List Ranking and List Scan on the Cray C-90*

Margaret Reid-Miller
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
mrmiller@cs.cmu.edu
(412) 268-3056

Abstract

List ranking and list scan are two primitive operations used in many parallel algorithms that use list, trees, and graph data structures. But vectorizing and parallelizing list ranking is a challenge because it is highly communication intensive and dynamic. In addition, the serial algorithm is very simple and has very small constants. In order to compete, a parallel algorithm must also be simple and have small constants. A parallel algorithm due to Wyllie is such an algorithm, but it is not work efficient–its performance degrades for longer and longer linked lists. In contrast, work efficient PRAM algorithms developed to date have very large constants. We introduce a new fully vectorized and parallelized algorithm that both is work efficient and has small constants. It does not achieve $O(\log n)$ running time, but we contend that work efficiency and small constants is more important, given that vector and multiprocessor machines are used for problems that are much larger than the number of processors and, therefore, the $O(\log n)$ time is never achieved in practice. In particular, to the best of our knowledge, our implementation of list ranking and list scan on the CRAY C-90 is the fastest implementation to date. In addition, it is the first implementation of which we are aware that outperforms fast workstations. The success of our algorithm is due to its relatively large grain size and simplicity of the inner loops, and the success of the implementation is due to pipelining reads and writes through vectorization to hide latency, minimizing load balancing by deriving equations for predicting and optimizing performance, and avoiding conditional tests except when load balancing.

*This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the Pittsburgh Supercomputing Center (Grant ASC890018P) which provided Cray Y-MP and Cray Y-MP C90 time.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, NSF, Cray Research or the U.S. government.

SPAA 94 - 6/94 Cape May, N.J, USA
© 1994 ACM 0-89791-671-9/94/0006..$3.50

1 Introduction

As production parallel and vector machines become faster and common place, solving larger and larger problems becomes feasible. However, large problems that have irregular sparsity structure or are dynamic are often most efficiently represented and manipulated using lists, trees, and graphs. Use of such data structures has become natural and common on sequential machine, but have been shunned in parallel implementations. Theory indicates that use of irregular data structures can significantly reduce the problem size and, therefore, can improve asymptotic performance. Many Parallel Random Access Machine (PRAM) algorithms for such data structures have been developed. But are these PRAM algorithms practical? Can we perform even the most primitive operations used by PRAM algorithms efficiently? We contend that there is hope. For example, scan (prefix sum) is such a primitive operation and is applied to arrays. For each element in the array it computes the "sum" of all the preceding elements in the array, where "sum" is a binary associative operator. Elsewhere, the efficient vector parallel implementation of the scan primitive has been shown to lead to greatly improved performance of several applications that cannot be vectorized with existing compilers [6, 26]. However, scan can only be applied to arrays that are linearly ordered in consecutive locations. If the data are stored unordered and the ordering is provided by links or pointers then we need to use other approaches for scan.

In this paper we consider a vector parallel implementation of list ranking and the scan operation applied to linked lists. List ranking and list scan are two fundamental primitives that are commonly used in solving problems on linked lists, trees, and graph data structures. Parallel algorithms frequently use list ranking for ordering the elements of a list, finding the Euler tour of a tree, load balancing [11], contention avoidance [15, 1], and parallel tree contraction [17], and these problems are subproblems of applications such as expression evaluation, graph 3-connectivity, and planar graph embedding [18]. In addition, list ranking is very interesting because it involves the kinds of problems for which it is hard to get good vector or parallel performance. In particular, it uses an irregular data structure, is highly communication bound, and its communication patterns are dynamic. From an algorithmic point of view it is interesting because it has features common to many problems: contention avoidance and load balancing. Furthermore, because the serial algorithm is so simple, it is very hard to get significantly faster parallel performance.

Algorithm	Time	Work	Constants	Scratch Space
Serial	$O(n)$	$O(n)$	small	c
Wyllie	$O(\frac{n \log n}{p})$	$O(n \log n)$	small	$n + c$
Ours	$O(\frac{n}{p} + \frac{n \ln m}{m})$	$O(n)$	small	$5m + c$
Random	$O(\frac{n}{p} + \log n)$	$O(n)$	large	$> 2n$
Optimal	$O(\frac{n}{p} + \log n)$	$O(n)$	huge	$> n$

Table 1 Comparison of several list ranking algorithms, where n is the length of the list, p is the number of processors, and m is a parameter of our algorithm ($m < n/\log n$, and for the CRAY C-90 $m \approx O((\log n)^3)$).

List ranking finds the position of each vertex in the list, by counting the number of links between each vertex and the head of the list. This position information can be used to reorder the vertices of the list into an array in one parallel step. Then, for example, scan can be applied to the array. Alternatively, scan can be applied directly to the linked list. We call this operation *list scan* and for each vertex in the linked list it computes the "sum" of the values of the all prior vertices in the list. List ranking and list scan are related in that list ranking is the list scan where plus is the operator and the values to be summed are all equal to one.

In the comprehensive review of PRAM list ranking algorithms by Halverson and Das [13] there is only one reference to an implementation of list ranking, which was Wyllie's algorithm on the CM-2. The only other parallel implementations of list ranking of which the author is aware use a randomized pointer jumping technique. Wyllie's algorithm [24] is work inefficient since it takes $O(n \log n)$ operations on a n element list, whereas a serial implemtation takes $O(n)$ operations. But, because it is very simple it works well for short lists or when we can increase the number of processors according to the linked list size. On the other hand, randomized pointer jumping techniques [17, 3] suffer from having to take multiple trials on average before being able to perform a pointer jump and, therefore, results in larger constants. Other work efficient parallel PRAM list ranking algorithms have very large constants, which has inhibited their implementation. Table 1 gives a comparison of list ranking algorithms, and Figure 1 compares the running times of five list ranking algorithms on one processor of the CRAY C-90. The Miller/Reif and Anderson/Miller algorithms use randomized pointer jumping, and the Belloch/Reid-Miller algorithm is the one on which we report here.

