Ultimate Parallel List Ranking ?

Jop F. Sibeyn

MPI–I–1999–1–005 September 1999

Author's Address

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/`

Abstract

Two improved list-ranking algorithms are presented. The “peeling-off” algorithm leads to an optimal PRAM algorithm, but was designed with application on a real parallel machine in mind. It is simpler than earlier algorithms, and in a range of problem sizes, where previously several algorithms where required for the best performance, now this single algorithm suffices. If the problem size is much larger than the number of available processors, then the “sparse-ruling-sets” algorithm is even better. In previous versions this algorithm had very restricted practical application because of the large number of communication rounds it was performing. This main weakness of this algorithm is overcome by adding two new ideas, each of which reduces the number of communication rounds by a factor of two.

Ultimate Parallel List Ranking ?

Jop F. Sibeyn

September 8, 1999

1 Introduction

A *list* is a basic data structure: it consists of nodes which are linked together, so 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: determining for each node i of a set of lists the index of the last node j of its list and the number of links between i and j.

Parallel 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). The simplicity and small memory usage of the presented algorithms facilitates their implementation. At the same time they are highly efficient and outperform existing algorithms. After years of intensive study of this problem, we believe that finally we have found the "ultimate" parallel list-ranking algorithms.

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 [25, 6, 7, 1, 2]. List ranking appears as a subroutine in many graph problems particularly because it is the key ingredient of the *Euler-tour technique* [23].

List-ranking has linear sequential complexity, and can be solved efficiently by a trivial algorithm. This makes it very hard to achieve good speed-ups on a parallel computer, and 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 lower bound on the performance that may be expected for general purpose parallel or external computing.

List ranking also has real practical importance. Here we must be careful not to confuse applications in theory and applications in practice. A true application, which is important in its own right, which appears to not have alternative more practical solutions, and which one would really like to solve for very big problems, is found in the lowest-common ancestor, *LCA*, problem. The LCA problem has wide applications. A recent and outstandingly important one is for performing queries on phylogenetic trees in computational biology [18]. 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 phylogenetic 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 some logarithmic number of accesses.

The LCA problem has been considered by several authors [9, 17, 4]. The algorithm of Berkman and Vishkin [4] 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 (see [11]). 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 [25]. This technique can be used in a list-ranking algorithm which, for a list of length N, runs in $\mathcal{O}(\log N)$ time with $\mathcal{O}(N \cdot \log N)$ work. Using accelerated cascading, the work of this algorithm is reduced to the optimal $\mathcal{O}(N)$, while maintaining running time $\mathcal{O}(\log N)$ [6]. 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 [11]. A variant of [6] and [2] tuned towards the requirements of the BSP model is given in [5].

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 [10] (for the MasPar) and [22] (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 $P = 100$ *PUs* (processing units), the maximum obtained speed-up[1] is 14 [22]. The version of independent-set removal presented in this paper is slightly better: it achieves speed-up 17 on a Paragon with $P = 100$ and $k = N/P = 10^6$. Reid-Miller [16] 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 et al. [22]. This *sparse-ruling-set* algorithm is unbeatable when either the *start-up costs* (the costs for sending a packet away) are 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. For more practical values of k, acceptable speed-ups are achieved in [20].

1.3 New Results

Peeling-Off Algorithm. The first algorithm of this paper is based on a further development of the ideas from [20]. In one of the algorithms in [20] the input is divided in two halves, which can be solved independently. After $\log P$ halvings, the resulting subproblems can be solved in each PU.

---

[1] *Speed-up* is defined as the ratio of the sequential time consumption and the parallel time consumption. In our case, we compare our parallel times with the time consumption of the simple sequential algorithm running on a single PU of the Paragon.

Our new algorithm is based on the same underlying idea for dividing the input. However, now the input is not divided, but a fraction of it is split off. Hereafter the problem is solved on the other fraction, and finally the fraction that was split off is reinserted. This combination of the structure of independent-set removal with techniques from [20] is new. A great advantage over independent-set removal is that the fraction to be split off is not fixed. This makes that the algorithm can be tuned to the parameters of the input and the machine.

This algorithm performs comparable to the best three algorithms of [20] in their respective ranges of optimality. It essentially relies on the new observation that pointer-jumping is highly efficient for a set of lists that have constant expected length. An implementation on an Intel Paragon with $P = 100$ and $k = 10^6$ achieves speed-up 25.

A further strong point of our algorithm is that it 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 algorithms, among them independent-set removal, become inefficient or break-down for tree rooting. A major exception is pointer-jumping, but this algorithm is very inefficient in itself when the trees are not shallow. Thus our algorithm allows to compute basic tree functions without resorting to the Euler-tour technique (see [11]), saving the involved overhead.

