From Parallel to External List Ranking

Jop F. Sibeyn*

Abstract

Novel algorithms are presented for parallel and external memory list-ranking. The same algorithms can be used for computing basic tree functions, such as the depth of a node.

The parallel algorithm stands out through its low memory use, its simplicity and its performance. For a large range of problem sizes, it is almost as fast as the fastest previous algorithms. On a Paragon with 100 PUs, each holding 10^6 nodes, we obtain speed-up 25.

For external-memory list-ranking, the best algorithm so far is an optimized version of independent-set-removal. Actually, this algorithm is not good at all: for a list of length N, the paging volume is about $72 \cdot N$. Our new algorithm reduces this to $18 \cdot N$. The algorithm has been implemented, and the theoretical results are confirmed.

1 Introduction

A *linked list*, hereafter just *list*, is a basic data structure: it consists of nodes which are linked together, such that every node has precisely one predecessor and one successor, except for the *initial node*, which has no predecessor, and the *final node*, which has no successor. Connected to the use of lists is the *list ranking* problem: computing for each node i the final node j of its list, and the number of links between i and j. Once a set of lists has been ranked, it can be turned into an array, on which many operations can be performed more efficiently.

Parallel and external list ranking is a challenge, because it is hard to obtain good performance. In this paper we present novel algorithms for performing list-ranking on various models of parallel computers (ranging from PRAMs to practical parallel systems), and in external memory. The simplicity of the presented algorithms facilitates their implementation. At the same time they are highly efficient and out perform existing algorithms. In particular the external-memory algorithm is several times faster than the best existing algorithm. It requires three passes over the input, only one more than sorting.

The central point in the algorithms presented is a basic step that allows for an efficient split-off of a parametrizable fraction of the nodes. This may sound familiar, but it is done in a completely new way. A second important point (for the parallel algorithms) is that there are algorithms that solve a problem with many short lists considerably faster than an arbitrary list-ranking problem.

*Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: jopsi@mpi-sb.mpg.de. URL: http://www.mpi-sb.mpg.de/~jopsi/

1.1 Motivation

There are several reasons for performing a detailed study of the list-ranking problem for parallel and external applications. We distinguish three types of motivation, which are discussed hereafter:

- Theoretical interest.
- Benchmark character for the class of irregular problems.
- Practical applications as a subroutine in other problems.

The theoretical interest of list-ranking is evident: it is one of the most basic problems, and in the theory of parallel computation (and thus by "inheritance" also in the theory of external computation). Therefore it has been considered extensively [41, 9, 10, 11, 1, 2]. List ranking appears as a subroutine in many graph problems particularly because it is the key ingredient of the Euler-tour technique [39] (see [24] for a detailed description).

List-ranking has linear sequential complexity, and can be solved very efficiently by an almost trivial algorithm. This makes it very hard to achieve good speed-ups on a parallel computer, and means that by comparison one may expect to lose a rather large factor when solving the problem externally: the communication or paging can impossibly be hidden by the computation. Because the problem is in addition very irregular, we believe that the performance obtained for the list-ranking problem gives a kind of a lower bound on the performance that may be expected for general purpose parallel or external computing.

The above two reasons are already motivation enough. However, the list-ranking problem also has real practical importance. Here we must be extremely careful not to confuse applications in theory and applications in practice. For example, one might believe that list-ranking is an essential ingredient for rooting the trees that appear in most connected-components algorithms based on [22]. However, in practice most nodes of these trees will lie close to a root (if necessary one could randomize the indices of the vertices), and it will be much faster to apply a variant of pointer-jumping. Expression evaluation [30, 17, 25] is another pseudo-application: probably one needs the Euler-tour technique for solving this problem, but where do we find expressions that are so big that they do not fit internally, or that we would like to evaluate in parallel?

A true application, which is important in its own right, which appears to not have alternative easier solutions, and which one would really like to solve for very big problem sizes, is found in the lowest-common ancestor, *LCA*, problem. The LCA problem has wide applications. A recent and outstandingly important application of the problem is for performing queries on philogenetic trees in computational biology [34]. The LCA problem is to preprocess the entries of a tree such that afterwards, for any pair of nodes (i, j), their lowest-common ancestor $LCA(i, j)$ can be computed in constant sequential time. Such a preprocessing pays off if one has to answer many of these queries, which appears to be the case for philogenetic trees. Clearly the amount of data in this application may be overwhelming, and thus there is a natural need for solving the LCA problem in parallel or in external memory. In a parallel context one may wonder why one cannot use the parallel computer for the later queries (though it may not be available all the time). But, in an external context, the goal is highly desirable: after preprocessing, the later queries can be performed with three accesses to the external memory, whereas searching through a tree requires at least some logarithmic number of accesses.

The LCA problem has been considered by several authors [21, 33, 5]. The algorithm of Berkman and Vishkin [5] is really simple and easy to implement. In the first stage of this algorithm, one has to compute an "Euler array" and the depth of every node. In the second stage one has to solve a range-minima problem (a complete description is given in [24]). The range minima problem can be solved by computing prefix- and suffix-minima, well-structured problems that can be solved efficiently by parallel computers and in external memory. So, the total time for the LCA problem is, to a large extent, determined by the time for computing the Euler array, which boils down to solving a list-ranking problem on an Euler tour of the tree. The depths can be computed by keeping track of some additional information.

1.2 Previous Results

PRAMs. On PRAMs, the basic approach is 'pointer jumping' [41]. This technique can be used in a list-ranking algorithm which runs in $\mathcal{O}(\log N)$ time with $\mathcal{O}(N \cdot \log N)$ work on an 'EREW' PRAM. Using "accelerated cascading", the work of this algorithm is reduced to the optimal $\mathcal{O}(N)$, while maintaining running time $\mathcal{O}(\log N)$ [9]. These improved algorithms start by repeatedly selecting an "independent set", which reduces the size of the graph by a constant factor in every phase. Then, if it has been reduced to $N/\log N$, pointer jumping is applied. Numerous variants of this idea have been developed. More references are given in [24]. A variant of [9] and [2] tuned towards the requirements of the BSP model has recently been given in [6].

Meshes, Hypercubes, … . On parallel computers that communicate through an interconnection network, it is hard to achieve anything worth mentioning: by its nature, list ranking is an extremely non-local problem. For hypercubes, Ryu and JáJá [32] have shown that linear speed-up can be achieved if every PU holds at least $k = P^\epsilon$ nodes. Here P is the number of PUs, and $\epsilon > 0$ a constant. Using randomization, the problem can be solved in $(2 + o(1)) \cdot k$, if $k = \omega(\log^2 P)$ [35].

The list-ranking problem on meshes has been considered in [3, 19, 35]. $\mathcal{O}(n)$ algorithms are derived for ranking a list of length $N = n^2$ on two-dimensional $n \times n$ meshes. The algorithms in [35] give the best constants. If every PU holds k nodes, it takes $17 \cdot k \cdot n + \mathcal{O}(n)$ steps beating pointer jumping for practical values of k and n. Randomly, $10 \cdot k \cdot n + \mathcal{O}(n)$ steps are sufficient [35]. For $k = \omega(1)$, near-optimal performance is achieved: $(1/2 + o(1)) \cdot k \cdot n$ steps.

Earlier Practical Results. Several recent papers report on implementations of list-ranking algorithms on parallel computers.

Experiences with algorithms based on the independent-set-removal idea are described in [23] (for the MassPar) and [37] (for the Paragon). Asymptotically these algorithms are optimal, but the involved constants are just too large to achieve really convincing results. For example, on a Paragon with 100 PUs, the maximum obtained speed-up was 14 [37].[1] The version of independent-set removal presented in this paper is better: it achieves speed-up 17 on a Paragon with $P = 100$ and $k = 10^6$.

Reid-Miller [31] describes a randomized algorithm in the spirit of [2] on a Cray T-90. A similar algorithm has been implemented on the Paragon by Sibeyn e.a. [37]. This "sparse-ruling-set" algorithm is unbeatable when either the start-up costs are (relatively) low, or when the load (the number k of nodes per PU) is extremely high: it achieves speed-up 26 on a Paragon with $P = 100$ and $k = 10^6$, for larger k the speed-up would be much higher.

In [36] algorithms are given that achieve a better trade-off between the number of required start-ups (the time for initiating the sending of a packet) and the routing volume (the total number of integers sent and received by each PU). In this way we obtain better speed-ups for practical values of k. The most original algorithm in this paper is the "one-by-one cleaning" approach, which consists of $P-1$ rounds, in which PUs only communicate in pairs. This is one of the very few algorithms which does not require an all-to-all routing at the end of every round.

External Memory Algorithms. In comparison to the numerous parallel results, there are very few results on list-ranking in the domain of external computation. Actually, we are not aware of any algorithms that go beyond simulation of PRAM algorithms. In [8] the application of the provided PRAM simulator directly to the algorithms of [1, 10] is suggested. Asymptotically this is optimal, but one cannot expect to obtain good constants with such a coarse

[1] All speed-up results are given with respect to an optimized version of the simple sequential algorithm running on a single PU of the Paragon. For problems that are so large that they do not fit internally, the results are scaled-up linearly for the sake of comparison.

approach: every single PRAM instruction requires several sorts and scans of all the involved data.

Still, with some obvious optimizations, it appears that until now this is essentially the best idea. Instead of full sorting operations, one should perform bucket sorts, and for the selection of the independent set it is better to perform the much simpler random coin tossing. In Section 3, we consider this algorithm more closely, to obtain an estimate of the number of required paging operations.

