Portable Randomized List Ranking on Multiprocessors Using MPI

Jesper Larsson Träff[1]

Technische Universität München, Lehrstuhl für Effiziente Algorithmen
D-80290 München, Germany
email: `traeff@informatik.tu-muenchen.de`

Abstract. We describe a simple multiprocessor list ranking algorithm with low communication volume and simple communication pattern. With p processors the algorithm performs $< 4p$ (pipelined) communication rounds involving only point-to-point communication. For lists with N elements the algorithm runs in $O(N \ln p/p + p)$ time. Experiments with an implementation using MPI on a network of workstations and an IBM SP-2 comparing the algorithm to the well-known pointer jumping algorithm are reported. On the NOW the new algorithm is significantly better than pointer jumping. On the IBM SP-2 only the new algorithm was able to produce (modest) speed-up.

1 Introduction

The *list ranking problem* is a prototypical irregular problem of fundamental importance in the design of parallel graph algorithms under the PRAM model of parallel computation [4]. Many problems on trees and graphs can be reduced to list ranking, and the problem is of relevance in parallel computational biology and computer vision. Hence, the problem may serve as a *benchmark problem* for the feasibility of parallel combinatorial algorithms on parallel computers. In its simplest form the problem consists in computing for each element of a linked list the distance to the end of the list, ie. the number of links that have to be traversed to reach the last element of the list. Sequentially the problem is solved in linear time in the length of the input list; but the irregular nature of the problem manifests itself also in the sequential setting when solving very large problem instances [8]. In a parallel setting, where the list is divided among several processors, the problem is considerably more difficult, giving rise to algorithms with data dependent, irregular communication patterns. The problem is well-studied in the theoretical literature on parallel algorithm design [4], and has recently attracted considerable interest also from a more practical point of view [1, 3, 6, 7, 9, 11].

In this paper we describe portable implementations using the message passing interface MPI [10] of two algorithms for the problem and evaluate their performance on a small network of workstations (NOW) and on a powerful IBM SP-2 distributed memory multiprocessor. The first algorithm is the standard pointer-jumping algorithm of Wyllie [13]. Due to its simplicity this algorithm typically serves as a point of reference against which to compare other algorithms. As will be seen pointer jumping is not a very good algorithm for the NOW. The second algorithm is based on a different approach of ranking sublists on one processor by "folding" already ranked sublists on other processors into these sublists. Although also theoretically non-optimal, this algorithm has the advantage of a simpler communication pattern than the pointer jumping algorithm, and empirically it performs substantially better: on a NOW it is easily a factor of 5 faster. On the SP-2 only the new algorithm was able to produce speed-up, although limited.

The algorithms are implemented using MPI [10] for programming in a simple message passing model. The prime mode of communication MPI is by explicit point-to-point message passing, but in addition MPI provides a set of convenient collective operations. In contrast to other paradigms for portable parallel programming, like PVM [2] or BSP [12, 5], MPI does not force upon the programmer a particular virtual machine or programming style, although it is possible to use MPI for programming in a BSP-like style. The presence of collective operations makes MPI more flexible than BSP. Algorithm complexity can be measured in terms of basic MPI-communication operations, and local computation.

The list ranking problem is defined in Section 2. Sections 3 and 4 describe the two algorithms, and Section 5 gives a preliminary experimental evaluation.

2 The problem

Given a set of *list elements* each having a successor index pointing to some other element, or to a special *tail* element. Each element has at most one element pointing to it, and there are no cycles. Each element has an integer rank associated with it. The (inclusive) *list ranking problem* consists in computing for each list element the sum of the rank of the element itself and the ranks of all elements that succeed it. In addition a pointer to the last reachable element must be computed for each element. Note that in this formulation of the problem the set of list elements may contain several disjoint linked lists. By the second requirement each element will know after ranking to which list it belongs by having a pointer to the unique last element of that list. The special case of computing for each element the distance to the end of the list is obtained by setting the initial rank of all elements to 1. Finally note that lists are *singly linked.*

