List Ranking and List Scan on the CRAY C-90

Margaret Reid-Miller Guy E. Blelloch

March 15, 1994

CMU-CS-94-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and in part by the Pittsburgh Supercomputing Center (Grant ASC890018P) who provided Cray Y-MP and Cray Y-MP C90 time. Guy Blelloch was partially supported by a Finmeccanica chair and an NSF Young Investigator award (Grant CCR-9258525).

Keywords: list ranking, list scan, prescan, prefix-sum, bulk synchronous processor (BSP) model, parallel algorithm, vector algorithm, linked list, load balancing, randomized algorithm, vector multiprocessor, CRAY Y-MP C-90

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.

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, 25]. However, this version of 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.

List ranking finds the position of each node in the list, by counting the number of links between each node and the head of the list. This position information can be used to reorder the nodes 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 node in the linked list it computes the "sum" of the values of the all prior nodes 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 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 random pointer jumping technique. Wyllie's algorithm [23] 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, the random pointer jumping technique [17, 3] suffers 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 random 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

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 Mate	$O(\frac{n}{p} + \log n)$	$O(n)$	large	$> 2n$
Optimal	$O(\frac{n}{p} + \log n)$	$O(n)$	very large	$> n$

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 $m \approx O((\log n)^3)$).

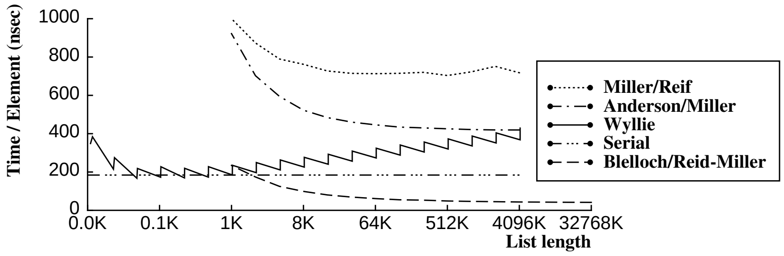

Figure 1: Execution times per element for several list ranking algorithms on one processor of the CRAY C-90. The times for Wyllie's algorithm and our algorithm were obtained on a dedicated machine. The saw-tooth shape of the Wyllie curve is due to the algorithm performing $\lceil \log n - 1 \rceil$ rounds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

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 7000 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 four speed up over a serial 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. We obtain an additional 6.7 speedup on 8 processors. 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.

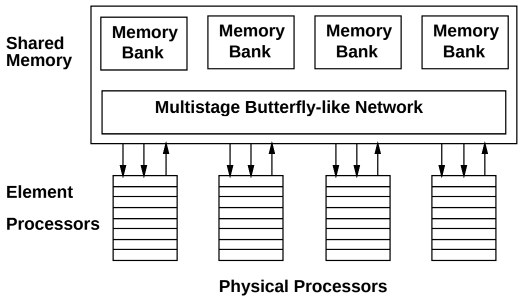

Figure 2: Vector multiprocessors as viewed as a PRAM

Zagha proposes several vector multiprocessing programming techniques for avoiding bank conflicts and hiding latencies [24]. 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 [21]. 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 latencies.

In Zagha's programming model we implement PRAM algorithms by treating a vector processor as a SIMD (distributed memory) multiprocessor, where each element in the vector acts as a processor of the SIMD machine, see Figure 2. 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. 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 discuss the five list ranking algorithms we implemented. Section 3 describes our implementation on the CRAY C-90 and gives timing equations for each part of the implementation. 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 and review other PRAM list ranking algorithms. Finally, in Section 7 we discuss our conclusions and future directions.

2 The List Ranking and List Scan Algorithms

List ranking computes the distance each node is from the head of the linked list. List scan computes the "sum" of the values on the links in a linked list from the head of the linked list to each node in the linked list, where "sum" is any binary associative operator. 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. We represent the linked list as a pair of arrays. The value array contains the value of each node of the list and the link array contains the index of the next node in the list. The tail of the list is a self-loop, ie, the link at the tail is the index of the tail node.

2.1 The serial algorithm

The serial list scan simply walks down the list saving the accumulated values of the previous nodes 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, and can be coded as follows. Let the array *l_next* represent the linked list where each element contains the index of the next node in the list. The tail of the list is indicated by a self-loop, ie, if *tail* is the index of the last element in the list then $l\_next[tail] = tail$.

```
Serial_List_Scan(l_sum, l_next, l_value, head)
{
   / * l_sum – list scan results
     * l_next – linked list terminated with a self loop
     * l_value – values of the nodes
     * head – head of linked list
     */
  sum = ZERO;
  next = head;
  do {
    l_sum[next] = sum;
    sum += l_value[next];
    next = l_next[next];
  } while (next ≠ l_next[next]);
}
```

2.2 Wyllie's algorithm

The first parallel algorithm for list ranking is due to Wyllie [23]. The algorithm uses a technique common to all parallel list ranking algorithms, "pointer jumping" or "shortcutting". A processor is associated with each node of the list. In parallel, each processor modifies its next pointer *l_next* to point to its successor's successor. For each round of pointer jumping the number of list elements that *l_next* jumps over can double from the previous iteration. The actual number of elements it jumps over is retained in *l_value* array. This array is computed by adding an element's value with its successor's value during each pointer jump. After $\lceil \log_2 n \rceil$ rounds of pointer jumping all elements point to the tail of the list and idl_value contains the distance each node is from the tail of the list. The data-parallel version of the inner loop of Wyllie's algorithm is as follows:

```
Wyllie_Loop(l_next, l_value, n)
{
   / * l_next – next link
     * l_value – sum of values of list between self and next
     */
  for (i = 0; i < n; i++) {
    next = l_next[i];
    l_value[i] += l_value[next];
    l_next[i] = l_next[next];
  }
}
```

For each statement we must ensure that all the data are read before they are written back to the same array. We accomplish this in our vector multiprocessor version by writing to a different array from which we are reading. Then, on each call to the inner loop we switch back and forth between arrays we read from and arrays we write to.

The simplicity of this algorithm makes it quite attractive. However, there are two main problems with Wyllie's algorithm:

- On each iteration the number of nodes of the list that concurrently read the values at the tail doubles. At the final iteration anywhere from half the nodes to all but one of the nodes may concurrently read the values at the tail. On CRAY X-MP/CRAY Y-MP computers concurrent reads are serialized.
- The algorithm is not work efficient and does $\log n$ times as much work as the serial algorithm.

Figure 3 shows the run times of Wyllie's algorithm on 1 to 8 processors 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. Wyllie's algorithm quickly degrades in performance as the list lengths grow. However, it does scale linearly with the number of processors.

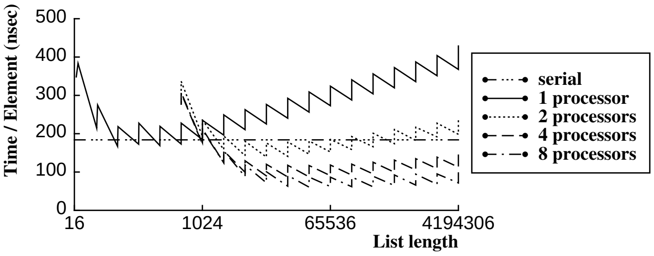

Figure 3: The running time per element to perform Wyllie's list ranking algorithm on 1, 2, 4, and 8 processors on the CRAY C-90. Whenever $\lceil \log n - 1 \rceil$ increases by one there is a corresponding jump in the per element running time of the algorithm, where n is the list length. The implementation on one processor has no overhead due to multitasking and, hence, performs better on small lists than the multiprocessor version.

2.3 Random mate

One of the simplest work efficient parallel algorithms was devised by Miller and Reif [17, 20]. It uses randomization to break contention so that processors at neighboring nodes do not attempt to dereference their successor pointers simultaneously. Once a processor "splices out" a successor node, the processor for the successor node becomes idle. At each iteration only $\frac{1}{4}$ of the remaining nodes are spliced out average. After $O(\log n)$ rounds all the nodes either point to the tail of the list or have been spliced out. Finally, there is a reconstruction phase, in which spliced out nodes 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 by packing the vectors on every round in order to make the implementation work efficient.