1.3 New Results

The algorithms of this paper are based on a further development of some of the ideas from [36], most notably from "repeated-halving". However, different from [36], in our new algorithm we repeatedly perform a reduction step as in independent-set removal. This novel and highly efficient way to split off a parametrizable fraction of the nodes is common to the parallel and the external algorithm, but in the details they are quite different.

The parallel algorithm uses pointer jumping for chasing down sets of lists with small expected length. This goes very fast, because the number of participating nodes decreases rapidly. For the final subproblem we use one-by-one cleaning from [36], which, for general problems, is far more efficient than pointer jumping. If d is the number of performed reduction rounds, then the algorithm requires $12 \cdot d$ all-to-all routing operations. If the reduction factors are appropriately chosen, then the resulting routing volume can be bounded to $6 \cdot (1 + \ln P/d) \cdot k$. Thus, we establish a trade-off similar to that in [36]. On a Paragon with P PUs we obtain speed-up up around $P/3$. For large P the speed-up is somewhat smaller due to the start-up losses: On a Paragon with $P = 100$ and $k = 10^6$ the speed-up is 25.

The parallel algorithm is interesting due to its approach and simplicity, but does not really give an improvement over existing results. Our external algorithm is more of a breakthrough. In this algorithm, the reduction factors are chosen such that in every iteration a chunk of the size of the memory is split-off. Operations on this chunk are internal. The whole algorithm requires three passes over the input, going back-and-forth in a wave-like fashion. "Questions" and "answers" are pushed on stacks, and popped as soon as the next wave comes by. Our analysis shows that the total *paging volume* (the number of integers that have to brought from the hard-disc into the main-memory) is less than $18 \cdot N$. Our version of independent-set removal in Section 3 has paging volume $72 \cdot N$, and it appears that this cannot be improved much further. To be complete, we mention an experimental result (though we think it may be hard to judge its value): For $N = 64 \cdot 2^{20}$, our algorithm requires 5740s. This was achieved on a 175Mhz UltraSparc, whose hard-disc requires about 11ms for reading an 8KB page and 16ms for writing one. Internally the simple sequential algorithm runs 45 times faster, and externally it runs 300 times slower.

All results are given for randomly arranged lists. In the parallel algorithm, a random distribution is essential for balancing the work that the PUs have to do in each round, and for balancing the size of the packets during the communication rounds. With a bad distribution of the nodes, the total work of the algorithm also increases, but only by a small factor: for the external algorithm, $21 \cdot N$ is a worst-case bound on the paging volume. The difference with the average-case bound is so small, that it does not pay off to first randomize the input.

A further strong point of the algorithms is that they require only little additional space: our implementation of the parallel algorithm requires storage for $3.6 \cdot N$ integers, the external algorithm for $5.4 \cdot N$ integers This is hardly more than the sequential algorithm which requires $3 \cdot N$ (and much less than that required by independent-set removal).

Finally, the same algorithms can be used for finding the roots and depths of the nodes in a set of trees. We will commonly refer to this task by *tree rooting*. Most other techniques become inefficient or break-down. The major exception is pointer jumping, but this technique is very inefficient in itself when the trees are not shallow. Thus our algorithms allow the computation of some basic tree functions without applying the Euler-tour technique, saving the involved overhead (more than a factor two).

2 Preliminaries

Problem Definition. The input is a set of lists or trees of total length N. Every node has a pointer to a successor. The final nodes can be recognized by the distinguished value of their successor field. The lists are given by the array of their successor values, *succ*. The output consists of two arrays, *mast* and *dest*. Here, for every $0 \leq j < N$, $\textit{mast}[j]$ should give the index of the final node of the list or tree to which j belongs, and $\textit{dest}[j]$ should give the number of links between j and $\textit{mast}[j]$. In our parallel algorithms, the number of PUs is P, and every PU holds exactly $k = N/P$ nodes of the lists. PU_i, $0 \leq i < P$, holds the nodes with indices $j \cdot P + i$, for all $0 \leq j < k$. Unless indicated otherwise, we assume that the nodes are indexed in a random way. Notice that we do *not* assume that the lists or the trees are random, the assumption is made only for the indexing.

Cost Model. Except for a PRAM section, we will express the quality of our parallel algorithm by giving its *routing volume*, the number of integers sent and received by a PU, and the number of all-to-all routing operations.[2] Both these notions are well-defined, and can be determined precisely. Actually, the time for the internal work may be more important. For example, on the Paragon, which has a powerful network, the communication time may account for less than 10% of the total time consumption. But, in all list-ranking algorithms the internal work is proportional to the routing volume, and it is hard to give a definition of the internal work that is meaningful up to the constants. In the

[2] An *all-to-all routing* is a communication pattern in which every PU has to send some packets to all other PUs. It is the typical pattern that arises when a shared-memory algorithm is run on a distributed-memory machine.

particular case of list ranking, our cost measure has proven to be a fairly reliable instrument for predicting the practical behavior of algorithms [36]. Our cost model can be viewed as a simplification of BSP or BSP* [40, 28, 4].

The quality of the external algorithms is measured by determining their *paging volume*, the number of integers that have to be brought from the hard-disc into the main-memory. Actually, we are slightly more precise, by distinguishing between pages from which data are only read, and those on which data are (also) written. In general, one should also take into account the internal work of external algorithms, but in the case of list ranking, where the work is linear in the paging volume, it is fully justified to neglect it (only about 2% of the time consumption of our external algorithm is due to internal work).

Basic Assumptions. In the analysis of our parallel algorithm, we will mostly assume that N is much larger than P. For all-to-all communication patterns, it is even important that k is considerably larger than P, so that the start-up costs for sending packets can be amortized.

In the context of the external algorithms, we denote the memory size by M, and the page size by B (both given in integers). Furthermore, it is convenient to define

$$\begin{aligned} P &= 2 \cdot c \cdot N/M, \\ k &= N/P. \end{aligned}$$

Here c is some small constant, later on we will take $c = 3$. If an external algorithm is obtained as a simulation of a parallel algorithm, then P corresponds to the number of PUs in the parallel algorithm [38, 14]. Throughout this paper we assume, that

$$P \cdot B < M/2. \tag{1}$$

This implies that the main memory is large enough to accommodate c integers for the k input elements the algorithm is currently working on, plus one page for every "PU". These pages contain "messages" from the other PUs, or are used to write away messages to other PUs. It also means that bucket sort with P buckets can be performed with one scan through the data and linear work.

Theoretically, (1) is a limitation, but practically it is not: nowadays even a small PC has $M = 4 \cdot 10^6$, and a typical value for B is 2000 or smaller. So, (1) means that we should not try to handle problems that involve more than $1000 \cdot c \cdot k = 500 \cdot M$ data. Considering that currently RAM costs less than 50 times as much as hard-disc storage, such a system would be very unbalanced.

Probability Theory. In addition to some well-known results we will need

Lemma 1 (Azuma Inequality) [29] *Let $X_1, \ldots, X_m$ be independent random variables. For each i, X_i takes values in a set A_i. Let $f : \prod_i A_i \rightarrow \mathbb{R}$ be a measurable function satisfying $|f(x) - f(y)| \leq c$, when x and y differ only in a single coordinate. Let Z be the random variable $f(X_1, \ldots, X_m)$. Then for any $h > 0$,*

$$P[|Z - E[Z]| \geq h] \leq 2 \cdot e^{-2 \cdot h^2/(c^2 \cdot m)}.$$

3 Independent-Set Removal

We describe the best version of independent-set removal we can think of. As a parallel algorithm this version may be almost competitive with other parallel list ranking algorithms. However, in the external algorithm, it is essential that the active nodes stand in a compact interval of the memory at all times. This rearrangement requires considerable extra work and some additional data structures. This makes the already rather weak performance even worse.

3.1 Parallel Algorithm

In the independent-set-removal algorithm, reductions are repeated until the problem size has become sufficiently small to terminate with some other algorithm. Then the excluded nodes are reinserted in reverse orde. At all times, there is a set of active nodes. Initially all non-final nodes are active. In Phase t of the reduction we perform

Algorithm REDUCTION(t)

1. Each active node chooses independently a 0 or a 1 with probability 1/2. Each node p that has chosen a 1 sends a packet to *mast*(p).

2. If a node p which selected a 0 receives a packet, then it is inserted in the list of nodes that were excluded during Phase t, and is excluded from the list of active nodes. It sends *mast*(p) and *dest*(p) back to the sending node. Otherwise p sends back the number -1, to indicate that it was not excluded.

3. If an active node p receives -1, then it does nothing. Otherwise it uses the received data to update *mast*(p) and *dest*(p).

Every phase reduces the problem size to about $3/4$. The reinsertion is even simpler. Here we assume, by induction, that for all nodes p that were still active during the corresponding reduction phase, *mast*(p) gives the index of the last node of the list and *dest*(p) the distance thereto.

Algorithm REINSERTION(t)

1. Each node that was excluded during Phase t sends a packet to its master.

2. Each node p that received a packet sends back *mast*(p) and *dest*(p).

3. Each node p that was excluded during Phase t uses the received data to update *mast*(p) and *dest*(p).

Lemma 2 *A parallel implementation of the independent-set-removal algorithm has routing volume $(8 + o(1)) \cdot k$. Each round requires 4 all-to-all routings.*

Proof: In Step 1 of REDUCTION, $1/2$ of the nodes sends a packet of size 1. In Step 2, $1/4$ sends a packet of size 1, and $1/4$ of size 2. In Step 1 of REINSERTION, $1/4$ of the nodes sends a packet of size 1. In Step 2, $1/4$ sends a packet of size 2. Together, this gives a volume of $2 \cdot k$ for the first phase. Multiplying by 4 for the later phases, we obtain $8 \cdot k$. $\square$

$\log_2 k$	$P = 4$	$P = 16$	$P = 64$
10	0.03	0.02	0.01
12	0.09	0.06	0.04
14	0.18	0.13	0.09
16	0.22	0.19	0.16
18	0.22	0.20	0.17
20	0.21	0.20	0.18

Table 1: Measured efficiencies of independent-set removal, running on an Intel Paragon for various numbers of PUs, and various values of k. In all cases we performed ten reduction phases.