We introduce a new fully vectorized and parallelized algorithm that both is work efficient and has small constants. However, it does not achieve $O(\log n)$ running time. But we contend that work efficiency and small constants are more important, given that vector and multiprocessor machines are used for problems that are much larger than the number of processors and, therefore, the $O(\log n)$ time is never achieved in practice. For lists shorter than 2000 elements Wyllie's algorithm is faster than ours. But for long lists our implementation of list ranking and list scan on the CRAY C-90 is the fastest implementation to date, to the best of our knowledge. In addition, it is the first implementation of which we are aware that outperforms fast workstations. For example, it achieves over two orders of magnitude speedup over a DECstation 5000 workstation. On a single processor it also achieves a factor of five speed up over a serial

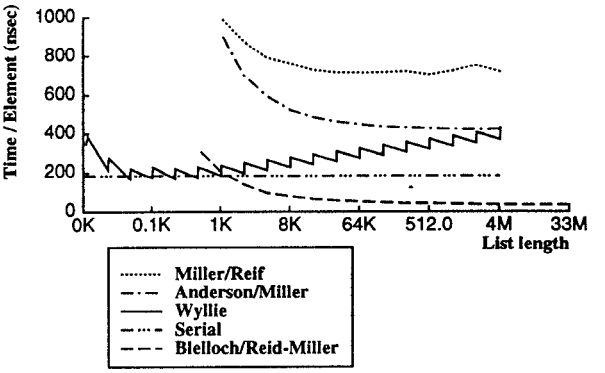

Figure 1: Execution times per element for several list ranking algorithms on one processor of the CRAY C-90. The times for Wyllie's algorithm and our algorithm were obtained on a dedicated machine. The saw-tooth shape of the Wyllie curve is due to the algorithm performing $\lceil \log n - 1 \rceil$ rounds of pointer jumping over all the data.

list scan on the CRAY C-90, which is significant since CRAY computers are also very fast scalar machines (see fallacy in Section 7.8 of [14]). In particular, when vectorizing a serial problem that requires gather/scatter operations, the best speedup one can expect on a single processor CRAY C-90 is about a factor of 12–18; if the vectorized algorithm does twice as much work as the serial code (both a reduction and contraction phase as our does) then the best you can expect is a 6–9 fold speedup on one processor. On 8 processors we obtain a 40 fold speedup over the serial code. In addition, our algorithm uses much less space than other algorithms, including Wyllie's.

1.1 Vector Multiprocessors as PRAMs

We chose to implement list ranking on a vector multiprocessor because these machines, such as the CRAY family of vector computers, closely approximate the abstract EREW PRAM machine (see Figure 2). These machines use a shared memory model, have fine-grain access to memory, have extremely high global communication bandwidth, and can hide functional and memory latencies through vectorization. The most important feature that distinguishes these machines from MMP machines is the pipelined memory access. Processors communicate to memory via a multistage butterfly-like interconnection network. As long as there are no memory bank conflicts, the network can service one memory request per clock cycle for each memory pipe. Thus, the PRAM model assumption that often is cited as unrealistic, namely unit-time memory access, holds on vector multiprocessors as long as we can avoid memory bank conflicts and hide latencies.

Zagha proposes several vector multiprocessing programming techniques for avoiding bank conflicts and hiding latencies [25]. To address bank conflicts he proposes a data distribution technique to manage explicitly the memory system. To address memory and functional units latencies, he proposes *virtual processing*, which is based on Valiant's Bulk Synchronous Processor (BSP) model [22]. This unifying model requires that algorithms are designed with sufficient *parallel slackness*, so that programs are written for rather more *virtual processors* than physical processors. For vector multiprocessors, this slack allows for vectorization so that computations and communication can be pipelined to hide

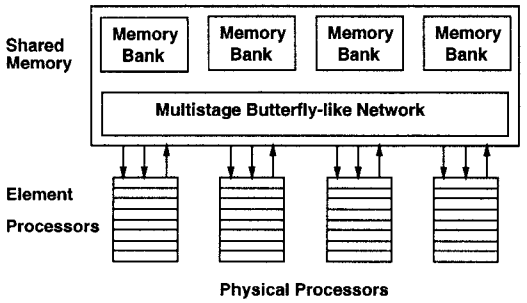

Figure 2: Vector multiprocessors as viewed as a PRAM

latencies.

In Zagha's programming model we implement PRAM algorithms by treating a vector processor as a SIMD (distributed memory) multiprocessor, where elements of the vector registers acts as element processors of the SIMD machine, see Figure 2. We call the processors *element processors* to distinguish them from a full vector processor. Because processors in data parallel algorithms do not use the results of another processor in the same time step, there are no recurrences to worry about in the corresponding vectorized implementation. By using strip-mining [19] or loop-raking [26, 5] we can assign the work of several virtual processors to a single element processor. Extending the vectorized algorithm to vector multiprocessors is straightforward if the machine is SIMD; simply treat the vector multiprocessor as a $l \times p$ SIMD multiprocessor, where l is the length of the vector-registers and p is the number of processors, and apply the vectorized algorithm. If the machine is MIMD, it can be treated the same way except that, for efficiency, the number synchronization points should be minimized.

The paper is organized as follows. In Section 2 we highlight the five list ranking algorithms we implemented. In Section 3 we describe the implementation of our algorithm in more detail and give timing data. In Section 4 we analyze the expected performance, describe how we tuned the parameters, and give our overall performance results. In Section 5 we describe the multiprocessor version of the algorithm and its performance. In the next section review some other PRAM list ranking algorithms. Finally, in Section 7 we discuss our conclusions and future directions.

2 The List Ranking and List Scan Algorithms

We chose to implement a few of the simplist list ranking algorithms in order to compare their performance with our parallel algorithm. In Section 6 we conclude that other work efficient algorithms have much larger constants than the ones we implemented and, therefore, are likely to worse performance. We describe the main features of the five algorithms we implemented below. Since list ranking is a list scan with all weights equal to one, we discuss list scan only. For simplicity we use integer addition as the "sum" operator.