Improved Sparse-Ruling-Set Algorithm. A basic version of the sparse-ruling-set algorithm was presented in [16] and a more elaborate version, with another application in mind in [19]. Recently, Ranade has rediscovered the algorithm [15], adding a few interesting ideas that went unnoticed so far. In this paper we add two more new ideas.

The first idea is universal and can be applied within any model of parallel computation. The basic idea of the sparse-ruling-set algorithm is a reduction round, in which a certain number of nodes is selected as rulers, which initiate waves running down the lists link by link. Such a wave runs until it reaches a next ruler, or until after a certain maximum number of hops all waves are stopped. Hereafter, we must further deal with the rulers and with the unreached nodes. In [19], they are treated separately. This makes that the number of subproblems grows exponentially with the depth of the recursion, practically limiting this depth to at most two. In [15], rulers and unreached nodes are not distinguished, solving one bigger subproblem instead. In the earlier sparse-ruling-set algorithms, all rulers are chosen at the beginning of the reduction round. This results in a gradually decreasing number of running waves. There is an alternative! We suggest that if a wave dies upon reaching a next ruler, a new ruler is selected from among the unreached nodes, thus keeping the number of running waves more or less constant. This leads to a very different and much simpler situation: after a fixed number of hops, all nodes and rulers have been reached by a wave from the preceding ruler. On PRAMs, it makes allocating the work simpler than before. On interconnection networks, the maximal size of the communication buffers is smaller. It even turns out that now, with a given number of hops, the reduction of the problem size is twice as large. This approach has an additional advantage. In the earlier algorithms, each ruler had to be informed about the node in which its wave had ended, and the distance thereof, to form a sublist of rulers and unreached nodes. If there are no unreached nodes, we can create a reversed list of rulers without exchanging data. This saves a step at the end of each reduction round and makes the algorithm simpler.

The second idea is more specifically suited for interconnection networks. Most parallel programs work according to the following pattern: computations are carried out for a certain time, then there is a synchronization and all PUs exchange data in a communication round. This structure is so general, that in the BSP-model [24, 12, 3] it is imposed on all parallel programs. Unless the problem has a specific structure, the communication will be of the *all-to-all* type, meaning that every PU has to exchange data with all other PUs. There are several ways to perform such an all-to-all communication, but on a network with P PUs, for sufficiently large packets, it is most efficient to decompose it into $P - 1$ *permutations*, in which each PU is the source and destination of exactly one packet. How about the following alternative: instead of performing the $P - 1$ permutations at the end of each computation round, we compute all there is to compute, then perform the first permutation, then again some computation, then the second permutation, and so on. In this way, the computation may profit from data that, on the average, become known earlier. In general this may not be a very interesting approach, but for all algorithms in which information is gradually gathered it is. We will show that with this approach, on the average, a wave progresses twice as fast, halving the required number of communication rounds for reaching all nodes. Other advantages of this approach are a reduced need for routing buffers and better possibilities for overlapping computation and communication (*latency hiding*).

1.4 Structure of the Paper

After some preliminaries, we first describe an optimized version of independent-set removal. Then we present the peeling-off algorithm in Section 4. In Section 5 we present the improved sparse-ruling-set algorithm. Missing details might be found in [21].

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, stored in a field *succ*. A final node j can be recognized by a distinguished value of *succ*(j). The output consists of two arrays: for every $0 \leq j < N$, the master of j, *mast*$[j]$, should give the index of the final node of the list or tree to which j belongs, and *dist*$[j]$ should give the number of links between j and *mast*$[j]$. The number of PUs is denoted P, and every PU holds exactly $k = N/P$ nodes of the lists: PU_i, $0 \leq i < P$, holds the nodes with indices $k \cdot i + j$, for all $0 \leq j < k$.

We will assume that the *indexing* of the nodes is selected uniformly from the set of all $N!$ permutations of N elements. Notice that we do *not* assume that the structure of the lists or the trees is random.

Cost Model. Except for a PRAM section, we will express the quality of our parallel algorithms by giving their *routing volume*, the number of integers sent by a PU, and the number of all-to-all routing operations (informally we call this the number of communication rounds). Both these notions are well-defined, and can be determined precisely. Our cost measure has proven to be a fairly reliable instrument for predicting the practical behavior of algorithms [20]. It can be viewed as a simplification of BSP or BSP* [24, 12, 3]. In the analysis of our parallel algorithms, we will mostly assume that N is much larger than P. For all-to-all communication patterns, we should even have $k \gg P$.

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

Lemma 1 (Azuma Inequality) [13] *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 give an optimized version of independent-set removal.

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 order. At all times, there is a list of active nodes. Initially, all non-final nodes p are active and set $\mathit{mast}(p) = \mathit{succ}(p)$ and $\mathit{dist}(p) = 1$. 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 $\mathit{mast}(p)$.