About 5 reduction phases are needed for reducing the problem size by a factor 4 (because $0.75^5 = 0.24$), thus for a given reduction, the number of all-to-all routings is of the same order as in four-reduction from [36], and the routing volume is only slightly larger.

3.2 Experimental Results

In Table 1, we provide a few examples of measured values of the efficiency of the algorithm, where by *efficiency* we mean *speed-up*$/P = T_{\text{seq}}/(P \cdot T_{\text{par}})$. As a basis for the computation of our efficiencies, we assumed that T_{seq}, the sequential time, equals $3.9 \cdot 10^{-6} \cdot N$, for all N.

A plot of the speed-ups is given in Figure 1. The given algorithm is much better than the one that was described in [37], but still cannot compete with four-reduction. The algorithm of Section 6 also performs better.

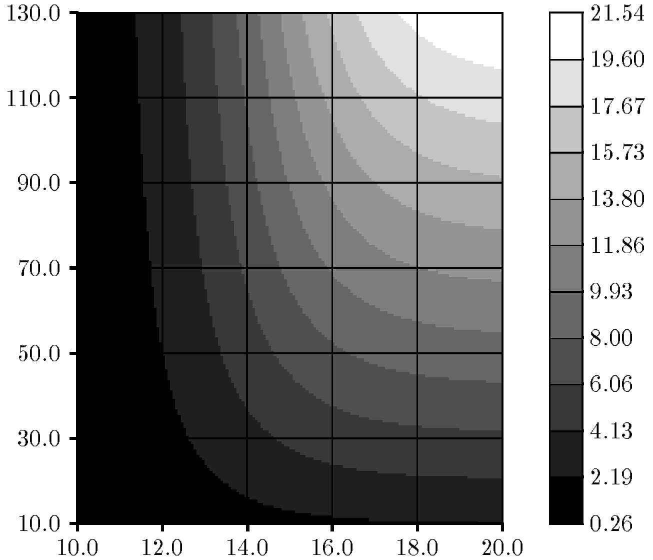

Figure 1: Experimental results for independent-set removal: the x-axis gives $\log k$, the y-axis P, and the gray-tones the speed-up.

3.3 External Algorithm

In the parallel algorithm, we maintained a list of active nodes. Hereby, we did not need to rearrange the remaining active nodes after each application of REDUCTION and REINSERTION. The fact that the active nodes are standing spread-out leads to poor cache behavior (as opposed to the algorithm presented in this paper!), but is still preferable to a rearrangement.

In an external algorithm, we have no choice: the active nodes *must* be rearranged. Unfortunately, this has unpleasant consequences: either the nodes must be renumbered, or it becomes non-trivial to find the data of the nodes (initially, the data related to Node j were stored in position j of the respective arrays). Renumbering is a lot of work. An alternative is to apply a variant of hashing that has a collision-handling strategy that guarantees that data are not stored too far away from the expected position.

In the following we do not give a detailed description of the algorithm, but rather an optimistic estimate of its paging volume. Particularly, we assume that the hashing does not require any slack, which is obviously not true. Furthermore, we neglect all slack that is required in order to accommodate data structures of randomly fluctuating sizes.

Lemma 3 *An application of* REDUCE *together with the required rearrangement on a set of lists with a total of N nodes, requires a paging volume of about* $9.5 \cdot N$.

Proof: Step 1, 2 and 3 can be performed with two passes through the data. In each pass, the *mast* and *dest* fields are read and written, and in addition, the "packets" must be written and read once. This gives a paging volume of $(2 \cdot 2 + 0.5 \cdot 2 + 0.25 \cdot 4 + 0.25 \cdot 2) \cdot N = 6.5 \cdot N$. The rearrangement implies that all data must be read and written at least once. Together, this means that $4 \cdot N$ numbers must be read and written altogether. Furthermore, the prior arrangement must be recorded, which requires additional N writing operations. However, by overlapping this operation with the second pass through the data, the reading can be saved. □

Lemma 4 *The reversal of the operations in Lemma 3 by applying* REINSERT *and a restoration operation, requires a paging volume of about* $8.5 \cdot N$.

Proof: For the restoration, $3 \cdot N$ integers must be read and $2 \cdot N$ written. These operations can be overlapped with Step 1 and part of Step 2 of REINSERT. But, a second pass through the data is needed to complete Step 2 and for Step 3. This adds a volume of $(2 + 0.25 \cdot 2 + 0.25 \cdot 4) \cdot N$. □

Theorem 1 *A complete list-ranking algorithm based on independent-set removal along the sketched lines requires a paging volume of more than* $72 \cdot N$.

Proof: After every round, the problem-size is reduced by a factor 0.75. Thus, the total paging volume is $4 \cdot (9.5 + 8.5) \cdot N$. □

Possibly, one might slightly further optimize the implementation of the algorithm, but quite surely one will not come below $60 \cdot N$: just the rearrangement and the restoration, which appear inevitable, require $40 \cdot N$.

4 Power of Autoclean and Altroclean

The basic idea of our algorithm is to split the input into two sets: $\mathcal{S}_0$ and $\mathcal{S}_1$. Then we perform

Algorithm PEELING_OFF($\mathcal{S}_0$, $\mathcal{S}_1$)

1. AUTOCLEAN($\mathcal{S}_1$);
2. ALTROCLEAN($\mathcal{S}_0$);
3. SOME_RANK($\mathcal{S}_0$);
4. ALTROCLEAN($\mathcal{S}_1$).

Here SOME_RANK designates any ranking algorithm, possibly PEELING_OFF itself. By AUTOCLEAN($\mathcal{S}_j$) we mean:

> All nodes in $\mathcal{S}_j$ follow the links running through nodes in $\mathcal{S}_j$ until a link out-off $\mathcal{S}_j$ is found or a final node of the list is reached. Then they update *mast* and *dest*.

By ALTROCLEAN($\mathcal{S}_j$) we mean:

> All nodes in $\mathcal{S}_j$ that have not reached a final node and whose master is not an element of $\mathcal{S}_j$, ask their master for its *mast* and *dest* fields. Then they update their *mast* and *dest* fields with the received values.

Later we will give efficient algorithms for performing auto- and altroclean. For the time being, we assume that they are performed according to the above specifications. If initially

$$\begin{aligned} \textit{mast}(j) &= \textit{succ}(j), \\ \textit{dest}(j) &= 1, \end{aligned}$$

for all $0 \leq j < N$, then we get

Lemma 5 PEELING_OFF *correctly computes the values of mast and dest for all nodes.*

Proof: After Step 1, every node in $\mathcal{S}_1$ has either found a master in S_0 or reached a final node. Hence, after Step 2, all nodes in $\mathcal{S}_0$ have either a master in $\mathcal{S}_0$ itself or reached a final node. So, in Step 3 we indeed have to solve an ordinary weighted list-ranking problem. In the altroclean of Step 4, all nodes of $\mathcal{S}_1$ that have not reached a final node participate. They ask their masters for their *mast* values, and the answer is some final node. □

The time consumption of PELLING_OFF is given by:

$$\begin{aligned} T_{\text{peeling\_off}}(\mathcal{S}_0 \cup \mathcal{S}_1) &= T_{\text{some\_rank}}(\mathcal{S}_0) + T_{\text{autoclean}}(\mathcal{S}_1) \\ &+ T_{\text{altroclean}}(\mathcal{S}_0) + T_{\text{altroclean}}(\mathcal{S}_1) \end{aligned}$$

The two altrocleans in Step 2 and Step 4 are more or less the same, though the fraction of participating nodes is larger in Step 4. If the autoclean were as hard as an arbitrary ranking problem, then going on recursively, would give a logarithmic factor in the overall time consumption. Fortunately, this is not the case.

For the performance of the parallel algorithm, it is crucial that in the autoclean in Step 1 a fraction $\alpha = \#\mathcal{S}_1/N$ of the nodes plays the role of a terminal node (all those with a master in $\mathcal{S}_0$). That is, we have lists of expected length $1/\alpha$. Such a list-ranking problem is substantially easier than a general one. For example, if we apply pointer jumping, then the number of participating nodes in Round t is only $(1-\alpha)^{2^t}$.

Along these lines one can also obtain an efficient external algorithm. It appears, however, to be even more efficient to choose $\mathcal{S}_1$ such that $\#\mathcal{S}_1 = M/6$. Then, Step 1 is a simple internal operation. The details are discussed in Section 7.

5 PRAMs

5.1 List Ranking Algorithm

We show that along the lines of PEELING_OFF there is an easy randomized PRAM algorithm running in $\mathcal{O}(\log N)$ time with $N/\log N$ PUs.

First one should perform some randomization: every node is placed in a randomly chosen bucket of size $N/\log N$. With P PUs this can be done in $\mathcal{O}(N/P + \log N)$ time. Using Chernoff bounds [7, 20], it is easy to see that no bucket will hold more than $(1+o(1)) \cdot N/\log N$ nodes. The buckets are numbered from 0 through $\log N - 1$, and the set of nodes in Bucket j is denoted $\textit{Buc}_j$.