2.1 The serial algorithm

The serial list scan simply walks down the list storing the accumulated values of the previous vertices until it reaches the end of the list. On the CRAY C-90 it takes 44 clock cycles or 180 nsec to traverse each element of the list (see Figure 1).

2.2 Wyllie's algorithm

The first parallel list ranking algorithm was introduced by Wyllie [24]. The algorithm uses a technique common to most parallel list ranking algorithms, "pointer jumping" or "short-cutting". A processor is associated with every vertex and each processor repeatedly replaces its successor pointer with its successor's successor pointer, in unison with the other processors. The new value for the vertex is its old value plus the value of its successor. For a list of length n and after at most $\log n$ iterations, every vertex points to the tail of the list and its value is the sum of the values from the vertex to the tail of the list. Although this algorithm is simple and has a $O(\log n)$ running time it is not work efficient since the total number of operations is $O(n \log n)$, whereas the serial algorithm takes $O(n)$ operations. When n is small, $\log n$ appears much like a constant.

Figure 1 shows the run times of Wyllie's algorithm on one processor of the CRAY C-90. The saw tooth shape of the curves is due to the addition of another round of pointer jumping whenever $\lceil \log n - 1 \rceil$ changes value. The negative slope between a pair of teeth is due to the amortization of the additive constant terms over larger size lists. As you can see from the figure, Wyllie's algorithm quickly degrades in performance as the list lengths grow. However, it does scale almost linearly with the number of processors and so is faster than the serial algorithm on multiple processors for moderately long lists.

2.3 Random mate

One of the simplest work efficient parallel algorithms was devised by Miller and Reif [17, 21]. It used randomization to break contention so that processors at neighboring vertices do not attempt to dereference their successor pointers simultaneously. Once a processor "splices out" a successor vertex, the processor for the successor vertex becomes idle. At each iteration only $\frac{1}{4}$ of the remaining vertices are spliced out on average. After $O(\log n)$ rounds all the vertices either point to the tail of the list or have been spliced out. Finally, there is a reconstruction phase, in which spliced out vertices are reintroduced in reverse order from which they were removed. We implemented this algorithm on a single processor of the CRAY C-90. Our version removes idle processors on every round by compressing the remaining vertices into contiguous vector elements (an operation we call "pack"). It also preallocates space required for recursive calls.

Anderson and Miller [3, 21] modified the above algorithm so that it avoids load balancing (packing). Processors are assigned the work of $\log n$ vertices. At each round a processor attempts to remove one vertex in its queue of vertices. In order to splice out its own vertex, a processor needs the reverse link so that it can get the previous vertex to jump over the processor's vertex. When a processor splices out its vertex, it attempts to splice out the next vertex in its queue on the next round. In this simple way processors remain busy without load balancing. After about $4 \log n$ rounds about $O(n/\log n)$ vertices are left, at which point they can be compressed in memory and Wyllie's algorithm can be applied. Finally, there is reconstruction phase to reintroduce spliced out vertices. Again only a constant proportion $(\frac{1}{4}$-$\frac{1}{2})$ of the processors remove vertices on each round. In our implementation we did not apply Wyllie's algorithm. We simply stopped processors from attempting to splice out vertices once they had completed their block of vertices.

Both implementations are an order of magnitude slower than our algorithm on one processor (see Figure 1), and should be similarly slower on multiple processors, since all the algorithms scale almost linearly on multiple processors. Although we did not spend much effort tuning these implementations, we doubt that we could get more than a factor of two improvement in their running time.

2.4 Our parallel algorithm

Many other work efficient and optimal PRAM algorithm have been developed for list ranking. Most use contract-rank-expand phases and address two considerations. One is how to find elements on which to work to keep all the processors busy and the second is how to avoid contention so that two processors are not working on neighboring list elements [2]. We deal with contention by randomly breaking up the linked list of length n into m sublists that can be processed independently and in parallel. We briefly describe the three phases of the algorithm below.

Phase 1 Randomly divide the list into m sublists. Traverse each sublist computing the the "sum" of the values at each vertex. Form a new linked list of length m connecting the sublists sums in the order the sublists appear in the original list.

Phase 2 Find the list scan of the reduced list found in Phase 1. These values are the scan values for the heads of the sublists.

Phase 3 Traverse each sublist computing list scan of each vertex as the sum of the vertex value and list scan of the previous vertex.

Phase 1 and 3 are vector parallel. The list scan in Phase 2 can be done recursively for large m, using Wyllie's pointer jumping technique [24] for moderate size m, or serially for small m. For small m serial list ranking works best because it avoids the overhead associated with multiprocessing and filling vector pipes (see Figure 1). Wyllie's algorithm performs best on moderate size lists where it can take advantage of vectorization and multiprocessing and where $\lceil \log n \rceil$ is small. For large m we use our algorithm recursively, until the number of sublists becomes small enough to use either the serial or Wyllie's algorithm. We determined empirically the size m should be when we switch between algorithms.

There are two problems with our algorithm that make it appear poor theoretically:

- The sublists lengths vary a great deal, from approximately $\frac{n}{m} \ln\left(\frac{m}{m-1}\right)$ to $\frac{n}{m}\ln(m)$ on average, where ln is log base e. Thus, the processors' work is imbalanced.
- Since the expected length of the longest sublist is approximately $\frac{n}{m}\ln(m)$ the parallel running time can be no better than that, ie, $O(\frac{n}{p}+\frac{n}{m}\log(m)), m < n/\log n$. In contrast, there are many parallel algorithms that have an $O(\frac{n}{p}+\log n)$ running time.

We ameliorate both problems by requiring m to be much greater than the number of processors, p. In this way a processor is responsible for several lists, namely m/p. Periodically we perform load balancing to regroup the lists, which addresses the first problem. When $p < m/\ln m$ the running time is dominated by n/p and the length of the longest sublist is not a problem.

The primary advantage to our algorithm is that it is both work efficient and has very small constants. The algorithm is fully vector parallel, and scales almost linearly with the number of processors. Figure 3 shows the speedup relative to one processor for various size lists.

