms are referred to as *optimal* parallel algorithms since little penalty is incurred with the introduction of parallelism.

In this paper, we present a simple optimal algorithm for list-ranking. The list-ranking problem is: given a linked list in memory, compute the distance that each cell is from the end of the list. This problem can be solved sequentially in linear time by traversing the list and recording the distances. The parallel complexity of this problem has attracted considerable attention for a number of reasons. The problem is a fundamental data structure problem and is equivalent to a number of other problems. List-ranking is an important subroutine in parallel algorithms for such problems as expression evaluation [1], [4], tree traversal [7] and connected components. The list-ranking problem has become an important test bed where ideas for efficient parallel algorithms have been developed and improved.

The list-ranking problem was introduced by Wyllie [9]. He gave an $O(\log n)$ time, n processor algorithm for the problem. The bound was gradually improved through a series of results [8], [5], [3] until Cole and Vishkin gave an $O(\log n)$ time, $n/\log n$ processor deterministic algorithm [2]. The Cole-Vishkin result disproved a conjecture [9] that an $O(\log n)$ time optimal list-ranking algorithm did not exist. The Cole-Vishkin algorithm is an extremely

*Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195, USA
†Department of Computer Science, University of Southern California, Los Angeles, California 90089, USA

complicated algorithm. It requires a number of different stages and at c
pander graphs to handle a scheduling problem that arises. The other pi
optimal list-ranking algorithm is a randomized algorithm due to Miller and .
result is a corollary to their more general result on expression evaluation.

Our list-ranking algorithm is a randomized algorithm that runs in $O(\log n)$ time
$O(n/\log n)$ processors. The algorithm is much simpler than the other optimal algorith
[2], and would be substantially faster in practice. The algorithm is for the EREW P-h
so that it relies on exclusive reads and writes to memory.

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 $v.next$ and at each time step the assignment $v.next := v.next.next$ 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 runtime 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 $v.next$ is replaced by $v.next.next$ the cell $v.next$ 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 approach that we take is to splice elements out of the list until all elements have been removed. We then reconstruct the list and compute the distances of the cells.

2 List-ranking Algorithm

Our algorithm consists of two phases. The first phase splices cells out of the list and the second phase reconstructs the list. The reconstruction phase is simpler than the first phase, since the reconstruction is performed by undoing the work performed in the first phase and it is not difficult to retain this information. The first phase removes all of the cells from the list in $O(\log n)$ time. Since there are n cells and $O(n/\log n)$ processors, $O(n/\log n)$ cells must be removed per phase. There are two major considerations in removing a large number of cells. The first is identifying a large number of cells that have not been removed. We do this by a fairly natural approach in scheduling work to be done. Although our solution to this is relatively simple, the problem of finding available cells has lead to considerable complications and inefficiencies in other list-ranking algorithm. The second issue that arises is to avoid contention in removing cells. Adjacent cells cannot be removed in the same phase. To avoid this problem, we use a simple randomized protocol.

We now describe the list-ranking algorithm in detail. We assume that the list is doubly linked, so that a list cell v has fields $v.next$ and $v.prev$. It is not difficult to set up the backward links in $O(\log n)$ time with $n/\log n$ processors.

The basic structure of the algorithm is to divide the list elements into groups of $\log n$

and assign a processor to each group. The groups consist of consecutive cells from memory, there is no assumption that the elements in a group are adjacent in the list. Each processor is responsible for splicing out the elements in its group or *queue*. A processor will work from the head of the queue, attempting to splice out the top element. When a processor succeeds in splicing out an element, it goes on the next element in its queue.

At a time step, each processor takes the cell from the head of its queue and attempts to remove it from the list. Let v be a list cell, and let $v.next$ and $v.prev$ denote the cells adjacent to v in the list. If v is at the head of the queue of processor p, then p attempts to remove v with the assignments $v.prev.next := v.next$ and $v.next.prev := v.prev$. The only thing that could go wrong is if $v.prev$ or $v.next$ is also a queue head for a different queue. When two adjacent list cells are at the heads of separate queues, it is necessary to arbitrate to find out which one can be removed. The following simple, probabilistic scheme can be used. When adjacent queue heads arise, each processor chooses independently and uniformly a one or a zero. These bits can be thought of as being associated with the list cells at the heads of the queues. Now, if a cell has a one, and the next cell has a zero, the cell with a one can be removed from the list. It is clear that adjacent elements will never be removed at the same time with this scheme. Each list cell v has a field $v.dist$ that records the distance covered by $v.dist$. This value is initially one and is updated whenever the adjacent list cell is removed. The value of $v.dist$ is used in the reconstuction phase.

The probability that a processor gets to remove the cell from the top of its queue in a phase is at least $\frac{1}{4}$. Thus, the expected time for a queue to become empty is at most $4 \log n$. In the next section we show that the expected time for all the queues to become empty is $O(\log n)$.