Then we perform $\log\log N$ rounds of PEELING_OFF in which the problem size is halved each time. In Round t, $1 \leq t \leq \log\log N$, we take

$$\begin{aligned} \mathcal{S}_0(t) &= \cup_{j=0}^{\log N/2^t - 1} \textit{Buc}_j, \\ \mathcal{S}_1(t) &= \cup_{j=\log N/2^t}^{\log N/2^{t-1} - 1} \textit{Buc}_j. \end{aligned}$$

Finally we perform pointer jumping on $\mathcal{S}_0(\log\log N) = \textit{Buc}_0$. An iterative formulation of the given recursive algorithm may be easier to understand:

Algorithm PRAM_RANK
for $t = 1$ **to** $\log\log N$
  AUTOCLEAN($\mathcal{S}_1(t)$);
  ALTROCLEAN($\mathcal{S}_0(t)$);
POINTER_JUMPING($\mathcal{S}_0(\log\log N)$);
for $t = \log\log N$ **downto** 1
  ALTROCLEAN($\mathcal{S}_1(t)$).

The correctness of PRAM_RANK follows from Lemma 5 and the fact that $\mathcal{S}_0(t+1) \cup \mathcal{S}_1(t+1) = \mathcal{S}_0(t)$, for all $1 \leq t < \log\log N$.

On a PRAM, the altrocleans are trivial: for every node two numbers must be read. So, the work is linear in the size of the set on which it is performed. As $\sum_{t=1}^{\log\log N} \mathcal{S}_0(t) =$

$N/2$, and $\sum_{t=1}^{\log\log N} \mathcal{S}_1(t) = N - N/\log N$, the total work is linear. The processor allocation is no problem.

The final pointer jumping has to be performed on a set of size $N/\log N$. With P PUs, this can be done in $\mathcal{O}(N/P + \log N)$ time (see [24]).

The autocleans we perform by applying the basic pointer jumping step (every node which has a master in $S_1(t)$ asks its master for its *mast* and *dest* values) until no nodes are active anymore. Their time consumption is analyzed in Lemma 7.

A practical way of handling a final node f is, that it, when asked for its master, passes back the value $-$*index*, where *index* is the index of f: as negative values automatically lie outside the range of other values, there is no further need to single-out nodes that point to f.

5.2 Analysis

In a PRAM algorithm, in which we do not care much about the leading constant in the time consumption, we could have randomized the nodes we are working on at the beginning of every iteration, which would make it obvious that the probability that the master of a given node in $\mathcal{S}_1$ lies in $\mathcal{S}_0$ is $1/2$ and independent of any other such event. But, we will prove that this is even the case with just the single initial randomization. For this we need several results. At this point they may appear to be of little importance, but they give insight into the operation of the algorithm, and we will reuse them in our analysis of the other algorithms.

Lemma 6 (Hangglider Lemma) *At the end of Iteration t, $0 \leq t \leq \log\log N$, an arbitrary node $n \in \mathcal{S}_0(t)$ has as master the final node f of its list iff all nodes between n and f lie in $N - \mathcal{S}_0(t)$. If this is not the case, then its master is the first node $m \in \mathcal{S}_0(t)$ such that all nodes between n and m lie in $N - \mathcal{S}_0(t)$.*

Proof: By the way final nodes are handled, the first case can be viewed as a special case of the second, So, we may concentrate on a node with master m, m not being a final node. Clearly $m \in \mathcal{S}_0(t)$, due to the autocleaning and altrocleaning during Iteration t.

For the rest of the proof we proceed by induction on t. Let m be the above defined node. So far, n has not asked any node in $\mathcal{S}_0(t)$ to give its master. Thus, either *mast*(n) = *succ*(n), for which the lemma holds, or n must have heard *mast*(n) from some node $n' \in N - \mathcal{S}_0(t)$. n' may have updated its master in two ways: during some previous altroclean and during an autoclean. By the induction hypothesis, n' can certainly not have "looked" beyond m during an altroclean. During the autoclean this is excluded altogether. □

This lemma states that after Iteration t, the linking structure in $\mathcal{S}_0(t)$ is identical to the initial one, except for short-cuts over $N - \mathcal{S}_0(t)$. From this we conclude

Corollary 1 *At the beginning of Iteration t, the probability that any node from $\mathcal{S}_0(t) \cup \mathcal{S}_1(t)$, which has not reached a final node, has its master in $\mathcal{S}_0(t)$ is $1/2$.*

Corollary 2 *At the beginning of Iteration t, no two nodes from $\mathcal{S}_0(t) \cup \mathcal{S}_1(t)$ have the same master, except for those whose master is a final node.*

Lemma 7 *On a PRAM with P PUs,* AUTOCLEAN*($\mathcal{S}_1(t)$), $1 \leq t \leq \log\log N$, can be implemented to run in $\mathcal{O}(\#\mathcal{S}_1(t)/P + \log\log N)$ time, with high probability.*

Proof: A node n in $\mathcal{S}_0$ is active in Iteration $s \geq 0$, if n and all nodes up to distance 2^s from s have a master in $\mathcal{S}_1$. The probability that any given node lies in $\mathcal{S}_1$ is $1/2$, so the probability that n is active in Round s equals 2^{-2^s}. Hence, the expected number of nodes that is active in Round s equals $2^{-2^s} \cdot \#\mathcal{S}_1$. So, for the expected number of the sum of active nodes over all rounds we find

$$E(\mathit{actives}(t)) = \sum_{s \geq 0} 2^{-2^s} \cdot \#\mathcal{S}_1(t) < 0.82 \cdot \#\mathcal{S}_1(t). \quad (2)$$

Here we make use of the fact that expected numbers can be computed by simply adding probabilities, and that the expected number of a sum equals the sum of the expected numbers, even if the random variables are not independent (see [16, p. 222]).

Now anyone will also believe that, with high probability, the work is bounded by $\mathcal{O}(\#\mathcal{S}_1(t))$, but this is not so easy to prove. We will first put a bound on the number of rounds that has to be performed, then ensure that during the first rounds the number of active nodes decreases as it should do, and then argue that the contribution from the remaining nodes is minor.

Claim 1 *After $\mathcal{O}(\log\log N)$ rounds there are no active nodes left, with high probability.*

Proof of claim: The probability that any of the $\#\mathcal{S}_1(t)$ nodes is active in Round s is at most $\#\mathcal{S}_1(t) \cdot 2^{-2^s}$. For $s = 2 \cdot \log\log N$, this is less than $1/N$.

For bounding the number of active nodes during the first rounds, we use the Azuma inequality, Lemma 1. Here the X_j are the random variables given by: "$X_j = 1$ if Node j lies in $\mathcal{S}_1(t)$, otherwise it is 0." The function f gives the number of active nodes in Round s. Flipping the value of one of the X_j may change the value of f by at most 2^s, so $c = 2^s$. Thus, substitution yields

$$P[|Z - E(Z)| \geq h] \leq 2 \cdot e^{-2 \cdot h^2/(2^{2 \cdot s} \cdot N)}.$$

For $h = 2^s \cdot (N \cdot \log N)^{1/2}$, this is a very small number. Thus, as long as $2^s \cdot (N \cdot \log N)^{1/2} = o(2^{-2^s} \cdot \#\mathcal{S}_1(t))$, we may assume that the deviations from the expected values are negligible. So, we may assume that after $\log\log N^{1/3}$ rounds only $(1 + o(1)) \cdot \#\mathcal{S}_1(t)/N^{1/3}$ active nodes remain. Therefore, the sum over the remaining rounds of the number of active nodes is less than $(2 \cdot \log\log N) \cdot (1 + o(1)) \cdot \#\mathcal{S}_1(t)/N^{1/3} = o(\#\mathcal{S}_1(t))$. □

Combining the above results, we find

Theorem 2 *On a PRAM with P PUs,* PRAM_RANK *ranks a set of list with N nodes in $\mathcal{O}(N/P + \log N)$ time, with high probability.*

5.3 Tree Rooting

For tree rooting we apply the same algorithm. The only difference is that now several nodes may have the same successor. Lemma 6, the conclusion about the structure and Corollary 1 still hold, and we have the following partial analogue of Theorem 2:

Theorem 3 *On a PRAM with P PUs,* PRAM_RANK *roots a set of trees with N nodes in $\mathcal{O}(N/P + \log N)$ expected time.*

Proof: The crucial point is again that expected values may be added together, even if their random variables are not independent. □

The good news is that Theorem 3 holds for *all* trees (as opposed to pointer-jumping, whose expected time consumption depends on the structure of the tree). The bad news is that we cannot put a high-probability bound on the time consumption. Consider a *tailed star*: a tree which is obtained by attaching a tail to a star, see Figure 2. Assume

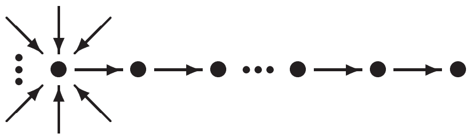

Figure 2: A tailed star.

that the star contains $N - \log N/\log\log N$ nodes, and the tail $\log N/\log\log N$. Then it cannot be exluded that all the nodes of the tail are allocated to the $\mathcal{S}_1(1)$. But then AUTOCLEAN($\mathcal{S}_1(1)$) will require $\Omega(N \cdot \log N)$ work. Similar problems arise with the altrocleans in the first for-loop of PRAM_RANK.

A possible solution would be to never answer more than one question at a time, and to tell the others that later they should ask here again. However, it appears that in this way it may happen that only a very small subproblem is solved, and that we have not gained much. As the Euler-tour technique gives a reduction of the tree-rooting problem to list ranking with a small constant factor loss, it is not worth the effort to develop some elaborate algorithm for it.