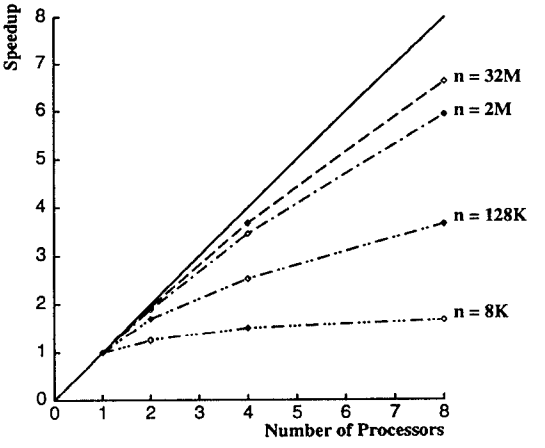

Figure 3: Relative speedups of our list ranking algorithm on the CRAY C-90.

3 Our Algorithm on a Vector Processor

This section describes our vector implementation of list scan (for a more detailed description see [20]). We used C and the standard Cray C compiler on a CRAY C-90. The runtime equations we give in this section are in CRAY C-90 clock cycles (4.2 nsec). Because many of our vector operations use indirect addressing we needed to give compiler directives in order to get the compiler to vectorize the loops; the only portion that is not vectorizable is the serial list scan in Phase 2. We attempted to reorder the statements within a loop in order to fill the multiple functional units for concurrent operations, to avoid contention between input/output memory ports and the gather/scatter hardware, and to avoid write after read dependencies [12]. Chaining is also possible within loops. Because memory access is dependent on the data, there is nothing we could do to avoid memory bank conflicts. However, since we are choosing random positions for the heads of the sublists, systematic memory bank conflicts are unlikely.

In the following description we assume that there is one virtual processor for every sublist. By using strip-mining, the element processors are assigned the work of an equal number of virtual processors. We represent the linked list as a pair of arrays. The value array contains the value of each vertex of the list and the link array contains the index of the next vertex in the list. The tail of the list is a self-loop, ie, the link at the tail is the index of the tail vertex.

Initialization: Each virtual processor picks a random position in the linked list to be the tail of a sublist. It saves the index of the successor link, which is the head of its sublist, sets the tail to a self loop and its list sum to *zero*, where *zero* is the identity of the list sum operator. Figure 4 shows the linked list that is the input of the list scan algorithm and the result of the initialization step.

It is possible that two virtual processors will pick the same random position at which to break the list. Then the

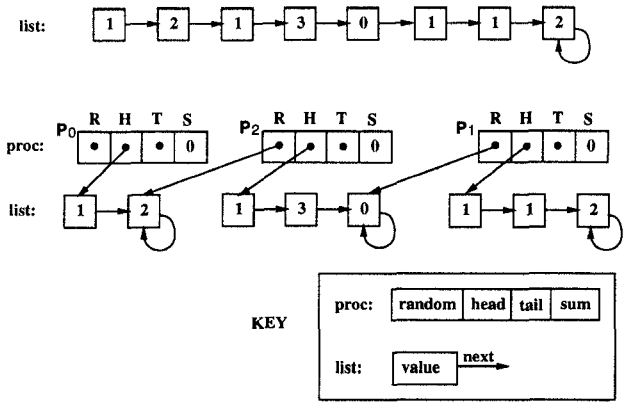

Figure 4: The top of the figure show the initial linked list with its values at each vertex. The bottom of the figure shows the results of initialization. The linked list is divided into 3 sublists, each terminating with a self loop. Each processor, P_0, P_1, P_2, saves two values: its chosen random position, R, and the successor of the random position in the original linked list, which becomes the head of its sublist, H. Each processor also initializes its sublist sum S to *zero*, the identity of the scan operator.

two processors will duplicate each other's actions and cause contention. We can either (1) use a parallel algorithm that guarantees to find no duplicate random numbers, such as in [16], or (2) we can remove duplicate random numbers by having a competition among the processors. Each processor writes its index at its random location and waits for all the processors to complete their writes. Then each processor reads the index at its random location. If the index is not its own it knows that it is a duplicate processor and can drop out of the computation. The first approach uses mod arithmetic, which is relatively slow on the CRAY and the second approach may require a pack, which to do efficiently is quite complicated, see [5]. We chose to use equally spaced positions and assume that linked lists are randomly ordered to simplify the implementation. The number of clock cycles needed for INITIALIZE to set up m sublists is:

$$T_{\text{Initialize}}(m) = 26m + 8980.$$

Phase 1: In Phase 1 alternates between summing the values along the links of the sublists and load balancing. Because of the predictable sizes of the sublists lengths we can determine how many links to traverse between load balancing steps and adjust this number as the procedure progresses (see Section 4). Figure 5 shows the status after the processors have found their sublist sums.

To illustrate just how streamlined the list traversal is we show the code written in pseudo C.

```
Initial_Rank (vp, ll, n_links) {
  for (j = 0; j < n_links; j++)
    for (i = 0; i < vp.n; i++) {          /* Vectorized */
      vp.sum[i] += ll.value[vp.next[i]];  /* Gather */
      vp.next[i] = ll.next[vp.next[i]];   /* Gather */
    }
}
```

The number of clock cycles needed for the inner loop is:

$$T_{\text{Initial\_Rank}}(x) = 3.5x + 60,$$

where x is the number of sublists. Notice that the loop only gathers the data necessary to compute the sum. There are no conditional tests or extra computation. We avoid conditional tests by destructively setting the sublist tail values to *zero* in the initialization step. In this way, we can repeatedly add the tail value without changing the sum. Because this loop (and the corresponding one in Phase 3) touches every link in the linked list the overall time of the algorithm lays primarily here and every economy is critical.

To load balance, the data for the completed sublists are saved and the incomplete sublists are packed into contiguous array locations. The number clock cyles needed to load balance x sublists is:

$$T_{\text{Initial\_Pack}}(x) = 8.5x + 1070.$$

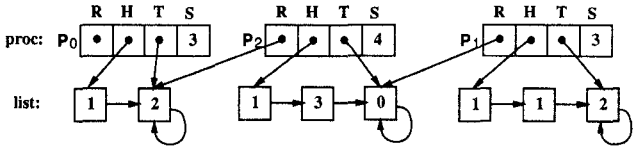

Figure 5: The figure shows the results of finding the sum of each sublist. Each processor has traversed its sublist until it has reached the sublist tail, T, and has accumulated the "sum" of the values along the sublist, S.