When solving the list ranking problem on multiprocessors the set of list elements is assumed to be distributed over a set of processors. The p processors, $P_1, \ldots, P_p$, have no shared memory and no common clock, but can communicate and synchronize by sending messages to each other. Each processor can communicate with any other processor through an interconnection network, the nature of which is left unspecified. The total number of list elements is denoted by N, and the number of list elements per processor, which we assume to be (roughly) equal, by n, eg. $N = \Theta(pn)$. The list elements are stored in arrays in some arbitrary order; in particular successor pointers and ranks are stored in arrays `list` and `rank` of size n, such that `list[i]` and `rank[i]` are the successor pointer and rank of the ith local list element. The list ranking procedures have the following MPI like prototype:

```
Listrank(Index list[], int rank[], int n, MPI_Comm comm)
```

Type `Index` represents a pointer to a successor element, and consists of the id (rank) of the processor at which the successor resides, and the index of the successor in the `list` array of that processor. Upon return `rank[i]` is the computed rank of the ith local element, and `list[i]` a pointer to the last reachable element. In the implementations of this paper an `Index` is a structure consisting of two integers. The `Listrank()` implementations can all be generalized to solve the more general *list scan* problem, in which the rank is of arbitrary type and some associative operation takes the place of addition.

Complexity is measured in terms of local computation and basic communication operations under the assumption of roughly synchronous operation of the processors. This assumption is only for the analysis; if needed synchronization can be enforced by appropriate `MPI_Barrier()` calls. The algorithms are randomized in the sense that a random distribution of the list elements over the p processors is assumed. This means that the probability of a list element at processor i to have its successor at processor j is $1/p$, independently of i and j. If this assumption is not fulfilled, the algorithms may run longer (but will still be correct). Alternatively, a random permutation of the list can be performed before list ranking.

3 List ranking by pointer jumping

List ranking by pointer jumping is done in a number of synchronized (super)steps. Each list element maintains a current successor and a current rank, the latter being the sum of the ranks of the successor elements up to but not including the current successor. In a superstep each element adds the rank of its current successor to its own rank, and updates its current successor to the current successor of the current successor. When the current successor points to an element which has already been ranked, correct rank (and last element pointer) has been computed, and the element does not have to be considered in subsequent supersteps. It is easy to see that the number of list elements "jumped over" doubles in each superstep, so that $\lceil \log_2 N \rceil$ supersteps are required. The algorithm can immediately be parallelized, since the list elements are treated independently of each other. The number of operations carried out in step $d, d = 1, \ldots, \lceil \log_2 N \rceil$, is $O(N - 2^{d-1})$ (since in step d the 2^d elements with distance $< 2^d$ to a last element have been ranked), so the total "cost" of the pointer jumping algorithm is $O(N \log N)$, in contrast to the linear number of operations required by a simple sequential algorithm. In this sense the pointer jumping algorithm is not (work) optimal. In a multiprocessor implementation accessing the information in the current successor elements is done by sending requests to the appropriate processors. A superstep requires 3 all-to-all communication operations. First each processor buckets its requests for current successor information and sends the bucket sizes to all other processors. Second the requests themselves are sent. Finally current successor and current rank are returned in acknowledgment to the requests. A request consists of a single integer (index in local list array) per list element. The information returned consists of index (2 words) and rank information. The extra space needed is about $8n$ words per processor; contrast this to the n words of the sequential algorithm.

There are two orthogonal improvements of the basic algorithm. For the special case of computing only the distance to the last reachable element, it is not necessary to send the rank of the current successor in every round, since, as is easily seen, this simply doubles in each round. Only when an element which has already been ranked is reached, does the rank of this element have to be sent. Hence, in all rounds but the last only the index of the current successor needs to be sent. This decreases the size of acknowledgment messages by one third. The other improvement consists in performing a local ranking on each processor after each superstep. On the assumption of a random distribution of the list elements this decreases the number of messages to be sent per round by a factor $1/p$. This version turned out to be by far the most efficient, and is used for reference in Section 5.