6 Distributed Memory Machines

6.1 List Ranking Algorithm

To a large extent, our algorithm for distributed memory machines is a direct simulation of the PRAM algorithm. Now PU_i, holds the $k = N/P$ nodes with indices $i + j \cdot P$, for all $0 \leq j < k$. Each PU has a buffer for every PU, in which it writes questions and answers. In any step, all questions or answers are generated, then the all-to-all routing is performed and so on. This is the standard way of running algorithms under the BSP paradigm, and is straightforward. This immediately leads to an algorithm with acceptable performance. To optimize the algorithm, one has to be slightly more careful.

For the final stage, we should not perform pointer jumping, but rather one-by-one cleaning from [36]. This routine is so much more efficient that we can stop with a larger subproblem:

Lemma 8 [36] *For ranking a set of list with a total of $k \cdot P$ nodes on a parallel computer with P PUs, one-by-one cleaning requires $3 \cdot P - 3$ start-ups, and has routing volume $6 \cdot \ln P$.*

Theoretically the most interesting feature is the choice of the number of reduction rounds d (in the PRAM algorithm we had $d = \log\log N$) and the *reduction factors*: the number f_t, $1 \leq t \leq d$, given by

$$f_t = \#\mathcal{S}_1(t)/\#\mathcal{S}_1(t-1),$$

where $\#\mathcal{S}_1(0) = N$. These must be tuned to obtain a good trade-off between the two components of our cost measure: the number of all-to-all routings and the routing volume. In any case the f_t must be chosen according to the following guide-lines.

- The f_t should decrease with t.
- f_1 should increase with d.
- f_d should not be too small.

One such choice is

$$f_t(d) = \frac{1 + d - t}{2 + d - t}. \tag{3}$$

With these f_t, we get a simple expression:

$$\#\mathcal{S}_0(t) = (d + 1 - t)/(d + 1) \cdot N.$$

There are better choices (see Section 6.2 for the choice of our implementation), but this choice facilitates the analysis of the routing volume:

Theorem 4 *When d reduction phases are performed with reduction factors as in (3), the routing volume is less than*

$$(6 + (3 \cdot \ln d + 6 \cdot \ln P)/(d + 1)) \cdot k,$$

with high probability.

Proof: We are going to compute the number of questions. For every question two answers will be sent, so the routing volume is three times the number of questions.

During the altroclean in Iteration t of the first loop, the expected number of questions equals $\#\mathcal{S}_0(t) \cdot \#\mathcal{S}_1(t)/(\#\mathcal{S}_0(t) + \#\mathcal{S}_1(t)) < \#\mathcal{S}_1(t)$. Summing over all rounds, we get less than $(1 - 1/(d-1)) \cdot k$. The same estimate holds for the altrocleans in the second loop.

Generally, if one performs an autoclean on a set $\mathcal{S}$, in which the probability that a node has master in $\mathcal{S}$ is α, then we find the following analogue of (2) for the total number of questions asked:

$$E(\mathit{questions}) = \sum_{s \geq 0} \alpha^{-2^s} \cdot \#\mathcal{S}.$$

In our case, α assumes the values $1/2, 1/3, \ldots, 1/(d+1)$. The computation of the sum is easy because $\#\mathcal{S}_1(t) = N/(d+1)$, for all t. The first-order term, $1/2 + 1/3 + \cdots + 1/(d+1) < \ln d - 1/4$, for $d \geq 10$. The quadratic terms are equal to those neglected in the estimate of the volume of altroclean, and all the remaining terms together are less than $1/4$.

The final one-by-one cleaning is performed on a set of size $k \cdot P/(d+1)$. □

If all-to-all routings are performed in the most straightforward way, then the $3 \cdot P - 3$ start-ups from one-by-one cleaning correspond to three all-to-all routings. Using this estimate, it is easy to express the number of all-to-all routings as a function of the number of reduction rounds:

Theorem 5 *When d reduction phases are performed, the algorithm requires $(6 + 2 \cdot \lceil \log\log n \rceil) \cdot d + 3$ all-to-all routings.*

Proof: For every reduction round we have to perform two altrocleans, each taking two all-to-all routings, and one autoclean. As $f_t(d) \geq 1/2$, the probability that the distance between two elements in $\mathcal{S}_0$ exceeds r is at most 2^{-r-1}. Thus, the probability that the pointer-jumping has not terminated after s steps, requiring two all-to-all routings each, equals 2^{-2^s-1}. For $s = \lceil \log\log n \rceil + 1$, this is less than $1/n^2$. □

6.2 Experimental Results

During the autocleans, we have to perform several rounds of pointer jumping. In order to prevent having to redetermine the active nodes every time, these should be selected from the former active nodes once they get to know their new masters.

In order to save buffer space, it turns out to be better not to send all the questions at one time: for every question two answers are generated, and thus we would need twice as much buffer space as questions being generated. We need only a fraction $1/P$ additional buffer space, if all PUs first send their questions to the PU with index one larger (cyclically), then return the answers to the received questions, and then repeat this for the questions to PUs with index two larger, and so on.

To obtain better performance for smaller values of k, it is essential to apply the two- and log-phase routers next to the simpler one-phase router (see [37]). In this way, at the expense of higher handling costs, the number of start-ups for an all-to-all routing can be reduced from $P - 1$ to $2 \cdot (\sqrt{P} - 1)$ or even just $\log P$.

To minimize the costs of the autocleans, one should make the first f_t somewhat larger, and then later on, when the size of the problem becomes smaller, somewhat smaller. We found good performance for

$$f_t(d) = \frac{1.6 + 1.05 \cdot d - t}{6 + d - t}.$$

$\log_2 k$	$P = 4$	$P = 16$	$P = 64$
10	0.06	0.03	0.01
12	0.19	0.07	0.03
14	0.36	0.10	0.09
16	0.47	0.29	0.18
18	0.44	0.35	0.23
20	0.41	0.35	0.26

Table 2: Measured efficiencies of the parallel algorithm running on an Intel Paragon for various numbers of PUs, and various values of k. In all cases $d = 6$.

The d for which the time consumption is minimized increases with k, and decreases with P. In all cases it lies between 4 and 16. However, beyond a certain point, the choice of d has only a minor influence. It turns out that $d = 6$ always gives results that are close to optimal. A fixed choice for d allows us to optimize the size of the buffers. For $d = 6$, the final problem is so small, $0.012 \cdot N$, that its solution takes very little time.

The number of pointer-jumping steps in the autocleaning of Round t must be chosen as a function of f_t, $\#\mathcal{S}_1(t)$ and d, in such a way that the probability that the whole algorithm is correct is constant (alternatively, one can test whether all nodes are done). In practice, we mostly need five pointer-jumping steps if $f_t < 0.4$, but for $f_t \geq 0.4$, four of these steps are generally enough.

Implementing these ideas, we obtained an algorithm which uses in every PU next to the three arrays of size k each, only two buffers which are used for several purposes. These have size $0.3 \cdot k$ each. The program, and its sequential variant (which is identical except that every variable is replaced by an array, and every instruction by a loop) are available at `http://www.mpi-sb.mpg.de/~jopsi/dprog/prog.html`.

We did extensive experiments and found that when only the one-phase router is applied, the time consumption on the Paragon can be described up to 10% by an expression of the following form:

$$T_1(P, k) = \alpha + \beta \cdot k + \gamma \cdot P + \delta \cdot k \cdot \log(P).$$

For the constants we found the following values:

$$\alpha = 38 \cdot 10^{-3}, \qquad \beta = 40 \cdot 10^{-7},$$
$$\gamma = 58 \cdot 10^{-4}, \qquad \delta = 24 \cdot 10^{-7}.$$

In Table 2, we provide a few examples of measured values of the efficiency of the algorithm. The *efficiency* is defined as in Section 3.2. From these numbers we can discern the following trends: the efficiency strongly increases with k, and slowly decreases with P. None of this is surprising. There are three reasons for the deterioration with P: the finite capacity of the network starts to become noticeable for 6×6 partitions and larger; the number of start-ups required for an all-to-all routing increases with P, which forces the algorithm to use an alternative, less efficient, router; for large P, the load-balancing becomes worse. A plot of the speed-ups is given in Figure 3.

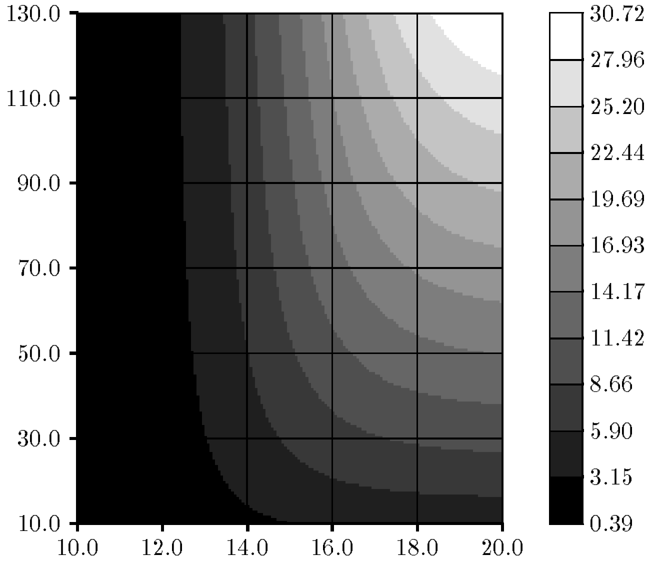

Figure 3: Experimental results for the parallel algorithm: the x-axis gives $\log k$, the y-axis P, and the gray-tones the speed-up. In all cases $d = 6$.