Next the processors create the reduced list of sublists sums. Each virtual processor has reached the tail of its sublist. It also has the tail of the previous sublist, which is the random position it chose during initialization. By writing the virtual processor's index into the tail of the previous sublist and then reading the index at the tail of its own sublist, the virtual processor determines the index of its successor's sublist. From this index the processor creates a link from its sublist sum to its successor sublist sum to form a new shorter linked list (see Figure 6). A processor can determine whether its sublist is the tail sublist because no processor wrote its index in the tail. The number of clock cycles needed to find the reduced list of length m is:

$$T_{\text{Find\_Sublist\_List}}(m) = 10m + 880.$$

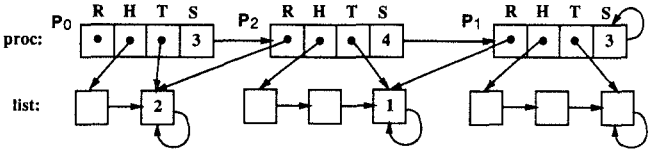

Figure 6: The figure shows finding the reduced list of sublist sums during Phase 1. Each processor writes its index at its random position in the linked list, R, and reads the index written at the tail of its sublist, T. This index is the index of the processor with the successor sublist. The tail sublist finds no index at the tail of its sublist.

For example, consider the tail of the first sublist in Figure 6. Processor 2 writes 2 in the tail of the first sublist. Then Processor 0 reads the 2 at the tail of its sublist. Thus, Processor 0 links its sublist to the sublist at Processor 2.

Phase 2: Depending on the size of this new linked list the algorithm finds the scan of the reduced linked list recursively, using Wyllie's algorithm, or serially. Figure 7 shows the result of this phase of the algorithm.

Figure 7: The list scan on the reduced list of sublist sums.

Phase 3: The scan value found in Phase 2 is the scan value for the head of a virtual processor's sublist, see Figure 8. As in Phase 1 each virtual processor traverses its sublist setting the scan of each vertex to the sum of the scan and value of the previous vertex. Again, load balancing is done periodically. The number of clock cycles needed to traverse x links, one from each sublist is:

$$T_{\text{Final\_Rank}}(x) = 4.7x + 60,$$

and to load balance x sublists is:

$$T_{\text{Final\_Pack}}(x) = 7.4x + 900.$$

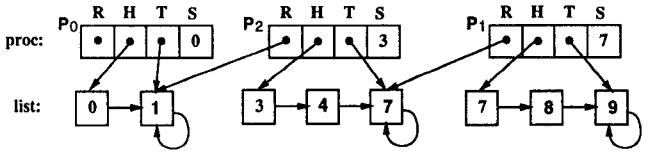

Figure 8: The scan values for the heads of the sublists are the scan values of the reduced list found in Phase 2. Phase 3 finds the remaining scan values.

Restoration: Finally each virtual processor reconnects the sublists to form the original linked list, using the values saved during initialization. The number of clock cycles required to reconnect m sublists is:

$$T_{\text{Restore\_List}}(m) = 4m + 400.$$

4 Analysis of the Algorithm

In Phase 1 and 3 of the algorithm we periodically perform load balancing so that completed sublists are removed from the computation. If we pack too frequently we remove none or only a few sublists, and when there are many sublists packing is expensive. If we do not pack often enough, we may have many processors performing needless work repeatedly chasing the sublists' tails. In order to determine when are good times to pack we first need a better understanding of what the expected distribution of the sublists lengths are. We find an estimate of the distribution in Section 4.1. Next, in Section 4.2 we determine the overall cost of performing the algorithm, given the timing data in Section 3. In Section 4.3 we used the expected distribution to minimize the execution time of the algorithm, given the number of sublists and number of times to pack. Finally, we explain how we compute the parameters values to mininimize the overall execution time. The main theorem is:

Theorem 1 *The list ranking algorithm in this paper has expected time $O(n/p + n \ln m/m)$ on p processors, when $m < n/\log n$.*

4.1 Analysis of sublist lengths

In this section we show that the distribution of the lengths of the sublists is approximately a negative exponential distribution, when n and m are large. We first consider the following situation. Let $X_1, \ldots, X_m$ be m random numbers in the range (0,1). For truly random numbers $\text{Prob}(X_i = X_j) = 0$ for $i \neq j$. Therefore, the numbers partition (0,1) into $m+1$ subintervals. Let $X_{(1)}, \ldots X_{(m)}$ denote the X's ordered by their sizes from smallest to largest.

Proposition 2 (Feller [10]) *If $X_1, \ldots, X_m$ are independent and uniformly distributed over the range (0,1) then as $m \to \infty$ the successive intervals in our partition behave as though they are mutually independent exponentially distributed variables with $E(X_{(i+1)} - X_{(i)}) = \frac{1}{m}$.*

In our case, we are choosing m random positions in a list of length n. If $n \gg m$, $n \to \infty$, and $m \to \infty$ then the lengths of the sublists tend to behave as mutually independent exponential variates with expectation $\frac{n}{m}$. That is, if L is a sublist length, then

$$\text{Prob}\{L > x\} \approx e^{-\frac{mx}{n}} = a.$$

We can estimated the expected length of the i^{th} shortest sublist by setting $a = (m - i + .5)/(m+1)$ and solving for x. This estimate seems to be reasonable for n and m as small as $n > 1000$ and $m > 100$. The expected length of the shortest sublist is:

$$\text{Exp}(L_{(0)}) \approx \frac{n}{m} \ln\left(\frac{m+1}{m+.5}\right),$$

and the longest sublist is:

$$\text{Exp}(L_{(m)}) \approx \frac{n}{m} \ln(2(m+1)).$$

Figure 9 shows the expected length of the i^{th} shortest sublist for several values of m when $n = 10000$ and compares it to some actual data averaged over 20 samples. Notice that as m increases the expected length of the longest sublist decreases and there is less variation in list lengths. Therefore, to reduce the parallel running time we want to make m large. However, as m increases the costs due to packs, initialization, and Phase 2 increases.