Anderson and Miller [3, 20] modified the above algorithm so that it avoids load balancing (packing). Processors are assigned the work of $\log n$ nodes. At each round a processor attempts to remove one node in its queue of nodes (not the successor node as in the previous algorithm). However, in order to splice out its own node, the processor needs reverse link pointers so that it can get the previous node to jump over the processor's node. If a processor is able to splice out its node in one round, in the next round it attempts to splice out the next node in its queue. In this simple way processors remain busy without load balancing being required. After about $4 \log n$ rounds $O(n/\log n)$ nodes are left, at which point they can be compressed in memory and processed using Wyllie's algorithm. Finally, there is reconstruction phase to reintroduce spliced out nodes. Again only a small constant proportion ($\geq 1/4$) of the processors remove nodes on each round. In our implementation of this algorithm we did not apply Wyllie's algorithm. We simply stopped processors from attempting to splice out nodes once they had completed their block of nodes.

Both implementations of the random mate approach are an order of magnitude slower than our algorithm on one processor, and should be similarly slower on multiple processors, since all the algorithms scale almost linearly on multiple processors (see Figure 1). The algorithms are also slower than the serial implementation on one processor. 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. The list ranking proceeds in three phases:

Phase 1 Randomly divide the list into m sublists. Reduce each sublist to a single node with value equal to the "sum" of the values in the sublist. Now the list is of length m.

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

Phase 3 Expand the nodes in the reduced list back into the original linked list filling in the scan values along the list.

Phase 1 and 3 can be done in parallel. The list scan in Phase 2 can be done recursively for large m, using Wyllie's pointer jumping technique [23] 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:

1. The sublists lengths vary, 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.
2. 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, $O(\frac{n}{p} + \frac{n}{m} \ln(m)), m < n/\log n$. In contrast, there are many parallel algorithms that have a $O(\frac{n}{p} + \log n)$ running time.

We ameliorate both problems by requiring m to be much greater than the number of processors, p. Each processor is responsible for several lists, namely m/p. Periodically we perform load balancing to regroup the lists, which addresses the first problem. If $p < m/\ln m$ then 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 4 shows the speedup relative to one processor for various size lists.

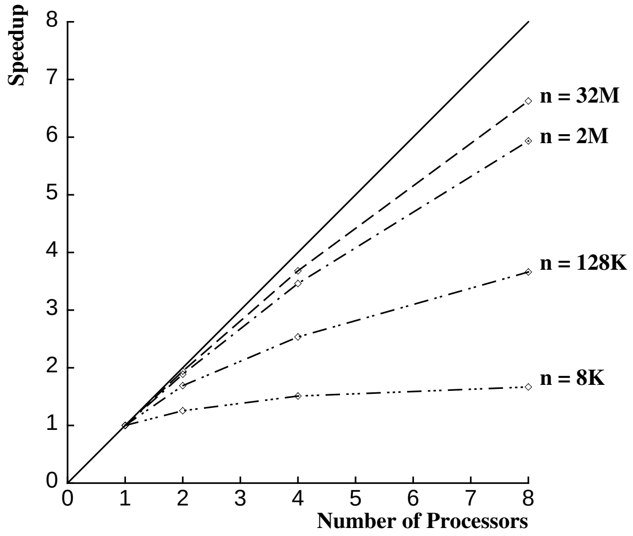

Figure 4: Relative speedups of our list ranking algorithm on the CRAY C-90.

In the following description we assume that there is one virtual processor for every sublist. Physical processors are assigned to do the work of an equal number of virtual processors. Because the algorithm is data parallel the physical processor performs one step on each virtual processor before proceeding to the next step. The algorithm proceeds in three stages.

Input: The head of a linked list, the linked list, and its associated list values. It assumes that the linked list is in contiguous memory and terminates with a self loop.

Output: The list scan of the linked list.

Initialization: Each processor picks a random position in the linked list to be the tail of a sublist. We do not let a processor choose the tail of the whole list as its random position because it is convenient not to worry about a zero length list in Phase 2. It saves the position of the tail and the successor link, and sets the tail to a self loop. It then prepares to find the list scan of the following sublist (which another processor created). It initializes its sublist head as the successor link saved above and the list sum to *zero*, where *zero* is the identity of the list sum operator. Figure 5 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 processors will pick the same random position at which to break the list. Then the 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].

Phase 1: Each virtual processor traverses its sublist adding the values along the links to the sum. When a virtual processor reaches the tail of its sublist it "drops out" of the computation. After traversing s_i links a load balancing step is done, reassigning virtual processors that have not completed their sublist to the

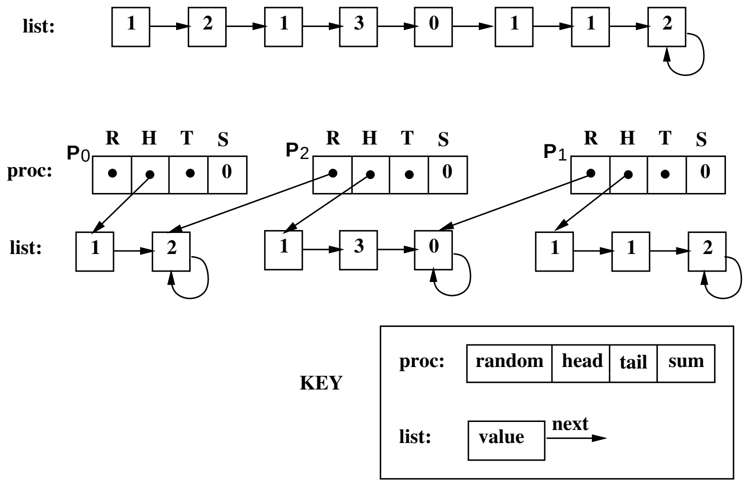

Figure 5: The top of the figure shows the initial link list with its values at each node. The bottom of the figure shows the results of initialization. The linked list is divided into 3 sublists, each terminating with a self loop. Each processor, P_0, P_1, P_2, saves two values: its chosen random position, 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.

physical processors. Each time the physical processors load balance they increment i to compute a new s_i. Because of the fairly predictable sizes of the sublists we can determine reasonable values of s_i (see Section 4). Figure 6 shows the status after every processor has dropped out and has found the sum of its sublist. Next the virtual processors create the reduced list of sublists sums.

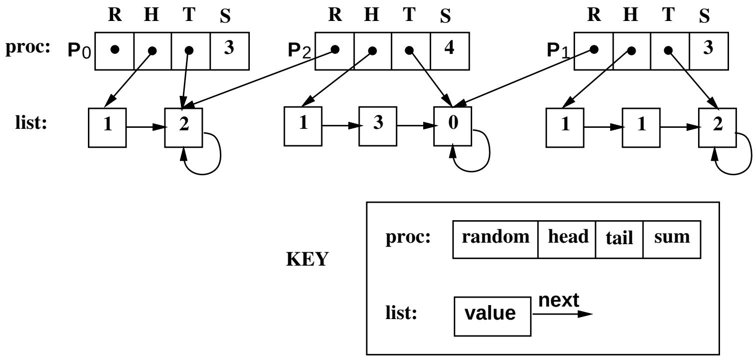

Figure 6: The figure show the results of finding the sum of each sublist. Each processor has traversed its sublist until it has reached the sublist tail, T, and has accumulated the "sum" of the values along the sublist, S.

At this point each virtual processor has reached the tail of its sublist. It also has the tail of the previous sublist, which is its random position found during initialization. By writing the processor's index into the tail of the previous sublist and then reading the index at the tail of its own sublist, the 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 7.