6.3 Tree Rooting

As for the PRAM, we may apply the same algorithm for tree rooting. The expected work is the same as for ranking lists.

In order to give a feeling of what happens when we apply the algorithm to different non-cyclic structures, we give some numbers for the special case $P = 8$ and $k = 65{,}536$. We have tested random lists, random binary trees, and stars with a tail of length 1,000. In each case we tested 20 inputs. The results are given in Table 3. These results may be taken to be representative for other values of P and k.

	min time	max time	av. time	st. dev.
lists	0.674	0.678	0.675	0.002
binary trees	0.595	0.626	0.607	0.010
tailed starts	0.692	1.772	1.070	0.280

Table 3: Time consumption in seconds for different types of input for $P = 8$ and $k = 65{,}536$.

Most apparent is the increase of the standard deviation. Also we see that in practice the average time consumptions may differ. Reasons why a problem can be solved faster than for lists are that more nodes may discover the final node at an earlier stage, and that when more nodes ask for information from the same node, this information still may be available in cache. The main reason why it may go slower than for lists is that the load-balancing may be poor.

7 External Memory Computation

7.1 Algorithm

The external algorithm is closely related to, but nevertheless not a direct translation of the parallel algorithm. If we look back at PRAM_RANK, then there are two important differences that go beyond the correct choice of the reduction factors, and the management of “packets”:

- AUTOCLEAN should not be implemented by pointer jumping, but as an internal list-ranking problem.
- In the second for-loop, one should not perform ALTROCLEAN($\mathcal{S}_1(t)$), but ALTROCLEAN($N - \mathcal{S}_0$).

High-Level Description. In the following we describe the algorithm in more detail. Let $P = 6 \cdot N/M$, and $k = N/P$. The input is subdivided in P buckets of size k each: Buc_i, $0 \leq i < P$, holding the data related to the nodes with indices j, such that $k \cdot i \leq j < k \cdot (i + 1)$. Furthermore, for all $1 \leq t \leq P$,

$$\begin{aligned} \mathcal{S}_0(t) &= \cup_{i=0}^{P-t-1} Buc_i, \\ \mathcal{S}_1(t) &= Buc_{P-t}. \end{aligned}$$

With these definitions, we get the following high-level description of the algorithm:

Algorithm EXTERNAL_RANK
for $t = 1$ **to** $P - 1$
  AUTOCLEAN($\mathcal{S}_1(t)$);
  ALTROCLEAN($\mathcal{S}_0(t)$);
AUTOCLEAN($\mathcal{S}_1(P)$);
for $t = P - 1$ **downto** 1
  ALTROCLEAN($N - \mathcal{S}_0(t)$).

The first loop is the same as before except for the size of the subsets. The second loop is also more or less the same. Only the order in which the questions are posed is slightly different. Now, all questions pending for the nodes in Buc_{P-t-1} are asked in Pass t.

Lemma 9 EXTERNAL_RANK *correctly solves a ranking problem on a set of lists or trees.*

Proof: As before, we may assume that at the end all nodes of $\mathcal{S}_1(P)$ know their ranks and the last nodes of their lists. This is the basis of our induction assumption: at the end of Pass t of the second loop, all nodes p for which $mast(p)$ lies in $\mathcal{S}_0(t)$ after the end of the first loop know their ranks and final nodes. But then, during Pass $t - 1$, all nodes with master in $Buc_{P-t} = \mathcal{S}_0(t-1) - \mathcal{S}_0(t)$ ask their questions, and get to know their ranks and final nodes. After Pass 1, all nodes with master in $\mathcal{S}_0(1)$, that is all nodes, know their ranks and final nodes. □

Design Principles. In the remainder of this section, we will talk about the nodes in Buc_i, as if they are managed by their own PU_i. In this way we can say things like “the questions to PU_i", which means "requests for data from the nodes in Buc_i."

In our implementation, we go through the data in a wave-like way: addressing PU_{i+1} after PU_i or vice-versa. In addition we apply the following principles:

- Bucketing the questions and answers.
- Lazy processing of answers.
- Use of stacks.

The *bucketing* means that all questions or answers of a given type that all PUs are sending to PU_i are pushed on a common stack. The *lazy processing* means that questions and answers destined for PU_i are not processed immediately, but only once the wave through the data hits PU_i the next time. The use of stacks implies that questions or answers destined for PU_i are popped, once they are processed. The newly generated questions and answers are pushed again, and for this the just freed space can be reused. Thus, instead of reading one block of the memory, and writing another, we may read and write the same block.

> Going in waves through the PUs, and applying the three given points, is not limited to list ranking, but constitutes a good set of design principles for any external algorithm.

Details of Implementation. The algorithm consists of three waves: the first starting at PU_0, the second starting at PU_{P-1} and the third starting at PU_0 again. We describe each of them.

Wave 1 consists of a kind of bucket-sort: for each Node j in PU_i, which has $succ(j) = j'$ in some $PU_{i'}$, with $i' < i$, $(j', j, 1)$ is pushed on the stack of 1-questions to be asked to $PU_{i'}$ during Wave 2. During Wave 1 the order in which the PUs are addressed is not so important, but it is profitable to end in PU_{P-1}, because PU_{P-1} is the starting point of Wave 2.

Wave 2 corresponds to the first for-loop from EXTERNAL_RANK. The PUs are addressed in decreasing order, starting with PU_{P-1} and ending with PU_0. For each PU_i, first all answers to the 1-questions are processed and popped from their stacks. Hereafter, the autoclean can be performed. If after this, Node j has $mast(j) = j'$, with j' held by $PU_{i'}$, then (j', j) is pushed on the stack of 2-questions to be asked to $PU_{i'}$ during Wave 3. Now all 1-questions to PU_i can also be answered. Here we must be careful: some of the questions are sent back to the asking PUs, others are forwarded. Consider a question by Node j' in $PU_{i'}$ to Node j in PU_i, and let $j'' = mast(j)$ lie in $PU_{i''}$. If $i'' \leq i'$, then $(j', j'', value + dest(j))$ is pushed on the stack of answers of $PU_{i'}$. Here *value* indicates the former value of this field. On the other hand, if $i'' > i'$, then in the original algorithm, $PU_{i'}$ would ask a second question during a later altrocleaning. So, on behalf of Node j', PU_i pushes an updated question $(j'', j', value + dest(j))$ on the stack of 1-questions of $PU_{i''}$.

Wave 3 corresponds to the second for-loop from EXTERNAL_RANK. The PUs are addressed in increasing order, starting with PU_0 and ending with PU_{P-1}. For each PU_i, first all answers to the 2-questions are processed and popped from their stacks. Hereafter, the 2-questions to PU_i can be answered: for a question (j, j') asked by $PU_{i'}$, $(j', mast(j), dest(j))$ is pushed on the stack of answers of $PU_{i'}$.

7.2 Analysis

The correctness of the algorithm is guaranteed because it only means a non-essential rescheduling of the order of operations. So, it remains to analyze the time consumption and the paging volume.

Worst-case Inputs. At first it is not clear that the algorithm has linear time consumption: during Wave 2, one question may generate a whole ripple of forwarded questions. Fortunately, for lists we have an analogue of the Hangglider Lemma, Lemma 6, stating that while processing PU_i during Wave 2, no two nodes in some $PU_{i'}$, with $i' < i$, address questions to the same node in PU_i. Hence, a PU has to answer at most k 1-questions. In the following we go in more detail.

During Wave 1 all *succ* fields are paged-in. In Wave 2, all *succ* fields are paged-in again, to initialize the *mast* fields, but that is it. So, *succ* contributes $(2 \cdot P - 1) \cdot k$ to the paging volume (read only).

The *mast* and *dest* fields are updated during Wave 2 and Wave 3. So, each of them contributes $(2 \cdot P - 1) \cdot k$ to the paging volume (read and write).

In total, at most $(P - 1) \cdot k$ 1-questions are generated, each consisting of three integers. Each question must be read again, then there is at most one answer of the same size, which must be written and read once.

All nodes in PU_i, with $1 \leq i < P$ ask one 2-question consisting of two integers. Each question must be read again, and then there is one answer of size three integers, which must be written and read once.

Theorem 6 *For ranking a set of lists of total length N, EXTERNAL_RANK has a paging volume of at most $21 \cdot N - 18 \cdot k$. Of this amount, $3 \cdot N - 2 \cdot k$ is read-only paging. The algorithm needs at most $6 \cdot N$ storage space in total.*

Proof: Adding together the above numbers gives an upper limit of $28 \cdot N - 25 \cdot k$. However, this includes a certain amount of double counting, because the stacks shrink and expand in the same memory blocks. We show that by the use of stacks, we save $7 \cdot (N - k)$ in total.

During Wave 2, a PU never handles more than k answers and k 1-questions. As a result it produces at most k answers and k 2-questions. So, at most $6 \cdot k$ integers are read, of which $5 \cdot k$ are written again. During Wave 3, each 2-question is read and answered as far as possible in the same space. □

So, we see that unlike the parallel algorithm, it is no problem at all to handle worst-case inputs. A comparison with Theorem 1 shows that even for bad inputs, the algorithm has less than one third of the paging volume of an algorithm based on independent-set removal.

Randomized Inputs. If the input may be assumed to be random, or if they are randomized first, then the performance is even better. Here we give an estimate of the expected paging volume. High-probability bounds can be derived as in the proof of Theorem 4.