4.2 Cost of the algorithm

Using the timing equations of each piece of the algorithm, given in Section 3, we can determine the cost of the complete algorithm, assuming we know the exact lengths of the sublists and when packs are performed. Let S_i be the total number of links traversed in each list before the i^{th} pack. Let $g(x)$ be the expected number of sublists that have length greater than x. From the previous section:

$$g(x) = m \times Prob(\text{sublist length} > x) \quad (1)$$

$$\approx m e^{-\frac{mx}{n}} \quad (2)$$

The dotted line in Figure 10 shows $g(x)$ when $n = 10000$ and $m = 200$. The x-axis is the sublist length and the y-axis is the number of sublists with that length. You can think of each sublist as being laid out from left to right, and placed one above the other from longest to smallest, each starting at x = 0. That is, the y-axis is the number of sublists that are still active in the computation, namely the vector lengths

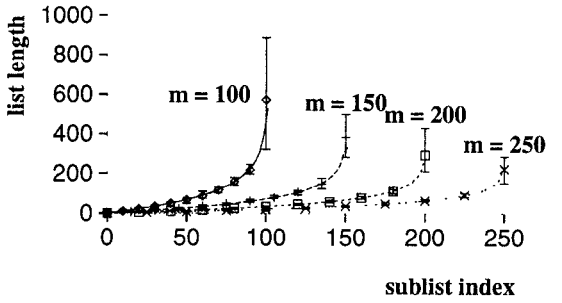

Figure 9: The function curves are the expected length of the i^{th} shortest sublist when $n = 10000$ for several values of m, the number of sublists. The observed lengths were found by taking 20 samples of dividing a list of size $n = 10000$ into m sublists and for the collection of the i^{th} shortest sublist of each sample, finding the average length (shown with a data symbol) and the minimum and maximum lengths (shown with an error bar).

of the computations, while the x-axis is the number of links traversed in each list. As we proceed from left to right, we are performing list ranking on a vector of length equal to the height of the step function. Every time we perform a pack operation (at the corner of a step) the vector length decreases. The area under the step function in Figure 10 is the expected total number of links traversed in either Phase 1 or Phase 3. If we packed every step then this number would be n, the area under the curve $g(x)$. Our aim is to minimize the area under the step function that is above the dotted line, while keeping the cost of packing down. The cost of packing is proportional to the sum of the heights of the step function.

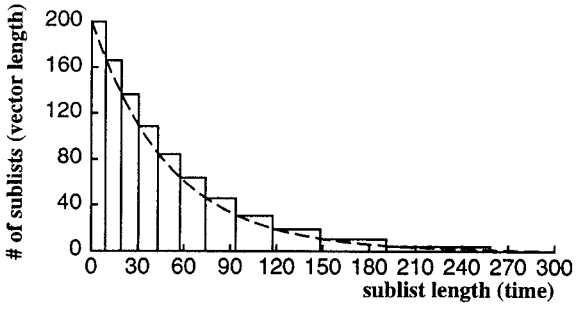

Figure 10: The dotted function is $g(x)$ the expected number of sublists that have length greater than x, where $n = 10000$ and $m = 200$. When the number of packing steps is 11, the expected execution time on the CRAY C-90 is minimized by packing at the vertical lines. The step function shows the expected number of sublists that are currently active at every iteration of list ranking. The size of a step is the expected number of sublists to complete since the previous pack.

From the equations for each part of the algorithm, given in Section 3, the expected number of CRAY C-90 cycles for Phase 1 and 3 of the algorithm is:

$$\begin{aligned} T_{P1+P3} &= \sum_{0}^{l-1}(S_{i+1} - S_i)(ag(S_i) + b) \\ &\quad + \sum_{0}^{l-1}(cg(S_i) + d) + em + f, \end{aligned} \tag{3}$$

where $T_{\text{Rank}}(x) = ax + b$, $T_{\text{Pack}}(x) = cx + d$, and $T_{\text{Other}}(x) = ex + f$, and $g(S_i)$ is the expected number of sublists remaining after traversing S_i links in each list. The first summation is the total time to traverse the links, the second summation is the time to pack out completed sublists, and the final $em + f$ is the time for creating the sublists, forming the reduced linked lists from the sublist sums, and restoring the linked list to its original form after the final phase.

4.3 Minimizing the time given fixed parameters

Consider n, m, and l fixed. We want to minimize the execution time with respect to $S_0, S_1, S_2, \ldots, S_l$, where S_i is the number of iterations of list ranking that have occured at the i^{th} pack. We can minimize Equation 3 by taking partial derivatives for each S_i and setting to zero to obtain a set of l simultaneous equations:

$$\begin{aligned} S_{i+1} &= S_i - \frac{g(S_{i-1}) - g(S_i)}{g'(S_i)} - \frac{c}{a} \\ &= S_i + \frac{g(S_{i-1}) - g(S_i)}{\frac{m}{n}g(S_i)} - \frac{c}{a}, \end{aligned} \tag{4}$$

That is, if we know the value of two consecutive packing points we can determine the following (or previous) packing point. Since we know $S_0 = m$, if we know S_1, we can compute $S_2, \ldots, S_l$ iteratively.

Notice in Figure 10 that the S_i's become increasingly further apart for larger i's reflecting the fact that the rate sublists complete slows down over time. The factor c/a in Equation 4 reflects the relative cost of packing and ranking. If we increase c and keep the number of packs constant, packing would occur less frequently during the initial iterations and occur more frequently during later iterations. Initially, packing is expensive because the vector lengths are long and becomes less expensive as vector lengths shorten. As we increase c relative to a eventually we find that the execution time is reduced by decreasing the number of packs even though the total number of ranking steps increases. In the next section we consider how to determine the best number of packs to perform.