For example, consider the tail of the first sublist in Figure 7. The random position for Processor 2 is the tail of the first sublist. Processor 2 writes 2 at that tail, ie the tail of the sublist previous to Processor 2's

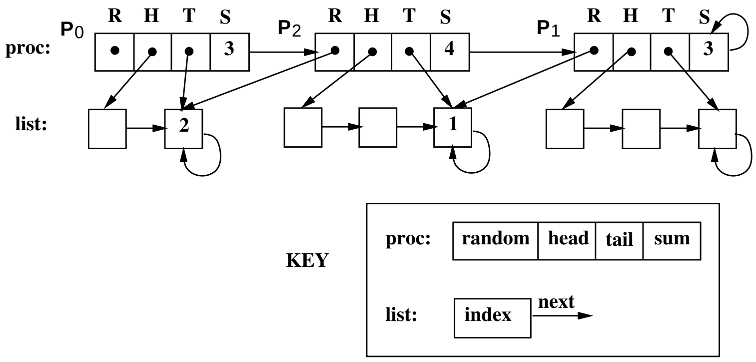

Figure 7: The figure shows finding the reduced list of sublist sums during Phase 1. Each processor writes its index at its random position in the linked list, R, and reads the index written at the tail of its sublist, T. This index is the index of the processor with the successor sublist. The tail sublist finds no index at the tail of its sublist.

sublist. Then Processor 0 reads the index at the tail of its own sublist, namely the first sublist. The value is 2, the processor index of its successor sublist. Thus, Processor 0 links its sublist to the sublist at Processor 2.

The tail of the new linked list corresponds to the tail sublist. A processor can determine whether its sublist is the tail sublist because no processor wrote its index in the tail. The processor for the tail sublist can now set the tail of the new reduced linked list to a self loop. The values of each node of the reduced sublist is the sublist sums found in Phase 1.

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 8 shows the result of this phase of the algorithm.

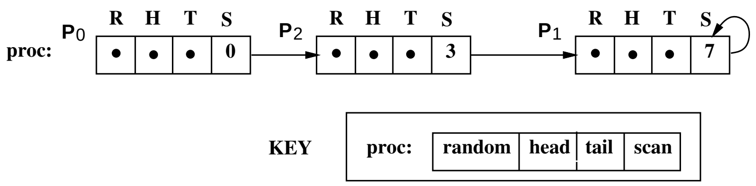

Figure 8: The list scan on the reduced list of sublist sums.

Phase 3: Phase 3 starts with the scan value found in Phase 2 as the scan value for the head of its sublist (see Figure 9). Each virtual processor finds the scan of the remaining nodes in its sublist in the same manner as in Phase 1. It traverses its sublist setting the scan of each node to the sum of the scan and value of the previous node. Again, the physical processors load balance after each virtual processor has traversed $s_i, i = 1, 2, 3, \ldots, l$ links of their sublist, where i is the number of times load balancing has been performed.

Restoration: Finally each virtual processor reconnects the sublists to form the original linked list, using the values saved during initialization. That is, each processor except Processor 0 replaces the self loop at its random position with link to its sublist head. Figure 10 shows the final scan values and the restored linked list that is the result of the completed algorithm.

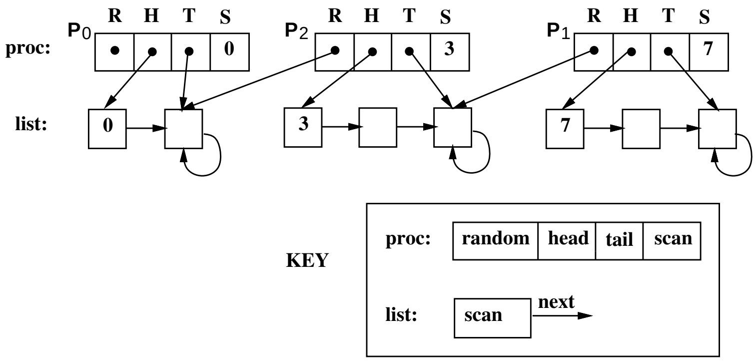

Figure 9: The scan of the reduced list found in Phase 2 are the scan values for the heads of the sublists.

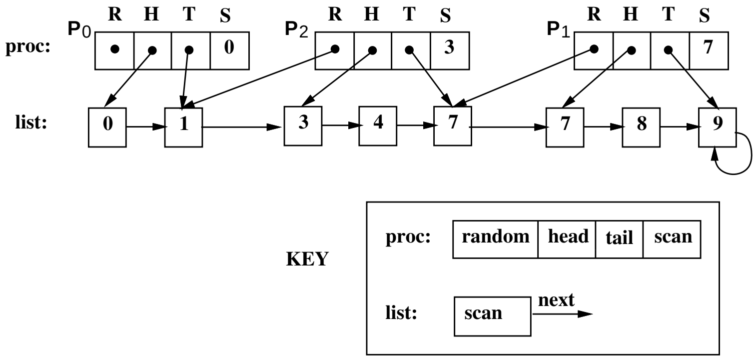

Figure 10: The resulting scan values of the linked list found in Phase 3. The links at the tails of the sublists, R, are replaced by links to the sublist heads, H, to restore the linked list to its original form.

3 Vector Implementation of List Scan

We implemented our list scan algorithm on a CRAY C-90, a vector multiprocessor. Vector multiprocessor machines consist of multiple scalar processors, each augmented with a bank of vector registers, pipelined functional units, and vector instructions. The functional units divide their operation into several stages so that the clock speeds can be increased. On every clock cycle another element from the vector register enters the pipeline of the functional unit while one results exits the pipeline. The delay between the time the first operands enters the pipeline until the first result leaves the pipeline is call the *latency* or *start up* time of the functional unit. If the functional units are fully pipelined, they can accept new operands every clock cycle. Multiple functional units can process data simultaneously, and if the hardware permits, the results of one functional unit can be *chained* directly to the input of another unit. Thus, to execute operands of length n on a fully pipelined functional unit with start up time s takes $s + n$ clock cycles, where $n \leq v$, the vector register length. To hide the latency s we want n as close to v as possible.

The vector multiprocessors are typically connected through a multistage interconnection network to a common memory. Memory is composed of multiple memory banks that can access different addresses in parallel using a single global address space. Once a memory bank has been accessed it cannot be access again until there is a delay, called the *cycle time*. Usually the memory is optimized for sequential access. That is, banks are fully interleaved so that successive addresses are on successive memory banks and the number of memory banks is greater than the cycle time. In this way, memory can be accessed sequentially one element per clock cycle. Vectors can be loaded and stored sequential (stride = 1) or at every k^{th} element (stride = k) of memory. Bad choices for k can result in the same memory bank being accessed at a rate higher than the cycle time and a *memory-bank conflict* occurs, causing memory stalls. Memory can also be accessed at arbitrary locations using an index vector. Often such loads are called *gather* operations and stores are called *scatter* operations. Because of memory bank conflicts, each element during a gather or scatter typically is accessed at a rate lower than the machine clock cycle (about 2 clock cycles/element for random access patterns on the CRAY Y-MP machines). In addition to cycle time delays there are *access latency* time delays. The latency for a load is the time to get the first word from memory to the register and often is much greater than the latency of the functional units. However, for CRAY Y-MP machines memory access latencies are about the same, but are getting longer as the machines get bigger.

When we implement a PRAM algorithm on vector multiprocessor, we treat each element in a vector register as an *element processor* of a SIMD machine. We call the vector element an element processor to distinguish it from a full vector processor. Any data parallel algorithm can be vectorized and parallelized by having an element processor do the work of a virtual processor in the algorithm. For example, on a CRAY C-90 each processor has a vector-register length of 128 and there are 16 processors. Therefore, we can have as many as 2048 element processors. However, by using strip-mining [19] or loop-raking [25, 5] we can assign the work of several virtual processors to a single element processor.

We implemented our list scan algorithm using C and the standard Cray C compiler on a CRAY C-90. 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. All loops we present in this section can be vectorized. In the actual implementation, 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, except possibly randomizing the input. However, since we are choosing random positions for the heads of the sublists, systematic memory bank conflicts are unlikely.

In this section we give pseudo C code to illustrate our implementation of the single vector processor version. In section 5 we show how we modified this algorithm for a vector multiprocessor machine. Below we use three structures containing sets of vectors to simplify the presentation. However, in our actual implementation we use individual vectors. For each subroutine we discuss the vectorization, develop an equation for estimating the execution time, and present the C pseudo code.