Theorem 7 *For ranking a set of random lists of total length N,* EXTERNAL_RANK *has an expected paging volume of less than $18 \cdot N - 10 \cdot k$. Of this amount more than $4 \cdot N$ is read-only paging. The algorithm needs at most $5.4 \cdot N$ storage space in total.*

Proof: The paging volume due to *succ*, *mast* and *dest*, is the same as before: $6 \cdot N - 3 \cdot k$, of which $2 \cdot N - k$ is read-only. In the following we analyze the paging volume due to the questions and answers.

During Wave 1, the expected number of pushed 1-questions equals $\sum_{i=1}^{P-1} i \cdot k/P = (N-k)/2$. This induces a paging volume of $3/2 \cdot (N-k)$.

During Wave 2, PU_i has to read $k \cdot i/(i+1)$ 1-questions, which are answered or forwarded without causing additional paging volume. In total this contributes a paging volume of three times $\sum_{i=1}^{P-1} k \cdot i/(i+1) = N - k \cdot \sum_{i=1}^{P} 1/i \simeq N - k \cdot \ln P$.

During Wave 2, a PU also has to handle as many updates as it asked 1-questions. That is, $k \cdot (P-i-1)/P$ by PU_i. In addition, the 2-questions are pushed: k for every PU_i, with $i > 0$. Generally, we must account for these two operations a paging volume of $k \cdot \max\{3 \cdot (P-i-1)/P, 2\}$. Over all PUs this gives $2 \cdot N + 3 \cdot k \sum_{i=1}^{P/3-1} i/P = 2^1\!/_3 \cdot N - k$. Of this $N/3 + k$ is read-only.

During Wave 3, each PU_i answers as many 2-questions as were posed to it: $k \cdot g_i$, where $g_i = \sum_{j=i+1}^{P-1} 1/j$. The questions have size two, the answers size three. In addition, for $i > 0$, PU_i reads k answers. That is, the paging volume for PU_i equals $k \cdot \max\{3 \cdot g_i, 2 \cdot g_i + 3\}$. Because $g_i \simeq \ln P - \ln i$, we see that the first term dominates for $i \leq P/e^3$. Thus, using $\sum_i g_i = P - 1$,

$$\begin{aligned}\sum_{i=0}^{P-1} \max\{3 \cdot g_i, 2 \cdot g_i + 3\} &\simeq \\ 2 \cdot P - 2 + 3 \cdot P \cdot (1 - e^{-3}) + \sum_{i=0}^{P/e^3} g_i &\simeq \\ (5 + e^{-3}) \cdot P - 2.\end{aligned}$$

Here we approximated the sum over g_i by an integral, and g_i by $\ln P - \ln i$. Of this amount, $(2 + e^{-3}) \cdot N$ is read-only.

The required storage space reaches its maximum during Wave 3: after processing the first few PUs, most questions have been answered, but are not yet processed. More precisely, after PU_i is addressed, the stack has expected size of approximately

$$2 \cdot (N - k) + k \cdot \sum_{j=1}^{i} (\ln P - \ln j - 1).$$

The maximum is assumed for $i = P/e$: $2 \cdot (N-k) + N/e$. □

We see that the algorithm may indeed be expected to run faster on random inputs, but the difference with the worst-case bound is not tremendous.

7.3 Refinement

The paging volume for the randomized case can be reduced by $\Theta(k \cdot \log P)$, at the expense of a similar amount of additional internal work. In comparison to the total of $\Theta(k \cdot P)$, this is asymptotically negligible, but for $P < 100$, this improves the performance by a few percent.

After the autocleaning, all nodes have a master in a PU with higher index, and therefore the number of 2-questions equals $(P-1) \cdot k$. There is a certain waste here: it is better not to ask 2-questions that could have been answered internally. The easiest way to realize this, is to postpone autoclean. During the altrocleans of the first loop, this means that a certain number of questions cannot be answered immediately, but the necessary search is internal, and the expected depth of search is small. Instead, a PU should perform autoclean during the second loop, after processing the answers to the 2-questions, and before answering the 2-questions. In this way, the number of 2-questions is reduced by $\sum_{i=2}^{P} k/i \simeq k \cdot \ln P$.

We resume the complete external algorithm, missing details can be found in the above descriptions.

Algorithm EXTERNAL_RANK

1. **for** $i = P-1$ **downto** 1 **do** with PU_i
  - **a.** Determine all nodes j with *succ*(j) in some $PU_{i'}$ with $i' < i$. For each such node, push a question on the stack of 1-questions of $PU_{i'}$.

2. **for** $i = 0$ **to** $P-1$ **do** with PU_i
  - **a.** For all nodes j set *mast*(j) = *succ*(j) and *dist*$(j) = 1$.
  - **b.** Process all answers to 1-questions.
  - **c.** Answer all 1-questions and push an answer on the stack of answers of the PU that asked the question, or push a new question on the stack of 1-questions of some intermediate PU.
  - **d.** For every node j, with *mast*(j) in $PU_{i'}$ with $i' > i$, push a question on the stack of 2-questions of $PU_{i'}$.

3. **for** $i = P-1$ **downto** 0 **do** with PU_i
  - **a.** Process all answers to 2-questions.
  - **b.** Perform autoclean.
  - **c.** Answer all 2-questions and push an answer on the stack of answers of the PU that asked the question.

7.4 Experimental Results

The algorithm has been programmed in C. Including lengthy comments and routines for generating lists, testing and visualization the program has 462 lines. Actually all essential work is performed by less than 100 lines of code. This conciseness means that optimization efforts can be focused. The program is available at `http://www.mpi-sb.mpg.de/~jopsi/dprog/prog.html`.

An important point is how the stacks are organized. In our case, we (de-) allocated chunks of the size of a memory page to (from) one of the sub-stacks (for example to the stack of 1-questions to PU_{14}). At the end of a full page of a sub-stack, it is indicated where the stack continues. This costs only one position out of every memory page (in our case a page consists of 2048 integers).

We did not observe an improvement by allocating larger chunks of memory to the sub-stacks. Allocating larger chunks should be profitable if there would be a large difference between random and streamed paging operations. We measured a difference between the costs of these types of access of about 50%, which apparently does not outweigh the additional cost of having larger "loose ends". In the program, this virtual page size can be modified with a parameter.

Another point is memory-alignment. We took care to arrange our stacks so that the virtual pages coincide with physical pages. This reduced the overall time consumption by 15%.

The program was designed to handle random inputs, and that is why, next to the three fields of size N for *succ*, *mast* and *dest*, we only need a stack of size $7/3 \cdot N$. In practice, using little additional memory is also pleasant because this allow larger problems to be solved on a given hardware.

We have measured the time consumption of our program on a standard work-station: a SUN UltraSparc with a clock rate of 175Mhz, a 64MB main-memory and a 2.2GB swap partition on the hard-disc. Notice that it was never our goal to break a benchmark for external list ranking: our goal was to design a better algorithm. Clearly, one would obtain much better performance with a faster hard-disc, and even more so if several hard-discs were used. If the three large fields and the stack are scattered over d discs, than it should be possible to gain almost a factor d.

We tested our program for several k, and found that its choice is non-critical: as long as k is not chosen too large or extremely small, we observed only minimal fluctuations in the time consumption. In Table 4 we give some experimental results, measured for $k = 2^{20}$ (inputs were generated on a machine with 1GB of RAM). In the second column, we give the time per million nodes. For the whole range these numbers continue to increase, which is disappointing.

After double-checking all proves of linearity, we discovered that the continuing deterioration is caused by an increase in the time per paging operation. It turns out that this time is quite accurately described by $t_{\text{op}} = 0.0135 + 0.0045 \cdot \textit{size}/10^9$ seconds, where *size* is the total number of bytes that have to be stored. Measured values are given in Table 5.

P	T	$\frac{T \cdot 10^6}{N}$	$\frac{T \cdot 2048}{t_{\text{op}} \cdot N}$
1	3.63	3.46	—
16	1393	87.8	12.2
32	3239	96.6	12.9
48	5378	106.9	12.9
64	7641	113.9	12.9
80	9556	114.0	12.1
96	12104	120.3	12.1
112	14583	124.4	11.8

Table 4: Performance of the external list-ranking algorithm. The first column gives P; the second the time (in seconds) for ranking a random lists of size $2^{20} \cdot P$; the third column the time per million nodes; and the last column a number that is closely related to the paging volume per node.

$\textit{size}/10^6$	time
500	0.0158
1000	0.0181
1500	0.0202
2000	0.0226

Table 5: Time (in seconds) per paging operation for various sizes of the data space (in MB).

In our case we have estimated $\textit{size} = 4 \cdot 4 \cdot N$ (the *succ* fields are only used initially and the full size of the stack is used only for a very short period). Thus, we get $t_{\text{op}}(P) = t_{\text{op}}(4 \cdot 2^{20} \cdot P) \simeq 0.0135 + 72 \cdot 10^{-6} \cdot P$. If we divide the ranking times by $t_{\text{op}}(P)$ and multiply by $2048/N$, we get the estimates for the paging volume per node, which are given in the last column of Table 4. This is an underestimate, because the first access to a page is much cheaper than later accesses, and the read-only operations cost only $2/3$ of the general operations. So, on basis of Theorem 7, we would expect values of around 12. This nicely coincides with the obtained values. Even more important is that now, finally, we have obtained values that are more or less constant, as they should be.

On the same machine, the sequential algorithm requires $2.37 \cdot 10^{-6} \cdot N$ seconds as long as it is running internally. For $P = 64$ our algorithm is 45 times slower. On the other hand, if the sequential algorithm were to be applied to a list of 2^{26} nodes, it would make two page faults per node (one for finding the initial node, and one for the updates). For that case, we can estimate that it takes about $2 \cdot 0.016 = 0.032$s per node, which is 300 times slower than our algorithm.