We can simpify Equation 3 by using equation 4 to substitute for $S_{i+1} - S_i$. That is,

$$\begin{aligned} &\sum_{0}^{l-1}(S_{i+1} - S_i)(ag(S_i) + b) \\ &= aS_1 g(S_0) + a\sum_{1}^{l-1}(S_{i+1} - S_i)g(S_i) + b\sum_{0}^{l-1}(S_{i+1} - S_i) \\ &= amS_1 + a\sum_{1}^{l-1}[\frac{n}{m}(g(S_{i-1}) - g(S_i)) - \frac{c}{a}g(S_i)] + bS_l \\ &< amS_1 + an - c\sum_{1}^{l-1} g(S_i) + bS_l, \end{aligned}$$

where $S_l \approx \frac{n}{m}\ln m$. Thus,

$$T_{P1+P3}(S_1, \ldots, S_l) < an + b\frac{n}{m}\ln m + (aS_1 + c + e)m + ld + f.$$

For the CRAY C-90 the expected number of clock cycles on one processor for Phase 1 and 3 is:

$$T_{P1+P3} < 8.1n + 140\frac{n}{m}\ln m + (8.1S_1 + 60)m + 1970l + 10260. \quad (5)$$

where m is the number of sublists, S_1 is the number of links traversed before the first pack, l is the number of packs and is a function of S_1, m, and n.

4.4 Overall vector performance

From the previous discussion we have a way to determine at which iterations we should pack, if we know the length of the whole linked list, the number of sublists, and the iteration number of the first pack. However, all we know is the the length of the whole linked list, namely n. We now need to find good choices for the number of sublists m and the iteration number of the first pack S_1, which determines the number packs l. Our approach is to estimate the running time of the algorithm, using Equation 3, for various values of m, S_1 and n. Then, for each value of n we find values of m and S_1 that minimized the running time within about two percent. Finally, we fit functions to m *vs* n and S_1 *vs* n. It appears that m and S_1 are approximately cubic polynomials of $\log n$. We use these fitted polylog functions in our implementation to determine m and S_1 given n and Equation 4 to find successive values of S_i. When we use these estimates of m and S_1 we find that Equation 3 accurately predicts and Equation 5 over estimates the actual execution time on one CRAY C-90 vector processor for random inputs.

5 Vector Multiprocessor List Scan

To extend the algorithm to multiple vector processors we divide the virtual processors equally among the physical vector processors and let vectorization proceed on the virtual processor data assigned to the physical processors. The CRAY C compiler makes parallelizing relatively easy. Loops are modified to be tasked loops with compiler directives that direct the compiler to divide the loops into equal size chunks, one chunk per processor, and to vectorize the chunks within the processors.

Data parallel algorithms assume that each data parallel step is synchronized, whether or not it is necessary. However, the most we need to synchronize is after every innermost vectorized loop. If we treat the vector multiprocessor strictly as a $l \times p$ multiprocessor, we need to synchronize before each pack in Phase 1 and Phase 3 so that load balancing can proceed globally across the physical processors. Instead we assign virtual processors to physical processors once at the beginning and pack locally within each physical processor only. In this way each processor completes all of Phase 1 and Phase 3 independently of the other processors. In effect, we synchronize only a *constant* number of times and do no load balancing across processors. Eliminating synchronization avoids needless delays at each synchronization point. No global load balancing across processors is important because most compilers do not know how to parallelize a pack operation across processors and requires extra communication.

Because we use randomization, we do not expect a significant load imbalance when we only load balance locally. Even if an imbalance should become a problem as the procedure progresses, only one across-processor load balance should be necessary. Our results are excellent without any global load balancing.

Unfortunately, to tune the parameters m and S_1 we need to minimize for every possible number of processors. For a highly or massively parallel machines tuning the parameters for every number of processors would not be practical. But since the CRAY C-90 has 16 processors there are only 16 sets of equations. We tuned the parameters for 1, 2, 4, and 8 processors and result in the asymptotic performance of 8.4, 3.9, 2.0, 1.1[1] clock cycles per element, respectively. Figure 11 shows a graph of the execution times.

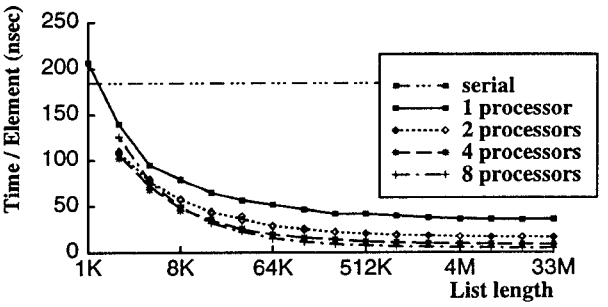

Figure 11: Execution times per element on 1, 2, 4, and 8 processors of the CRAY C-90 for our list ranking algorithm.

6 Other work efficient list ranking algorithms

On one CRAY C-90 vector processor our algorithm takes about 8.4 clock cycles per list element asymptotically to find the rank or scan of a linked list. If other algorithms are to be competitive, they must be able to use no more than 8.4 cycles per element on average. Below we discuss various other algorithms that have been described in the literature. Except for Wyllie's pointer jumping algorithm on short linked lists we conclude that other algorithms are unlikely to be competitive.

Cole and Viskin devised a parallel deterministic coin tossing technique [7] which they used to develop an optimal deterministic parallel list ranking algorithm [8, 4]. This algorithm breaks the linked list into sublists of two or three vertices long (the heads of the sublists are called 2-ruling sets); reduces the sublists to a single vertices; and then compacts these single vertices into contiguous memory to create a new linked list. It recursively applies the algorithm to the new linked list until the resulting linked list is less than $(n/\log n)$, at which point it applies Wyllie's algorithm. In the final phase it reconstructs the linked list by unraveling the recursion in the first phase to fill in the rank values of the removed vertices. The algorithm runs in $O(\log n \log\log n)$ parallel time and uses $O(n)$ steps. Later they modified their algorithm to give the first $O(\log n)$ time optimal deterministic algorithm [9, 23]. However, algorithms for finding 2-ruling sets that give either of these time bounds are quite complicated and have very large constants. They also give a much simpler 2-ruling set algorithm that is not work efficient but has smaller constants (see [4]). Because it is not

[1] The timing data in Section 3 and for a single processor were taken on a loaded machine Times for 2, 4, 8 processors were taken from a dedicated machine. We expect that a dedicated one processor would acheive about 7.5 clock cycles per element asymptotically

work efficient and its constants are larger than Wyllie's or ours, we chose not to implement it.