List_Scan We treat the linked list *ll* as a pair of vectors where one vector *ll.next* gives the indices of the successive nodes in the linked list and the other vector *ll.value* gives the values of the nodes. The scalar *ll.head* is the index of the head of the linked list and the scalar *ll.n* is the length of the linked list. We assume that the linked list terminates with a self loop. The resulting list scan will be store in the vector *ll.sum*.

We use another set of vectors *vp* that represent the virtual processors, which we periodically pack as processors drop out. The vectors *vp.next* gives the index of the next successor in each sublist, *vp.sum* the current "sum" of each sublist, and *vp.proc_id* the virtual processor id. The scalar *vp.n* is the number of currently active virtual processors.

In order to avoid having to check whether a processor has reached the end of a sublist at every pointer dereference we modify the parallel algorithm described in the previous section. During initialization at the tail of each sublist we destructively set $ll.next$ to its own index to create a self loop and set $ll.value$ to *zero*, where *zero* is the identity value of the scan operator. In this way, we can repeatedly add the tail value to the sublist sum without affecting the sum.

Finally, we use a set of vectors *sl* to save information about the sublists. So that we can restore ll before returning from LIST_SCAN we save the random indices of the tails of sublists in *sl.random*, the values of the tails in *sl.value*, and the successor links at the tails, namely the heads of the sublists in *sl.head*. During the course of the algorithm we save intermediate results, *sl.tail*, the index of the tail of each sublist, *sl.sum*, the sum of each sublist, *sl.next*, the index of the the successor sublist.

LIST_SCAN starts by calling INITIALIZE which sets up the sublists and returns the number of sublists created. In Phase 1 LIST_SCAN alternates between traversing each sublist and packing out completed sublists until no sublists remain. INITIAL_RANK traverses $s[l]$ links of each sublist summing the values at each node (in Section 4.3 we discuss how we determine the values of $s[l]$). Then INITIAL_PACK load balances the remaining lists by removing the completed sublists from *vp*. It saves the results of the completed sublists in *vp* and removes them from *sl* by packing the remaining sublists to the initial portion of the arrays. By packing the arrays we effectively reassign virtual processors to element processors. Finally, INITIAL_PACK returns the number of incomplete sublists remaining. After all the sublists have completed, FIND_SUBLIST_LIST forms a linked list, *sl.next*, of the sublist sums, *sl.sum*. In Phase 2 it finds the list scan of this linked list by calling either LIST_SCAN recursively, the vector multiprocessor routine WYLLIE, or the serial routine SERIAL_LIST_SCAN. The results are put in the virtual processor array *vp.sum*. Phase 3 is like Phase 1 and proceeds by alternating between traversing the sublists for $s[l]$ nodes and packing out finished sublists until no sublists remain. Finally, RESTORE_LIST puts back the original links and values at the tails of the sublists. All the routines are vectorized except SERIAL_LIST_SCAN.

```
List_Scan(ll.sum, ll)
{
  /* Phase_1 */
  l = 0;                                   /* Initialize the pack counter */
  vp.n = sl.n = INITIALIZE(vp, sl, ll);    /* Initialize the virtual processors */
  while (vp.n > 0) {                       /* Find the sublist sums */
    INITIAL_RANK(vp, ll, s[l++);
    vp.n = INITIAL_PACK(vp, sl, ll);
  }
  FIND_SUBLIST_LIST(ll, sl);               /* Turn the sums into a list */

  /* Phase_2 */
  if (sl.n > wyllie_cutoff)
    LIST_SCAN(vp.sum, sl);                 /* Recursive */
  elseif (sl.n > serial_cutoff)
    WYLLIE(vp.sum, sl);
  else
    SERIAL_LIST_SCAN(vp.sum, sl.next, sl.sum, 0);

  /* Phase_3 */
  l = 0;                                   /* Reset the pack counter */
  vp.n = sl.n;                             /* Reset number of virtual processors */
  while (vp.n > 0) {                       /* Find the scan of the sublists */
    FINAL_RANK(vp, ll, s[l++]);
    vp.n = FINAL_PACK(vp, sl, ll);
  }
  RESTORE_LIST(sl, ll)                     /* Restore values at sublist tails */
}
```

Initialize: Initialization starts by finding *sl.n*, the appropriate number of sublists to use given the length of the linked list. In Section 4.3 we discuss how to determine what is an appropriate value for *sl.n*. GEN_TAILS finds *sl.n* pseudo-random positions in the linked list, *sl.random*, which are to be the tails of the sublists. It also ensures that none of these positions are the tail of the whole list. To simplify the implementation we chose to use equally spaced positions and assumed that the linked lists are randomly ordered. If the ordering of the links is random then we can expect sublist lengths to follow the same distribution as when the heads of the lists are chosen randomly. Next INITIALIZE saves the links and values at the tails. The head of the first sublist is the head of the whole linked list. Because the links are retrieved from random positions to retrieve *ll.next* at *sl.random* requires a load and a gather, and to save *sl.head* requires a store. Then INITIALIZE gathers *ll.value* at *sl.random* and stores them in *sl.value*.

Next INITIALIZE turns the linked list into a set of sublists by setting the tails to self loops and tail values to *zero*. As the tails are at random positions these assignments require two scatter operations. We need not worry about the value of the tail of the whole list in Phase 1 because we do not need the correct sum for the tail sublist when finding the scan in Phase 2. Finally INITIALIZE initializes the virtual processor vectors: it stores the heads, stores *zero*'s at the sums, and stores the processor id's.

```
Initialize(vp, sl, ll)
{
  sl.n = COMPUTE_NUM_SUBLISTS(ll.n);
  GEN_TAILS(sl.random, sl.n, ll.n);          /* Find random positions */
  sl.head[0] = ll.head;                      /* Set head of first sublist */

  for (i = 1; i < sl.n; i++) {               /* Save tails of sublists */
    sl.head[i] = ll.next[sl.random[i]];        /* Gather heads and save */
    sl.value[i] = ll.value[sl.random[i]];      /* Gather values at tails and save */
                                             /* Set up sublists */
    ll.value[sl.random[i]] = ZERO;             /* Scatter zero at tails values */
    ll.next[sl.random[i]] = sl.random[i];      /* Scatter self loops at tails */
  }
  for (i = 0; i < sl.n; i++) {               /* Initialize virtual processors */
    vp.next[i] = sl.head[i];                   /* Store heads */
    vp.sum[i] = ZERO;                          /* Initialize sums */
    vp.proc[i] = i;                            /* Assign processor id's */
  }
}
```

The time for INITIALIZE in CRAY C-90 clock cycles (4.2 nsec) is

$$T_{\text{Initialize}}(m) = 13m + 8700,$$

where m is the number of sublists.

Initial_Rank: INITIAL_RANK traverses each sublist for *n_steps* times computing the sum of the links. Because the weight of the tail is *zero*, it can repeatedly accumulate the sum at the tail without affecting the sum. Each traversal of the vector of links requires retrieving the values and links at arbitrary locations in *ll*. Thus, it uses two gather operations. To increment the sum requires loading, adding to, and storing *vp.sum*. Finally it needs to store the current link *vp.next*.

```
Initial_Rank(vp, ll, n_steps)
{
  for (j = 0; j < n_steps; j++)                 /* dereference n_steps times on each sublist */
    for (i = 0; i < vp.n; i++) {
    vp.sum[i] += ll.value[vp.next[i]];            /* Gather value and increment sum */
    vp.next[i] = ll.next[vp.next[i]];             /* Gather successor link */
    }
}
```

The time for inner loop of INITIAL_RANK is

$$T_{\text{Initial\_Rank step}}(x) = 3.4x + 80,$$

where x is the vector length of *vp*.

Strip mining currently is performed on the inner loop. However, it would be more efficient to do strip mining of the inner loop outside the outer loop. In this way, we would not need to load and store intermediate results between successive iterations of the outer loop. The only way to get this effect on the CRAY is to unroll the inner loop. We did not do this optimization.