7.5 Tree Rooting

As with the parallel algorithm, the external algorithm can be applied to random trees without modification. The expected time consumption is the same as before, but the worst-case performance may be very bad, and not even if the indices are randomized can we give a high-probability guarantee. For the case $P = 8$, we have tested 20 random inputs from each of the categories that were listed in Table 3: list, binary trees and tailed stars. The results are given in Table 6. As in Table 3, we see that the difference between lists and binary trees is small, but that tailed stars may be distributed such that the time considerable exceeds the expected value. For this kind of problems, we also need a larger stack.

	min time	max time	av. time	st. dev.
lists	496	506	500	3
binary trees	498	511	504	5
tailed starts	387	884	530	130

Table 6: Time consumption in seconds for different types of input for $P = 8$ and $k = 2^{20}$.

8 Conclusion

New algorithms were presented for parallel and external list ranking. On the Intel Paragon the parallel algorithm is better than any other in an intermediate range of k and P values. The external algorithm is more of a jump forward. It appears to be several times faster than the best previous algorithm. Both algorithms can also be applied to trees, and stand out by their simplicity and memory-efficiency.

We have observed that the time for performing a paging operation increases linearly with the size of the used data space. This implies that minimizing the paging volume should not be the only guiding principle when designing external algorithms. In our case we have succesfully minimized the size of the data space by using stacks, even though this does not lead to a minimal paging volume in all steps. This slightly larger paging volume is more than compensated for by the cheaper paging operations.

We do not believe that the algorithms can be made deterministic without destroying their performance. Still it would be interesting to look deeper into this. A second open question is whether it is possible to modify the algorithms so that trees can be handled in a way that guarantees the expected time consumptions with high probability.

Acknowledgement

Tillmann Seidel assisted me with the programming of the parallel algorithms. The parallel programs where run on the Paragon at the KFA in Jülich.

References

[1] Anderson, R.J., G.L. Miller, ‘A Simple Randomized Parallel Algorithms for List-Ranking,’ *Information Processing Letters*, 33(5), pp. 269–273, 1990.

[2] Anderson, R.J., G.L. Miller, ‘Deterministic Parallel List Ranking,’ *Algorithmica*, 6, pp. 859–868, 1991.

[3] Atallah, M.J., S.E. Hambrusch, ‘Solving Tree Problems on a Mesh-Connected Processor Array,’ *Information and Control*, 69, pp. 168–187, 1986.

[4] Bäumker, A. W. Dittrich, F. Meyer auf der Heide, ‘Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP-Model,’ *Proc. European Symposium on Algorithms*, LNCS 979, Springer-Verlag, pp. 17–30, 1995.

[5] Berkman, O., U. Vishkin, ‘Recursive Star-Tree Parallel Data Structure,’ *SIAM Journal on Computing*, 22(2), pp. 221–242, 1993.

[6] Cáceres, E., F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, S.W. Song, ‘Efficient Parallel Graph Algorithms for Coarse Grained Multicomputers and BSP,’ *Proc. ICALP 97*, LNCS, Springer-Verlag, 1997.

[7] Chernoff, H., ‘A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations,’ *Annals of Mathematical Statistics*, 23, pp. 493–507, 1952.

[8] Chiang, Y-J, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter, ‘External-Memory Graph Algorithms,’ *Proc. 6th Symposium on Discrete Algorithms*, pp. 139–149, ACM-SIAM, 1995.

[9] Cole, R., U. Vishkin, ‘Deterministic Coin Tossing and Accelerated Cascades: Micro and Macro Techniques for Designing Parallel Algorithms,’ *Proc. 18th Symp. on Theory of Computing*, pp. 206–219, ACM, 1986.

[10] Cole, R., U. Vishkin, ‘Approximate Parallel Scheduling, Part I: the Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time,’ *SIAM Journal on Computing*, 17(1), pp. 128–142, 1988.

[11] Cole, R., U. Vishkin, ‘Faster Optimal Parallel Prefix Sums and List Ranking,’ *Information and Control*, 81, pp. 334–352, 1989.

[12] Cormen T.H., C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[13] Dally, W., C. Seitz, ‘Deadlock Free Message Routing in Multiprocessor Interconnection Networks,’ *IEEE Transactions on Computers*, 36(5), pp. 547–553, 1987.

[14] Dehne, F., W. Dittrich, D. Hutchinson, ‘Efficient External Memory Algorithms by Simulating Coarse-Grained Parallel Algorithms,’ *Proc. 9th Symposium on Parallel Algorithms and Architectures*, pp. 106–115, ACM, 1997.

[15] Dehne, F., S.W. Song, 'Randomized Parallel List Ranking for Distributed Memory Multiprocessors,' *Proc. Asian Computer Science Conference*, LNCS 1179, pp. 1–10, 1996.

[16] Feller, W., *An Introduction to Probability Theory and Its Applications*, Volume I, Third Edition, John Wiley & Sons, New York, 1970.

[17] Gibbons, A., W. Rytter, 'An Optimal Parallel Algorithm for Dynamic Evaluation and its Applications,' *Proc. 6th Conference of Foundations of Software Technology and Theoretical Computer Science*, LNCS 241, pp. 453–469, Springer-Verlag, 1986.

[18] Gibbons, A., W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[19] Gibbons, A.M., Y. N. Srikant, 'A Class of Problems Efficiently Solvable on Mesh-Connected Computers Including Dynamic Expression Evaluation,' *Information Processing Letters*, 32, pp. 305–311, 1989.

[20] Hagerup, T., C. Rüb, 'A Guided Tour of Chernoff Bounds,' *Information Processing Letters*, 33, 305–308, 1990.

[21] Harel, D., R.E. Tarjan, 'Fast Algorithms for Finding Nearest Common Ancestors,' *SIAM Journal on Computing*, 13, pp. 338–355, 1984.

[22] Hirschberg, D.S., A.K. Chandra, D.V. Sarwate, 'Computing Connected Components on Parallel Computers,' *Communications of the ACM,* 22(8), pp. 461–464, 1979.

[23] Hsu, T.-s, V. Ramachandran, 'Efficient Massively Parallel Implementation of some Combinatorial Algorithms,' *Theoretical Computer Science*, 162(2), pp. 297-322, 1996.

[24] JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Inc., 1992.

[25] Kosaraju, S.R., A.L. Delcher, 'Optimal Parallel Evaluation of Tree-Structured Computations by Raking,' *Proc. 3rd Aegean Workshop on Algorithms*, LNCS 319, pp. 101–110, Springer-Verlag, 1988.

[26] Kruskal, C.P., L. Rudolph, M. Snir, 'The Power of Parallel Prefix,' *IEEE Transactions on Computers*, C-34, pp. 965–968, 1985.

[27] Leighton, T., *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan-Kaufmann Publishers, San Mateo, California, 1992.

[28] McColl, W.F., 'Universal Computing,' *Proc. 2nd Euro-Par Conference*, LNCS 1123, pp. 25–36, Springer-Verlag, 1996.

[29] McDiarmid, C., 'On the Method of Bounded Differences,' in *Surveys in Combinatorics*, J. Siemons, editor, 1989 London Mathematical Society Lecture Note Series 141, pp. 148–188, Cambridge University Press, 1989.

[30] Miller, G.L., J.H. Reif, 'Parallel Tree Contraction and its Applications,' *Proc. 26th Symposium on Foundations of Computer Science*, pp. 478–489, IEEE, 1985.

[31] Reid-Miller, M., 'List Ranking and List Scan on the Cray C-90,' *Proc. 6th SPAA*, pp. 104–113, ACM, 1994.

[32] Ryu, K.W., J. JáJá, 'Efficient Algorithms for List Ranking and for Solving Graph Problems on the Hypercube,' *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 83–90, 1990.

[33] Schieber, B., U. Vishkin, 'On Finding Lowest Common Ancestors: Simplification and Parallelization,' *SIAM Journal on Computing*, 17, pp. 1253–1262, 1988.

[34] Setubal, J., J. Meidanis, *Introduction to Computational Molecular Biology*, PWS Publishing Company, Boston, 1997.

[35] Sibeyn, J.F., 'List Ranking on Interconnection Networks,' *Proc. 2nd Euro-Par Conference*, LNCS 1123, pp. 799–808, Springer-Verlag, 1996. Full version in: *Technical Report 11/1995, SFB 124-D6*, Universität Saarbrücken, Saarbrücken, Germany, 1995.

[36] Sibeyn, J.F., 'Better Trade-offs for Parallel List Ranking,' *Proc. 9th Symposium on Parallel Algorithms and Architectures*, pp. 221–230, ACM, 1997.

[37] Sibeyn, J.F., F. Guillaume, T. Seidel, 'Practical Parallel List Ranking,' *Proc. 4th Symposium on Solving Irregularly Structured Problems in Parallel*, LNCS 1253, pp. 25–36, Springer-Verlag, 1997.

[38] Sibeyn, J.F., M. Kaufmann, 'BSP-Like External-Memory Computation,' *Proc. 3rd Italian Conference on Algorithms and Complexity*, LNCS 1203, pp. 229–240, Springer-Verlag, 1997.

[39] Tarjan, R.E., U. Vishkin, 'Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time,' *SIAM Journal on Computing*, 13, pp. 862–874, 1984.

[40] Valiant, L.G., 'A Bridging Model for Parallel Computation,' *Communications of the ACM*, 33(8), pp. 103–111, 1990.

[41] Wyllie, J.C., *The Complexity of Parallel Computations*, PhD Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.