4 The fold-unfold algorithm

We now present a different list ranking algorithm. Let L be the global problem and L_i the set of list elements at processor i. Assume that the elements in each L_i have been ranked locally (the local rank of $x \in L_i$ is the rank of x treating each element having its successor outside L_i as a last element). Each L_i thus becomes a collection of sublists each with its successor outside L_i. By *folding L_i into L_j* we mean the following: each sublist in L_j having its successor in L_i has its rank updated with the rank of the successor sublist in L_i, and its successor updated to the successor of the L_i sublist; if the successor of the L_i sublist is in L_j the two L_j sublists are joined. We denote this operation by $\texttt{fold}(L_i, L_j)$. It runs in time proportional to the number of sublists of L_i folded into L_j by suitable bucketing of the sublists. The operation requires two communication steps, since processor j must first send a request (1 word) to processor i for rank information (3 words) for all L_j sublists with successor in L_i. The fold-unfold algorithm successively folds the sets L_k for $k = 1, \dots, p-1$ into L_p and looks as follows:

```
1. For all $k \in \{1, \dots, p\}$ do in parallel localrank($L_k$)
2. For $k = 1, 2, \dots, p-1$ do: for all $l \in \{k+1, \dots, p\}$ do (in parallel) fold($L_k, L_l$)
3. For $k = p, \dots, 3, 2$ do: for all $l \in \{1, \dots, k-1\}$ do (in parallel) getrank($L_l, L_k$)
```

After the kth round of Step 2 all sublists in L_l have their successor in $L_i, i \in \{k+1, \dots, p\}$, as had all sublists in L_k before round k. After completion of Step 2 final ranks for sublists in L_k can computed from the ranks of sublists in L_i. Correctness follows from these invariants. For the complexity we consider only the fold computations of Step 2; Step 3 is analogous.

Lemma 1. *The expected number of sublists per processor before round k is $\alpha = n\frac{p-k}{p+1-k}$.*

Proof. Before the first round processor $l, l \in \{1, \ldots, p\}$ has n list elements of which n/p have their successor in processor l. Hence the number of sublists is $n - n/p = n\frac{p-1}{p}$. Assume the claim holds before round k. In round k the α sublists in L_k are folded into $L_l, l \in \{k+1, \ldots, p\}$. Since $1/(p-k)$ of these have their successor in L_l, the number of sublists in L_l becomes $\alpha - \frac{\alpha}{(p-k)^2} = \frac{\alpha}{(p-k)^2}(p-k+1)(p-k-1)$. Substituting for α yields the claim for $k+1$.

In round k the sublists L_k are folded in parallel into all remaining subsets. Thus $O(n)$ words have to be communicated per round, giving an algorithm with running time $O(pn) = O(N)$, and no potential for speed-up. We solve this problem by *pipelining* the $\texttt{fold}(L_k, L_l)$ operations. This is possible since processor k cannot start folding before round k. It suffices that processor k has received fold information from processors $1, \ldots, k-1$ at round ck for some constant c. The pipelining also takes place in rounds, and works as follows: after receiving information from all lower numbered processors processor k waits one round and then performs the $\texttt{fold}(L_k, L_l)$ operations one after another in increasing order of l, doing one fold per round.

Lemma 2. *At round* $2(k-1)+i$ *processor* k *performs* $\texttt{fold}(L_k, L_{k+i})$.

Proof. Processor 1 can start folding immediately, thus in round i performs $\texttt{fold}(L_1, L_{1+i})$. Assume the claim holds for processors $1, \ldots, k-1$. Then processor $k-j$ performs $\texttt{fold}(L_{k-j}, L_k)$ in round $2(k-j-1)+j = 2(k-1)-j$. Hence processor k has received its last message (from processor $k-1$) in round $2(k-1)-1$. It waits one round and in rounds $2(k-1)+i$ performs $\texttt{fold}(L_k, L_{k+i})$.

Processor p has thus received information from all smaller numbered processors in round $2p-3$. An advantage of the pipelining scheme is that only point-to-point communication is called for.

Lemma 3. *The parallel time spent in exchanging messages in Step 2 is proportional to* $4n \ln p = O(n \ln p + p)$.

Proof. By Lemma 1 the number of words exchanged between processor k and $k+1$ in round k is $4\frac{n}{p+1-k}$. Summing gives $4n$ times the pth harmonic number.