Initial_Pack: After traversing the sublists s_i steps LIST_SCAN packs out any completed list. Packing requires saving the results of the completed lists according to their processor id's and then packing the remaining lists in *vp* so that they are contiguous. A sublist is complete if the virtual processor has reached the tail of the sublist, which is a self loop. To test for a self loop requires loading *vp.next*, gathering *ll.next* at *vp.next*, and testing whether the two are equal. There are two ways we could get the compiler to save the completed lists. One is to compute the indices of the completed lists and then using these indices to gather *vp.sum*, *vp.next*, and *vp.proc_id*. Then it can scatter *vp.sum* to *sl.sum* and *vp.next* to *sl.tail* at the indices in *vp.proc_id*. The other way is to load *vp.sum*, *vp.next*, and *vp.proc_id*, change them so that all active sublists use one of the completed sublist values by using a vector merge operation. Then, as before, it can scatter *vp.sum* to *sl.sum* and *vp.next* to *sl.tail* at the indices in *vp.proc_id*. The effect is to have all active sublists write to the same location in *sl.sum* and *sl.tail*. This approach causes much memory contention because most of the sublists are active. Clearly, the former approach is better and is what we used.

```
Initial_Pack(vp, sl, ll)
{
  j = 0;
  for (i = 0; i < vp.n; i++) {
    if (vp.next[i] == ll.next[vp.next[i]]) {    /* Save completed sublists */
      sl.sum[vp.proc_id[i]] = vp.sum[i];          /* Gather and scatter sum */
      sl.tail[vp.proc_id[i]] = vp.next[i];        /* Gather and scatter tail */
    } else {                                     /* Pack remaining sublists */
      vp.proc_id[j] = vp.proc_id[i];              /* Gather and store processor id's */
      vp.sum[j] = vp.sum[i];                      /* Gather and store current sum */
      vp.next[j++] = vp.next[i];                  /* Gather and store current link */
    }
  }
  return j;                                      /* Number of remaining sublists */
}
```

Packing the remaining active sublists is done by computing the indices of the active sublists and for each vector, *vp.next*, *vp.sum* and *vp.proc_id*, gathering the vector using the active indices and then storing contiguously. The time for one application of INITIAL_PACK for vectors of length x is:

$$T_{\text{Initial\_Pack step}}(x) = 7x + 540.$$

Find_Sublist_List: At this point each sublist has reached its tail and is ready to start Phase 2. Recall that *sl.tail* holds the tail of each sublist, while *sl.random* holds the tail of the previous sublist. Therefore, when FIND_SUBLIST_LIST writes the sublist index to *ll.next* at *sl.random*, then it is writing the index of the successor sublist to the tails of the sublists. (We write to *ll.next* because we can easily regenerate the self loops there.) This write requires loading *sl.random* and then scattering the index to *ll.next* at *sl.random*. Note that *sl.random* does not contain the index of the tail of one sublist, namely the tail of the whole list. Therefore, if it writes the negative index it can distinguish between values set at *sl.random* and the original self loops in *ll.next*. Next FIND_SUBLIST_LIST gathers these indices from *ll.next*, but this time using *sl.tail*. These indices are the indices of the successor sublists as long as they are negative. Only one index is positive and it is the index of the tail of the whole list. Notice that the writing and reading of the indices is done in separate loops because the reading of the indices may not be done in the same order as the writing. That is, we need to be sure that the write is complete before the read starts and that no chaining is allowed.

```
Find_Sublist_List(ll, sl)
{
  for (i = 1; i < sl.n; i++)
  ll.next[sl.random[i]] = -i;            /* Scatter index of next sublist */
  for (i = 0; i < sl.n; i++) {           /* Create list of sublists */
    next = ll.next[sl.tail[i]];            /* Gather index of next sublist */
    sl.next[i] = -next;                    /* Store the index */
    if (next > 0){                       /* Found tail of whole list */
      sl.next[i] = i;                      /* Set tail sublist to self loop */
      sl.random[0] = next;                 /* Save tail of whole list */
      sl.value[0] = ll.value[next];        /* Save its value */
      ll.value[next] = ZERO;               /* Set tail value to the identity */
    }
  }
  for (i = 0; i < sl.n; i++) {
    ll.next[sl.tail[i]] = sl.tail[i];    /* Scatter self loops at tails */
    sl.sum[i] += sl.value[sl.next[i]];   /* Gather tail values and increment sum */
    vp.next[i] = sl.head[i];             /* Reset virtual processor heads */
  }
}
```

Once FIND_SUBLIST_LIST finds the tail sublist it sets the tail of *sl.next* to a self loop, saves the tail of the whole list and its value in *sl.random*[0] and *sl.value*[0], and sets the value of tail of the whole list to *zero*. Note that it was not necessary to set the value of the tail of the whole list to *zero* for the Phase 1 because we do not need the correct sum of the tail sublist to find the scan of the reduce list. But in Phase 3 we need tail value set to *zero* because, otherwise, we may repeatedly add the tail value to the scan at the tail.

Finally, FIND_SUBLIST_LIST returns the tails of the sublists to self loops, which requires loading *sl.tail* and scattering it to *ll.next*. Since the tail values were never added to the sublist sums during INITIAL_RANK it next adds the tail values to the sublist sums. The tail values were saved in *sl.value* during INITIALIZE by the successor sublist and therefore must be indexed by *sl.next*. It loads *sl.sum*, gathers *sl.value* using *sl.next*, adds the values to the sums, which are then stored. Lastly, it reinitializes the virtual processor heads in anticipation of Phase 3. Reinitializing the heads requires loading *sl.head* and storing it in *vp.next*.

The time for FIND_SUBLIST_LIST is

$$T_{\text{Find\_Sublist\_List}}(m) = 9m + 770,$$

where m is the number of sublists.

Scan of the Reduced List: Next LIST_SCAN finds the list scan on the sublists sums *sl.sum* using the list *sl.next*. If the list is large then it finds it recursively. If list length lies between the recursive cutoff and the serial cutoff it uses WYLLIE. If the list length is small it uses SERIAL_LIST_SCAN. The time for SERIAL_LIST_SCAN is:

$$T_{\text{Serial\_List\_Scan}}(m) = 44.1m + 255,$$

where m is the number of sublists.

Final_Rank: In Phase 3 LIST_SCAN repeatedly calls FINAL_RANK to traverse the sublists for *n_steps* steps. The scan of each sublist is found in the same manner as in Phase 1. The only difference is that FINAL_RANK scatters the resulting scan *vp.sum* to *ll.sum* at the current positions in *vp.next*.

```
Final_Rank(ll, vp, n_steps)
{
  for (j = 0; j < n_steps; j++)
    for (i = 0; i < vp.n; i++) {
      ll.sum[vp.next[i]] = vp.sum[i];          /* Load and scatter sums */
      vp.sum[i] += ll.value[vp.next[i]];       /* Gather value and increment sum */
      vp.next[i] = ll.next[vp.next[i]];        /* Gather successor link and store */
    }
}
```

The time for one iteration of the inner loop of FINAL_RANK is:

$$T_{\text{Final\_Rank step}}(x) = 5x + 100,$$

where x is the number of sublists remaining.

Final_Pack: The packing step in Phase 3 is a little simpler than the packing step in Phase 1. Only the completed lists need to write their sums to *ll.sum* because the active sublists will write their sums on the next call to FINAL_RANK. However, it is faster to simply load all of *vp.sum* and scatter to *ll.sum* than it is to compute the indices of the completed sublists and to gather *vp.sum* and scatter them to *ll.sum.* For active sublist, FINAL_PACK packs *vp.sum* and *vp.next* as in INITIAL_PACK. However, it does not need to keep track of the *vp.proc_id.*

```
Final_Pack(vp, sl, ll)
{
  j = 0;
  for (i = 0; i < vp.n; i++) {
    ll.sum[vp.next[i]] = vp.sum[i];          /* Load and scatter sums */
    if (vp.next[i] != ll.next[vp.next[i]]) { /* Pack remaining sublists */
      vp.sum[j] = vp.sum[i];                 /* Gather and store sums */
      vp.next[j++] = vp.next[i];             /* Gather and store links */
    }
  }
  return j;
}
```

The time for FINAL_PACK is:

$$T_{\text{Final\_Pack step}}(x) = 6x + 400,$$

where x is the number of sublists to be packed.