2. If a node p which selected a 0 receives a packet, then it is removed from the list of active nodes and added to the list of nodes excluded during Phase t. It sends $\mathit{mast}(p)$ and $\mathit{dist}(p)$ back to the sending node. Otherwise it sends back the number -1 (as a place holder).

3. If an active node p receives -1, then it does nothing. Otherwise it uses the received data to update $\mathit{mast}(p)$ and $\mathit{dist}(p)$.

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, $\mathit{mast}(p)$ gives the index of the last node of the list and $\mathit{dist}(p)$ its distance.

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 $\mathit{mast}(p)$ and *dist*(p).

3. Each node p that was excluded during Phase t uses the received data to update $\mathit{mast}(p)$ and $\mathit{dist}(p)$.

Lemma 2 *A parallel implementation of the independent-set-removal algorithm has routing volume $(8 + o(1)) \cdot k$, with high probability. One level of* REDUCTION *and* INSERTION *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 2 and $1/4$ a packet of size 1. 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 $\sum_{t=0}^{\infty}(3/4)^t = 4$ for the later phases, we obtain $8 \cdot k$. The Azuma inequality can be used to prove that the deviation from the expected value is small. □

In Table 3.1, we provide a few measured efficiencies of the algorithm, where by *efficiency* we mean *speed-up*$/P = T_{\mathrm{seq}}/(P \cdot T_{\mathrm{par}})$. As a basis for the computation of our efficiencies, we assumed that $T_{\mathrm{seq}} = 3.9 \cdot 10^{-6} \cdot N$, for all N: running on one node of the Paragon, the simple sequential algorithm requires 3.9s for solving a problem with $N = 10^6$.

$\log_2 k$	$P = 4$	$P = 16$	$P = 64$
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 3.1: Efficiencies of independent-set removal, running on an Intel Paragon for various P and k. In all cases we performed ten reduction phases.

4 Peeling-Off Algorithm

4.1 Basic Idea

The basic idea of our first algorithm is to split the nodes into two sets: $\mathcal{S}_0$ and $\mathcal{S}_1$. The set of all nodes is denoted $\mathcal{N}$. 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 *dist*.

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 *dist* fields. Then they update their *mast* and *dist* 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

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

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

Lemma 3 PEELING_OFF *correctly computes the values of mast and dist 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. ☐

4.2 PRAMs

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, 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 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

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

Finally pointer jumping is performed on $\mathcal{S}_0(\log\log N) = Buc_0$. Unwrapping the recursion, gives the following equivalent iterative formulation:

```
    Procedure 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($\mathcal{S}_1(t)$).
```

Analysis. The correctness of PRAM_RANK follows from Lemma 3 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 $\sum_{t=1}^{\log\log N} \#\mathcal{S}_0(t) = \sum_{t=1}^{\log\log N} \#\mathcal{S}_1(t) = N - N/\log N$, the work for altroclean 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 [11]). The autocleans are performed by applying the basic pointer jumping step (every node which has a master in $S_1(t)$ asks its master for its *mast* and *dist* values) until no nodes are active anymore. Their time consumption is analyzed in Lemma 5.

Lemma 4 *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* $\mathcal{N} - \mathcal{S}_0(t)$*. 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* $\mathcal{N} - \mathcal{S}_0(t)$*.*

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 n be the node defined above. 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 $n' \in \mathcal{N} - \mathcal{S}_0(t)$. 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. $\square$

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 $\mathcal{N} - \mathcal{S}_0(t)$. 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 5 *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

$$E(\mathit{actives}(t)) = \sum_{s \geq 0} 2^{-2^s} \cdot \#\mathcal{S}_1(t) < 0.82 \cdot \#\mathcal{S}_1(t). \tag{4.1}$$

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 [8, 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 $(2 \cdot \log\log N) \cdot (1 + o(1)) \cdot \#\mathcal{S}_1(t)/N^{1/3} = o(\#\mathcal{S}_1(t))$. $\square$

Theorem 1 *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.*

Tree Rooting. For tree rooting we apply the same algorithm. The only difference is that now several nodes may have the same successor. Lemma 4, the conclusion about the structure and Corollary 1 still hold, and we have the following partial analogue of Theorem 1:

Theorem 2 *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. □

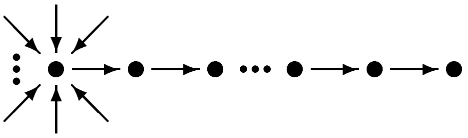

Figure 4.1: A tailed star.

The good news is that Theorem 2 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 4.1. Assume that the star contains $N - \log N/\log\log N$ nodes, and the tail $\log N/\log\log N$. Possibly all nodes of the tail are allocated to the $\mathcal{S}_1(1)$. But then AUTOCLEAN($\mathcal{S}_1(1)$) requires $\Omega(N \cdot \log N)$ work.