The first phase splices out elements until all of the list cells are removed. When that point is reached the second phase is run to rebuild the list and compute all of the distances. This is essentially the reverse of the first phase. The cells are put back into the list in the reverse order of their removal. The distance a cell is from the end of the list can be computed when it is put back into the list. Let $v.rank$ be the distance to the end. When v is added to the list, the assignment $v.rank := v.dist + v.next.rank$ suffices for the rank computation.

3 Analysis

In this section we establish a bound on the expected number of phases that are needed to remove all elements from the queue. The number of items removed from the queue for processor p at time t can be represented as a random variable $X_{p,t}$. We wish to establish that for $t = c \log n$, the probability that $X_{p,t} < \log n$ is small. The random variable $X_{p,t}$ can be thought of as a sum of random variables $Z_{p,1}, \ldots, Z_{p,t}$ where $Z_{p,i}$ is 0 or 1 corresponding to whether or not an item is removed. These random variables are not independent since the chances of being able to remove a cell depends upon which cells are queue heads. However, as long as $X_{p,t} < \log n$, (i. e. the queue is not empty), $\text{Prob}[Z_{p,t} = 1] \geq \frac{1}{4}$. It is not difficult to show that for $k \leq \log n$, $\text{Prob}[X_{p,t} < k] < \text{Prob}[\sum_{1<i\leq t} \tilde{Z}_i \geq k]$ where $\tilde{Z}_i, \ldots, \tilde{Z}_t$ is a family

of independent random variables with $\text{Prob}[\tilde{Z}_i = 1] = \frac{1}{4}$ and $\text{Prob}[\tilde{Z}_i = 0] = \frac{3}{4}$. We can therefore use standard estimates on the distribution of sums of Bernoulli trials to bound the random variables.

Let S_t^q be the sum of t Bernoulli trials with success probablility q. The following bound [6] shows that the probability is small that S_t^q is substantially less than its expected value of qt:

$$\text{Prob}[S_t^q < (1-\gamma)qt] < e^{-\frac{\gamma^2 qt}{2}} \quad \text{for } 0 < \gamma < 1.$$

If we take $q = \frac{1}{4}$, $t = 16 \log n$, and $\gamma = \frac{3}{4}$, we have:

$$\text{Prob}[\sum_{1 \le i \le 16 \log n} \tilde{Z}_i < \log n] < e^{-\frac{9}{8} \log n} < \frac{1}{n}.$$

The probability that a particular queue is not empty at time $16 \log n$ is less than $\frac{1}{n}$. Since there are $n/\log n$ queues, the probability that there is a non-empty queue at time $16 \log n$ is less than $\frac{1}{\log n}$. It follows that expected runtime is $O(\log n)$.

It is possible to improve the expected runtime of the algorithm by splicing out elements from the list until only $n/\log n$ cells remain. This will take just a little over $4 \log n$ phases. Then the list-ranking problem can be solved on the $n/\log n$ items with $n/\log n$ processors using Wyllie's algorithm in $O(\log n)$ time.

4 Program

In this section we give the program for the list-ranking algorithm. We give the code for a single processor. Each processor runs the same program. The list cells are represented as records with a number of fields. Each processor has an array of $\log n$ pointers to list cells which serves as the processor's queue and an index which gives the queue head. The first routine is the routine to splice out the cells and the second routine is the routine to reconstruct the list. The processors run the first routine until all of the cells are removed and then they all run the second routine.

The record to represent a list cell is:

```
cell_record=
  record
    next, prev : ↑ cell_record;
    rand : {0,1}              -- Random bit for arbitration
    head : {true, false};     -- True if the cell is a queue head, intially false
    dist : integer;           -- Distance in original list covered by the pointer to next, initially 1
    rank : integer;           -- The distance from the end of the list
    time : integer;           -- The time that the cell was removed from the list
  end

Splice
  t := 1; i := 1; cell := queue[1]; cell.head := true;
  while i ≤ log n do
    cell.rand := Random{0,1};
    if cell.rand = 1 and (cell.next.head = false or cell.next.rand = 0) then
      cell.prev.next := cell.next;
      cell.next.prev := cell.prev;
      cell.prev.dist := cell.prev.dist + cell.dist;
      cell.time := t;
      i := i + 1; cell := queue[i]; cell.head := true
    end
    t := t + 1;
  end
end

Reconstruct
  i := log n; cell := queue[log n]; t := t_max;
  while i ≥ 0 do
    if cell.time = t then
      cell.rank := cell.next.rank + cell.dist;
      i := i − 1; cell := queue[i];
    end
    t := t − 1;
  end
end
```

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *27th Symposium on Foundations of Computer Science*, pages 10–18, 1986.

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

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

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