Restore_List: Finally each processor returns the original links and values at the sublist tails. This requires loading *sl.random*, *sl.head*, and *sl.value* and scattering to *ll.next* and *ll.value* using *sl.random.* Because the tail of the whole list is supposed to be a self loop anyway, it does not set *ll.next* at the tail.

```
Restore_List(sl, ll)
{
  ll.value[sl.random[0]] = sl.value[0];      /* Reset value of list tail */

  for (i = 1; i < sl.n; i++) {
    ll.next[sl.random[i]] = sl.head[i];      /* Scatter links at tails */
    ll.value[sl.random[i]] = sl.value[i];    /* Scatter values at tails */
  }
}
```

The time for RESTORE_LIST is

$$T_{\text{Restore\_List}}(m) = 4m + 250,$$

where m is the number of sublists.

4 Analysis of the Algorithm

In Phases 1 and 3 of the algorithm we periodically perform load balancing so that virtual processors that have finished their sublists are removed from the computation. We would like to pack as soon as there are several finished sublists. However, if we pack too frequently we pack 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 appropriate 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, given n the length of the linked list, m the number of sublists, and s_1 the number of ranking steps to perform before the first pack, we determine how to minimize the costs of the packs and the unnecessary tail chasing. In Section 4.4 we discuss how to find m and s_1 given n. Finally, we summarize the costs, by giving an estimate of the overall performance of the algorithm and comparing it with the actual performance. The main theorem of this section 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 approximately follow a negative exponential distribution, when n and m are large. The analysis is from Feller [10]. 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 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 $E(X_{(i+1)} - X_{(i)}) = \frac{1}{m}$.*

Lemma 3 *If $X_1, \ldots, X_m$ are independent and uniformly distributed over the range (0,1) then*

$$Prob\{X_{(1)} > \frac{t}{m}\} = (1 - \frac{t}{m})^m \quad \underset{\infty}{\overset{m}{\rightarrow}} \quad e^{-t}.$$

Proof: The length of $(0, X_{(1)})$ exceeds t/m iff all $X_1, \ldots, X_2$ are in the interval $(\frac{t}{m}, 1)$. Because the events are independent and uniformly distributed the probability of combined events happening is $(1 - \frac{t}{m})^m$. As $m \to \infty$ this probability tends to e^{-t}. Thus, the distribution of the first interval in the limit is a negative exponential with mean m^{-1}. ∎

Lemma 4 *(without proof) If $X_1, \ldots, X_k$ follow an exponential distribution with expected value μ^{-1} then $X_1 + \cdots + X_k$ follows a gamma distribution*

$$G_{\mu,k}(x) = Prob\{X_1 + \cdots + X_k < x\} = 1 - \sum_{j=0}^{k-1} (\mu x)^j e^{\mu x}/j!.$$

Lemma 5 *If $X_1, \ldots, X_m$ are independent and uniformly distributed over the range (0,1) then for every fixed k*

$$Prob\{X_{(k)} > \frac{t}{m}\} = \sum_{j=0}^{k-1} \left(\begin{array}{c} m \\ j \end{array} \right) \left(\frac{t}{m}\right)^j (1 - \frac{t}{m})^{m-j} \quad \underset{\infty}{\overset{m}{\rightarrow}} \quad \sum_{j=0}^{k-1} t^j e^{-t}/j!,$$

which is the tail of the gamma distribution $G_{m,k}$.

Proof: In order for $X_{(k)} > \frac{t}{m}$, less than k of the X's lie in the range $(0, \frac{t}{m})$. Because the X events are independent and uniformly distributed the probability that exactly j of the X's lie in the range $(0, \frac{t}{m})$ follows a binomial distribution with probability of success equal to $\frac{t}{m}$ and probability of failure equal to $1 - \frac{t}{m}$. That is,

$$\begin{aligned}\binom{m}{j}\left(\frac{t}{m}\right)^j\left(1-\frac{t}{m}\right)^{m-j} &= \frac{m(m-1)\cdots(m-j+1)}{m^j}\frac{t^j}{j!}\left(1-\frac{t}{m}\right)^{m-j} \\ &\xrightarrow[\infty]{m} t^j e^{-t}/j!\end{aligned}$$

To obtain Prob$\{X_{(k)} > \frac{t}{m}\}$ we need to sum over the range $j = 0, \ldots, (k-1)$. ∎

Proof (of Proposition):

$X_{(k)}$ is the sum of the first k intervals, $X_{(1)}, X_{(2)} - X_{(1)}, \ldots, X_{(k)} - X_{(k-1)}$. In the limit as, $m \to \infty$, $X_{(k)}$ follows a gamma distribution with parameters m, k, which is the distribution of the sum of k mutually independent exponential variates with expectation $\frac{1}{m}$. Therefore, the successive intervals in the limit behave at though they are mutually independent exponential variates. ∎

In our case, we are choosing m random positions in a list of length n. Because these are random positions we can assume the list is laid out in order from left to right. Then the lengths of the sublists are the intervals determined by m random integral numbers $Y_1, \ldots, Y_m$ from 0 to $n-1$. Let

$$\begin{aligned} L_0 &= Y_{(1)} \\ L_i &= Y_{(i)} - Y_{(i-1)} \\ L_m &= n - Y_{(m)} \end{aligned}$$

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

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

If we let $a = (m + .5)/(m + 1)$ and solve for x we get an estimate of the expected length of the shortest sublist of $m + 1$ sublists, namely

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

If we let $a = .5/(m + 1)$ and solve for x we get an estimate of the expected length of the longest sublist, namely

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

In general, we can estimate the expected length of the i^{th} smallest 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 $m > 100$ for all but the smallest sublist. A better estimate for the smallest sublist seems to be

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

Figure 11 shows the expected length of the i^{th} 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 increase.

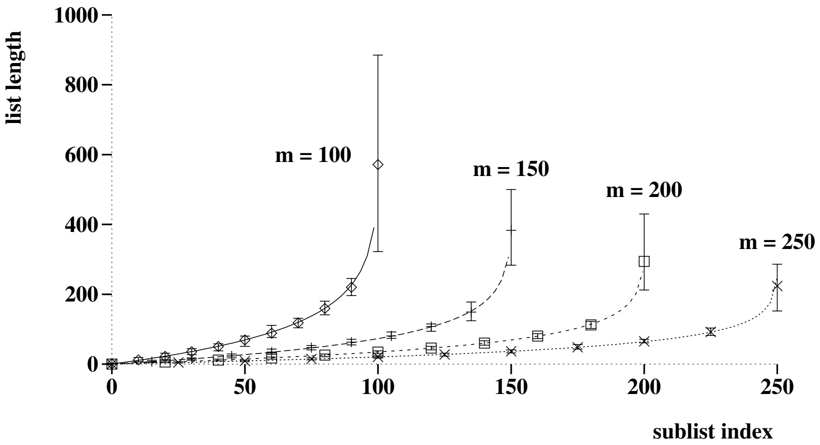

Figure 11: The function curves are the expected length of the i^{th} shortest sublist when $n = 10000$ for several values of m, the number of sublists. The observed lengths were found by taking 20 samples of dividing a list of size $n = 10000$ into m sublists and for the collection of the i^{th} shortest sublist of each sample, finding the average length (shown with a data symbol) and the minimum and maximum lengths (shown with an error bar).

4.2 Cost of the algorithm

Using the timing equations of each piece of the algorithm we can determine the cost of the complete algorithm, assuming we know the exact lengths of the sublists and when packs are performed. Assume we traverse $s_i, i = 1, \ldots, l$ links of each list between packs. Let S_i be the total number of links traversed in each list before the i^{th} pack. That is,

$$\begin{aligned} S_0 &= 0 \\ S_i &= \textstyle\sum_1^i s_i, & i = 1, \ldots, l \\ s_i &= S_i - S_{i-1}, & i = 1, \ldots, l. \end{aligned}$$

Let $g(x)$ be the expected number of sublists that have length greater than x. From the previous section

$$\begin{aligned} g(x) &= m \times Prob(\text{sublist length} > x) && (1) \\ &\approx me^{-\frac{mx}{n}} && (2) \end{aligned}$$

The dotted line in Figure 12 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 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 12 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.