Theorem 4. *The fold-unfold algorithm solves the list ranking problem in* $4p-6$ *communication rounds, each of which entails only point-to-point communication. The time spent in local computation and exchange of messages per processor is* $O(n \ln p + p)$.

The fold-unfold list ranking algorithm is non-optimal, but theoretically slightly better than the pointer jumping algorithm ($\log_2 N$ has been traded for $\ln p$). More importantly, the logarithmic overhead has been switched from a logarithmic number of phases (each having a non-negligible communication start-up latency) to a logarithmic overhead stemming from gradually increasing message lengths. An unattractive feature of the algorithm is that it is unbalanced; after the p first rounds of Step 2 processor 1 stands unemployed until Step 3. The extra space required by the algorithm is about $7n$ words per processor.

5 Experimental results

The algorithms have been implemented in C using MPI [10]. Experiments have been carried out on 1) a network of (different) HP workstations (80 to 125Mhz) connected via a standard 10Mbit/second Ethernet, using `mpich` version 1.0.12, and 2) an IBM SP-2 with 77 RS/6000 nodes; but only up to 32 67Mhz nodes were available. Running times are given in seconds, and are quite erratic on the SP-2; the machine at hand is run in batch mode and was often heavily loaded. The input lists are "random lists" generated by randomly permuting random local lists among the processors. Sequential running time is optimistically estimated as the sum of the times for locally ranking the locally generated lists. Results are reported for the best implementations of the two algorithms. For pointer jumping performing local ranking in each superstep gave an improvement of about 30% for 32 processors on the SP-2 compared to the standard implementation. The best version of the fold-unfold algorithm exchanges messages as described in Section 4; no explicit barrier synchronization was needed to make the pipelining work well. Attempts at enforcing division into rounds to prevent congestion by `MPI_Barrier()` calls lead to slowdown from 10% to 300%. Running times for fixed problem sizes for different numbers of processors have been measured, from which speed-up can be computed, see Tables 1 and 3. We have also measured the running times with problem size increasing linearly in the number of processors, ie. keeping $n \approx N/p$ fixed. The results are shown in Tables 2 and 4.

On the NOW the fold-unfold algorithm is clearly superior to pointer jumping, being easily a factor of 5 faster. Of course, none of the algorithms were able to produce actual speed-up, but with the limited communication capabilities of the Ethernet this was not expected. However, with the fold-unfold algorithm it is possible to solve problems larger than possible on a single workstation at a modest price: with 14 processors the "slowdown" for a list with 7000000 elements is only about a factor of 10 over the most optimistic estimate for the sequential running time. On the SP-2 the difference between the two algorithms is less striking, but significant. On the large problems fold-unfold is consistently a factor of two better than pointer jumping, and can better utilize the increasing number of processors. For $p = 32$ it gave speed-up close to two, both in the case where N was fixed, compared to the actual sequential running time (Table 3), as in the case where $N = np$ and sequential running time was estimated as p times local ranking time (Table 4). For the problem with $N = 4000000$ running times on the SP-2 differed a lot from run to run. For large problems where a straightforward sequential algorithm runs into trouble, a parallel algorithm might be of interest.

6 Discussion

We presented a new, simple list ranking algorithm for distributed-memory multiprocessors, and compared it to the well-known pointer jumping algorithm. On a network of workstations a significant improvement in performance was achieved. For an IBM SP-2 multiprocessor neither algorithms were good. Only the fold-unfold algorithm was capable of producing some small speed-up. On an Intel

		Pointer jumping						Fold-unfold					
N	Seq.	1	2	4	6	8	10	1	2	4	6	8	10
250000	0.58	0.75	2.67	9.21	16.59	41.65	64.04	0.93	2.88	4.44	5.07	9.84	11.65
500000	1.27	1.57	5.32	18.51	31.94	87.00	110.51	1.76	5.75	8.60	9.96	11.84	15.85
1000000	2.56	3.25	11.47	36.95	63.22	135.54	187.97	3.67	11.91	17.44	19.84	26.34	32.68