Anderson and Miller [2] combined their randomized algorithm with the Cole/Viskin deterministic coin tossing to get an optimal $O(\log n)$ time deterministic list ranking algorithm. As with their randomized algorithm, the processors are assign $\log n$ vertices which they process. At each round each processor executes a case statement that either breaks contention or splices out a vertex in its queue or splices out a vertex at another processor's queue. To break contention it finds a $\log \log n$ ruling set. Finding $\log \log n$ ruling sets are much simpler ($O(1)$ time) than finding 2 ruling sets ($O(\log n)$ time with large constants). But because each round involves a nonparallel three way case statement, where each case needs to be completed by all the processors before the next case can be executed, its constants are also much larger than ours.

7 Conclusions and Future Directions

In this paper we described a new parallel algorithm and its implementation for list ranking and list scan. List ranking and list scan are primitive operations on lists and the building block of many parallel algorithms using lists, graphs, and trees. Because of their expected poor performance on today's supercomputers, there are virtually no implementations of algorithms using these data structures. Our contribution is that we have implemented the most basic of these "untouchable" algorithms on the CRAY C-90 with success.

One of the primary problems with any list ranking algorithm is that the access pattern to the list is very irregular and unpredictable. But fortunately, the CRAY class of vector computers have a very fast memory access network that makes implementing a list ranking algorithm reasonable. It is CRAY's pipelined memory access and extremely high global bandwidth that makes our implementation so fast.

Although the parallel running time of our algorithm is $O(n/p + n \ln m/m)$, where m is small relative to n, it is work efficient. And because it is work efficient and has small constants it is the fastest implementation list ranking and list scan to date. Most parallel list ranking algorithms attempt to find a large number, at least $O(n/\log n)$ and as many as $n/2$, of nonadjacent elements in the list, which can be quite costly since it applies to the whole list. Our algorithm only tries to find a relatively small number, m, of such elements, and because the sublists are quite long, we can process the lists at full speed for their entire length. However, the amount work assigned to each processor can be quite different. But by a unique analysis of the expected work loads we are able to determine when to perform load balancing to minimize the overall running time of the algorithm.

As with any implementation there are a multitude of possible modification and enhancements that could improve its performance. A large part of the performance loss is due to short vector lengths. As lists drop out of the computation the vector lengths shorten. Not only are the vector lengths short, the number of iterations remaining with short vector lengths can be relatively large, since the longest sublists can be much longer that the other sublists. Short vectors are inefficient because with each iteration there is a latency due to filling the vector pipes. The issue is whether there is a way to keep the vector lengths long without unduly slowing down the key list ranking portion of the algorithm.

Finally, the question still remains whether having a fast list ranking implementation is useful as a primitive for other major applications. If so, we may have opened up major classes of PRAM algorithms that can have reasonable implementations. It also would be interesting to see whether our approach of subdividing a problem randomly into a moderate number of fairly coarse gain subproblems and applying load balancing periodically can be applied to other computational problems.

8 Acknowledgements

We thank the Pittsburgh Supercomputing Center for use of their CRAY C-90, Guy Blelloch for numerous insightful conversations, Marco Zagha who helped us understand the Cray Assembly Language code produced by the Cray C compiler and made many useful comments on an earlier draft of this paper, Alan Frieze who noted that the sublists lengths approximately followed an exponential distribution, and Gary Miller for several helpful conversations.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] R. Anderson and G. L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC88*, volume 319 of *Lecture Notes in Computer Science*, pages 81–90. Springer-Verlag, June/July 1988.

[3] R. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[4] S. Baase. Introduction to parallel connectivity, list ranking, and Euler tour techniques. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 61–114. Morgan Kaufmann, 1993.

[5] G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, pages 416–424, Mar. 1992.

[6] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

[7] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[8] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. In *Information and Control*, volume 70, pages 31–53, 1986.

[9] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, June 1989.

[10] W. Feller. *An Intoduction to Probability Theory and Its Applications.* Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, New York, 1971.

[11] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S. K. Tewsburg, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations*, pages 139–156. Plenum Publishing Corporation, 1988.

[12] A. R. Hainline, S. R. Thompson, and L. L. Halcomb. Vector performance estimation for CRAY X-MP/Y-MP supercomputers. *Journal of Supercomputing*, 6:49–70, 1992.

[13] R. Halverson and S. K. Das. A comprehensive survey of parallel linked list ranking algorithms. Technical Report CRPDC-93-12, University of North Texas, Aug. 1993.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann, San Mateo CA, 1990.

[15] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computation by ranking (extended abstract). In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC88*, volume 319 of *Lecture Notes in Computer Science*, pages 101–110. Springer-Verlag, June/July 1988.

[16] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings ACM Symposium on Theory of Computing*, pages 1–10, May 1985.

[17] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, Oct. 1985.

[18] G. L. Miller and J. H. Reif. Parallel tree contraction. part 2: Further applications. *SIAM Journal of Computing*, 20(6):1128–1147, Dec. 1991.

[19] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[20] M. Reid-Miller and G. E. Blelloch. List ranking and list scan on the Cray C-90. Technical Report CMU-CS-94-101, School of Computer Science, Carnegie Mellon University, Feb. 1994.

[21] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufmann, 1993.

[22] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] U. Vishkin. Advanced parallel prefix-sums, list ranking and connectivity. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 215–257. Morgan Kaufmann, 1993.

[24] J. C. Wyllie. The complexity of parallel computations. Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1979.

[25] M. Zagha. *Efficient Irregular Computations on Vector Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University. In Preparation.

[26] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, Nov. 1991.