Because the s_i's are the same for both Phases 1 and 3, we can combine the costs of INITIAL_RANK and FINAL_RANK to get a single cost equation for ranking. Similarly we can combine the costs of INITIAL_PACK

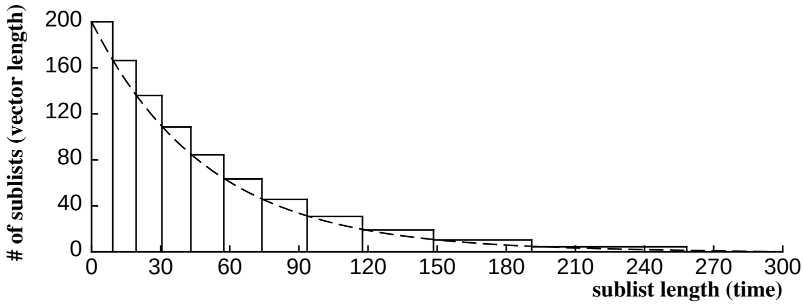

Figure 12: The dotted function is $g(x)$ the expected number of sublists that have length greater than x, where $n = 10000$ and $m = 200$. When the number of packing steps is 11, the expected execution time on the CRAY C-90 is minimized by packing at the vertical lines. The step function shows the expected 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.

and FINAL_PACK to get a single cost equation for packing. Thus, the costs of a single rank step and a single pack are:

$$\begin{aligned} T_{\text{Rank step}}(x) &= 8.4x + 180 \\ T_{\text{Pack step}}(x) &= 13x + 940, \end{aligned}$$

where x is the vector length over which the rank and the pack are taking place. The expected total time of all the packs are:

$$\begin{aligned} T_{\text{Pack}} &= \sum_{i=0}^{l-1}[13g(S_i) + 940] \\ &= 13\sum_{i=0}^{l-1} g(S_i) + 940l, \end{aligned}$$

where $g(S_i)$ is the expected number of sublists remaining after S_i steps of list ranking. Because we do s_i steps of list ranking between packs, the expected total time of the list ranking is:

$$\begin{aligned} T_{\text{Rank}} &= \sum_{i=0}^{l-1}[s_{i+1}(8.4g(S_i) + 180)] \\ &= 8.4\sum_{i=0}^{l-1} s_{i+1}g(S_i) + 180S_l \\ &\geq 8.4n + 180S_l, \end{aligned}$$

where S_l is the first S_i greater than the length of the longest sublist. If we pack everytime a sublist completes then $\sum_{i=0}^{l-1} s_{i+1}g(S_i) = n$. However, in general we delay packs and the sum is greater than n (see Figure 12).

Similarly, we can combine the times of INITIALIZE, FIND_SUBLIST_LIST, and RESTORE_LIST since they depend on the number of sublists only. These combined times are:

$$T_{\text{Other}}(m) = 26m + 9720.$$

Thus, the expected total time for Phases 1 and 3 the algorithm is:

$$T_{\text{P1+P3}} = T_{\text{Rank}} + T_{\text{Pack}} + T_{\text{Other}} \tag{3}$$

4.3 Minimizing the time given fixed parameters

Suppose we are given n the length of the original linked list, m the number of sublists and l the number of times to pack. How do we decide when to pack? The simplest way is to divide l in to the expected length of the longest sublist and pack every fixed number of intervals. However, from the previous discussion we know that the lists do not drop out at a constant rate. That is, $g(x)$ the expected number of sublists that have length greater than x is not linear; it is exponential. In this section we determine where to pack in order to minimize the expected execution time of the algorithm, given n, m, and l. In the next section, we describe how we determine m and l given n.

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 occurred at the i^{th} pack. That is, we want to minimize the following function:

$$T_{P1+P3}(S_0, \ldots, S_l) = \sum_{0}^{l-1}[(S_{i+1} - S_i)(ag(S_i) + b) + cg(S_i) + d] + em + f, \tag{4}$$

where $T_{\text{Rank step}}(x) = ax + b$, $T_{\text{Pack step}}(x) = cx + d$, and $T_{\text{Other}}(x) = ex + f$. We can minimize Equation 4 by taking partial derivatives for each S_i and setting them to zero to obtain a set of l simultaneous equations. That is,

$$\frac{\partial T}{\partial S_i} = a(S_{i+1} - S_i)g'(S_i) - (ag(S_i) + b) + (ag(S_{i-1}) + b) + cg'(S_i) = 0$$

$$g'(S_i) = \frac{g(S_{i-1}) - g(S_i)}{S_{i+1} - S_i + \frac{c}{a}} \tag{5}$$

Each equation specifies that for a set of three consecutive S's, S_{i-1}, S_i, and S_{i+1} the S_i is located where the slope of g at S_i is equal to the slope of the line through the points $(S_i, g(S_{i-1}))$ and $(S_{i+1} + \frac{c}{a}, g(S_i))$, see Figure 13.

Because g is the exponential function it is not obvious how to solve for S_i given S_{i-1} and S_{i+1}. However, it is not difficult to solve for S_{i+1} given S_i and S_{i-1} (or to solve for S_{i-1} given S_i and S_{i+1}). Namely,

$$\begin{aligned} S_{i+1} &= S_i - \frac{g(S_{i-1}) - g(S_i)}{g'(S_i)} - \frac{c}{a} \\ &= S_i + \frac{g(S_{i-1}) - g(S_i)}{\frac{m}{n}g(S_i)} - \frac{c}{a}, \end{aligned} \tag{6}$$

since $g(x) = me^{-mx/n}$. 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.

The vertical lines in Figure 12 were found using $S_1 = 14.7$ and the equations in Section 3. Notice 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 6 reflects the relative cost of packing and ranking. To see the effect of this factor, consider how the spacing of packs over all the iterations changes if we keep the total number of packs the same but increase the value of c. When c is increased, packing would occur less frequently during the initial iterations of ranking and occur more frequently during later iterations. That is, the vertical lines in Figure 12 would be further apart for small iteration numbers and closer together for

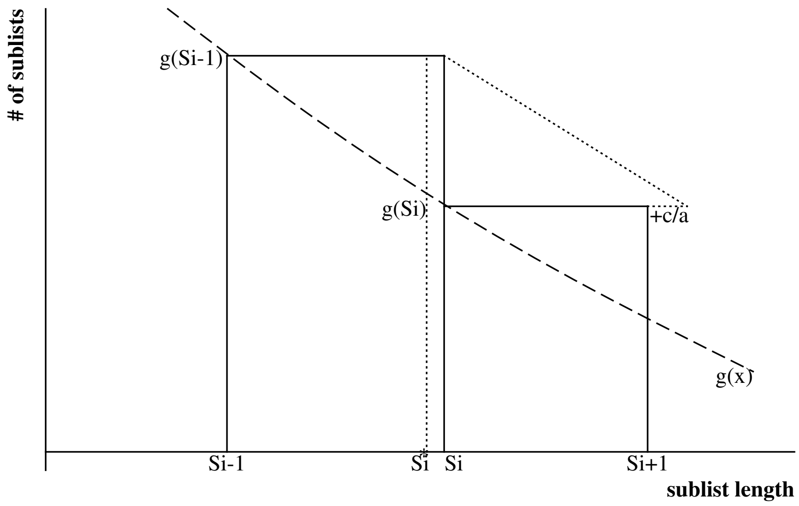

Figure 13: Time is minimized when, for each set of three consecutive S's, S_{i-1}, S_i, and S_{i+1}, the slope $g(S_i)$ is equal to the slope of the line through the points $(S_i, g(S_{i-1}))$ and $(S_{i+1} + \frac{c}{a}, g(S_i))$. $S_i^\star$ is the point where time would be minimized if there was no cost for packs.

large iteration numbers if we increase c. This reflects the fact that initially packing is expensive because the vector lengths are long and later packing become less expensive because the vector lengths are short. If we make c large enough eventually we find that the execution time is reduced by decreasing the number of packs even though the number of ranking steps increases. In the next section we consider how to determine the best number of packs to perform.