Table 1. Running times on the HP NOW for lists of fixed total length N for varying number of processors, and local list length $n = N/p$.

	n	Seq.	4	Seq.	6	Seq.	8	Seq.	10	Seq.	12	Seq.	14
Pj	250000	2.66	37.69	4.07	93.67	5.36	212.69	6.67	403.57	7.93	510.86	9.20	686.21
	500000	5.55	79.34	8.43	191.73	Out of Memory							
Fold	250000	2.69	16.91	4.01	29.71	5.33	50.97	6.62	68.30	7.90	94.27	9.18	126.34
	500000	5.55	35.29	8.46	66.13	11.34	97.21	13.82	143.72	16.75	179.10	19.22	211.76

Table 2. Running times on the HP NOW for lists with length proportional to the number of processors, $N = np$, for fixed local length n. The "sequential" running time is estimated as the sum of the running times for ranking a list of size n on each workstation.

Paragon [9] reports good speed-up of up to about 27 on 100 processors for problems of similar sizes to those considered here. However, it should be noted that even pointer jumping produced speed-up (up to 7 on 100 processors) on this machine. However, the Paragon had a better ratio between computation and communication speed than the SP-2. It would be interesting to test the performance of the algorithms of [9] on the SP-2. Reduction in problem size might lead to better algorithms. It is easy to devise a randomized scheme for reducing the list length by a factor of at least 1/2 in only two all-to-all communication rounds. Such a scheme is currently being implemented. Space limitations prohibit thorough discussion and comparison to relevant related work [1, 7, 9].

References

1. F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 25(1):1–16, 1997.
2. A. Geist, A. Beguein, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
3. T.-S. Hsu and V. Ramachandran. Efficient massively parallel implementation of some combinatorial algorithms. *Theoretical Computer Science*, 162(2):297–322, 1996.
4. J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

		Pointer jumping				Fold-unfold			
N	Seq.	8	16	24	32	8	16	24	32
2000000	5.58	10.18	7.45	6.03	5.33	6.36	4.47	3.59	3.04
4000000	11.34	19.18	15.91	16.48	14.55	14.05	9.49	7.22	6.11

Table 3. Running times on the IBM SP-2 for lists of fixed total length N for varying number of processors. For the large problem with $N = 4000000$ the sequential running time varied from 11.21 to 71.37 seconds.

	n	Seq.	8	Seq.	16	Seq.	24	Seq.	32
Pointer	1000000	21.09	33.28	41.51	58.71	64.93	111.36	86.90	133.68
	2000000	43.06	111.78	86.02	160.27	129.44	205.21	171.86	206.64
Fold	1000000	23.79	25.33	44.60	38.21	62.19	50.25	82.68	49.93
	2000000	42.82	50.24	85.98	71.73	129.27	91.05	172.84	103.59

Table 4. Running times on the IBM SP-2 for lists with length proportional to the number of processors and fixed local length n.

5. W. F. McColl. Scalable computing. In *Computer Science Today. Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61, 1995.
6. J. N. Patel, A. A. Khokhar, and L. H. Jamieson. Scalable parallel implementations of list ranking on fine-grained machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1006–1018, 1997.
7. M. Reid-Miller. List ranking and list scan on the cray C-90. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 104–113, 1994.
8. J. F. Sibeyn. From parallel to external list ranking. Technical Report MPI-I-91-1-021, Max-Planck Institut für Informatik, 1997.
9. J. F. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking. In *Solving Irregularly Structured Problems in Parallel (IRREGULAR'97)*, volume 1253 of *Lecture Notes in Computer Science*, pages 25–36, 1997.
10. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.
11. J. L. Träff. Parallel list ranking and other operations on lists. Technical Report SFB 124-D6 3/97, Universität des Saarlandes, Saarbrücken, Germany, Sonderforschungsbereich 124, VLSI Entwurfsmethoden und Parallelität, 1997. 69 Pages.
12. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
13. J. C. Wyllie. *The Complexity of Parallel Computation*. PhD thesis, Computer Science Department. Cornell University, 1979. Technical Report TR-79-387.