We can simplify Equation 4 by using equation 6 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,

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

For the CRAY C-90

$$T_{P1+P3} \approx 8.4n + 180\frac{n}{m}\ln m + (8.4S_1 + 39)m + 940l + 9720. \quad (7)$$

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 7, 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 $\log n$. It is these fitted polylog functions that we use in our implementation to determine m and S_1 given n and Equation 6 to find successive values of S_i.

However, we found that S_1 was a very sensitive parameter. From the previous section our intuition is that packing should occur less frequently as we proceed though the processing. However, if S_1 is too small, the packing steps become rapidly closer and closer until we are packing at every iteration of list ranking. The result is far too much packing and performance degrades rapidly. To protect ourselves from this sensitivity we modified Equation 6 so that successive S's are always increasing. With this modification we found the the fitted cubic log functions performed very well in practice.

Figure 14 compares the predicted time with the observed running time. The predicted time was computed by estimating the parameters values for each value of n using the fitted cubic equations and then applying the equation 3 for those parameter values. As the figure indicates the equation is an accurate predictor of the running time. Notice that the running time decreases until it reaches an asymptote of about 8.6 clocks per element.

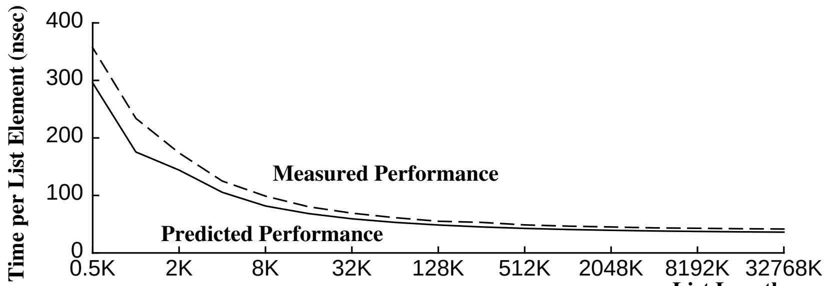

Figure 14: The predicted performance and measured performance of the vectorized LIST_SCAN on one processor CRAY C-90. The values for the parameters m and S_1 were determined by minimizing the predicted performance.

5 Vector Multiprocessor List Scan

In Section 3 we showed how we implemented the vectorized version of the data parallel list scan algorithm. In this section we show that extending the algorithm to multiple vector processors is relatively straightforward. We then discuss issues relating to the vector multiprocessor version performance and its speedup with respect to the single vector processor version.

The overall approach is to 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 using compiler directives so that different iterations of the loops are divided among the processors. Because our arrays are often longer than the vector length and we know that the loops can be vectorized, we chose to direct the compiler to divide the loops into equal size chunks, one chunk per processor, and to vectorize the chunk within each processor.

For MIMD processors we also tried to minimize the number of synchronization points. Data parallel algorithms assume that each data parallel step is synchronized, whether or not it is necessary. In the code we presented in Section 3 we need to synchronize at most after every innermost loop. In particular, we must synchronize after the loops in INITIALIZE and FIND_SUBLIST_LIST and after Phase 1 and Phase 3 for correctness. If we use the parallel algorithm as described in Section 2 and have equal number of active virtual processors assigned to physical processors at all times, we also need to synchronize before each pack in Phase 1 and Phase 3 so that load balancing can proceed globally across the physical processors.

However, we deviated somewhat from this strict form of assignment of virtual processors. Instead we assign them 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. The effect is that we need to do no synchronization within Phase 1 or Phase 3 and there is 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 do a pack operation across processors in parallel. Of course, with some effort we could apply loop raking to get a vector multiprocessor algorithm for pack [5].

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 the procedure progresses, only one across-processor load balancing should be necessary. Our results are quite good without any global load balancing. If we ignore load imbalance and synchronization costs we can get an estimate of the execution time of Phase 1 and Phase 2 by dividing the vector lengths equally among the processors. Namely,

$$\begin{aligned} T_{P1+P3}(S_1,\ldots,S_l) &\approx \sum_0^{l-1}[(S_{i+1}-S_i)(ag(S_i)/p+b)+cg(S_i)/p+d]+em/p+f \\ &\approx an/p+b\frac{n}{m}\ln m+(aS_1+c+e)m/p+ld+f \\ &= O(n/p+\frac{n}{m}\ln m), \end{aligned}$$

where p is the number of processors and $m < n/\log n$.

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. We tuned the parameters for 1, 2, 4, and 8 processors and result in the execution times shown in Figure 15. For 8 processors we achieve a speedup of 6.7.

6 Other work efficient list ranking algorithms

On one CRAY C-90 vector processor our algorithm takes about 10 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 10 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 these 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 nodes long (the heads of the sublists are called 2-ruling sets); reduces the sublists to a single nodes; and then compacts these single nodes into contiguous memory to create a new linked

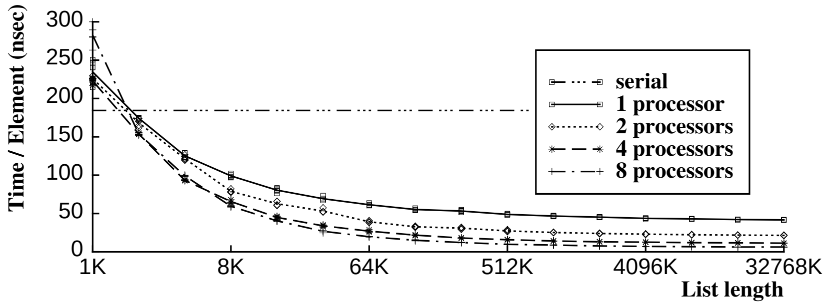

Figure 15: Execution times per element on 1, 2, 4, and 8 dedicated processors of the CRAY C-90 for our list ranking algorithm.

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 nodes. 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, 22]. 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 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 assigned $\log n$ nodes which they process. At each round each processor executes a case statement that either breaks contention or splices out a node in its queue or splices out a node at another processor's queue. To break contention the algorithm finds a $\log\log n$ ruling set. Finding $\log\log n$ ruling sets is much faster ($O(1)$ 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, the algorithm's constants are also much larger than ours.

The basic structure of Cole and Viskin's algorithms is similar to the structure of our algorithm. The main difference is that we break the linked list into a relatively few sublists that can be quite long, whereas Cole and Viskin divide the linked list into more than $n/3$ sublists that are only two or three nodes long. Getting such fine grain list lengths is quite expensive and needs to be repeated $O(\log n)$ times. We find sublists only a few times, and because our sublists are relatively long, we can process the lists at full speed for their entire length. The primary reason our algorithm is so successful is that it has very small constants and is work efficient. And as long as the number of processors is small relative to the size of the list the parallel running time is optimal. 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 at load balancing points.

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 and assign them equally among the processors. Our algorithm only tries to find a relatively small number, m, of such elements. 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 main ranking protion 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, 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] Richard Anderson and Gary 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] Richard Anderson and Gary L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

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

[5] Guy E. Blelloch, Siddhartha Chatterjee, and Marco Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, pages 416–424, March 1992.

[6] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

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

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

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

[11] Hillel Gazit, Gary L. Miller, and Shang-Hua Teng. Optimal tree contraction in the EREW model. In Stuart K. Tewsburg, Bradley W. Dickinson, and Stuart C. Schwartz, editors, *Concurrent Computations*, pages 139–156. Plenum Publishing Corporation, 1988.

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

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

[14] John L. Hennessy and David 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] Michael 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] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, October 1985.

[18] Gary L. Miller and John H. Reif. Parallel tree contraction. 2. further applications. *SIAM Journal of Computing*, 20(6):1128–1147, December 1991.

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

[20] Margaret Reid-Miller, Gary L. Miller, and Francesmary Modugno. List ranking and parallel tree contraction. In John Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufmann, 1993.

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

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

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

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

[25] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.