4.3 Distributed Memory Machines

Algorithm. Our algorithm for distributed memory machines is inspired by 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. To optimize the algorithm, we use one-by-one cleaning from [20] instead of pointer-jumping:

Lemma 6 [20] *For ranking a set of lists with a total of $k \cdot P$ nodes on a parallel computer with P PUs, one-by-one cleaning requires $3 \cdot P - 4$ start-ups, and has routing volume $6 \cdot \ln P \cdot k$.*

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 number of all-to-all routings and the routing volume. A handy choice is

$$f_t(d) = \frac{1 + d - t}{2 + d - t}. \tag{4.2}$$

With these f_t, we get a simple expression:

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

Theorem 3 *When d reduction phases are performed with reduction factors as in (4.2), the routing volume is less than*

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

with high probability. For each reduction phase, the algorithm requires $6 + 2 \cdot \lceil \log \log N \rceil$ *all-to-all routings.*

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 (4.1) for the total number of questions:

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

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 $k \cdot P/(d + 1)$.

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 $1/N^2$. □

Experimental Results. To minimize the costs of the autocleans, one should make the first f_t somewhat larger, and later on somewhat smaller:

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

The d for which the time consumption is minimized increases with k, and decreases with P. However, $d = 6$ always gives results that are close to optimal. 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. 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 suffice.

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 version are available at

$\log_2 k$	$P = 4$	$P = 16$	$P = 64$
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 4.1: Efficiencies of the parallel algorithm running on an Intel Paragon for various P and k. In all cases $d = 6$.

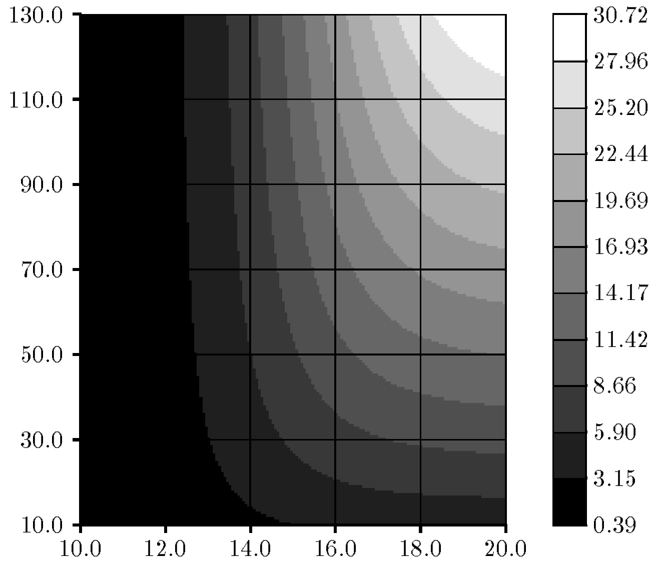

Figure 4.2: Experimental results for the parallel algorithm: the x-axis gives $\log k$, the y-axis P, and the gray-tones the speed-up. In all cases $d = 6$.

`http://www.mpi-sb.mpg.de/~jopsi/dprog/prog.html`. The time consumption on the Paragon can be described up to 10% by

$$T_1(P,k) = 38 \cdot 10^{-3} + 40 \cdot 10^{-7} \cdot k + 58 \cdot 10^{-4} \cdot P + 24 \cdot 10^{-7} \cdot k \cdot \log(P).$$

In Table 4.1, we provide a few measured efficiencies. 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 increases with P; the load-balancing becomes worse.

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. The results are given in Table 4.2. Most apparent is the increase of the standard deviation.

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

5 Sparse-Ruling-Set Algorithm

Earlier versions of the sparse-ruling-set algorithm were given in [16, 19, 15]. Its basic idea allows for a highly efficient reduction of the problem size by a large factor: during the whole algorithm, most nodes are addressed only twice. This is not more than in the sequential algorithm, and therefore one would hope to obtain very high speed-ups. Unfortunately, on parallel computers with a distributed memory, good performance is achieved only when N/P^2 is very large, because the number of communication rounds is too high. In this section we introduce several new ideas that allow to reduce the number of communication rounds considerably. Before presenting the algorithm, we give an example, which illustrates the orders of magnitude we are talking about.

On a single PU of an Intel Paragon, sequential list ranking takes $3.9 \cdot 10^{-6} \cdot N$s. So, on a machine with P PUs we strive for a time close to $3.9 \cdot 10^{-6} \cdot k$. Start-ups cost about $2 \cdot 10^{-4}$s, and each communication round requires $P - 1$ of them. The algorithm of [19] requires about $f \cdot \ln f$ communication rounds for reducing a problem of size N to two subproblems of size N/f. For a problem with $k = 10^6$, one might apply two reduction rounds with $f = 20$ each, requiring $3 \cdot (60 + 1 + 2) + 4 \cdot 3 = 201$ communication rounds. On a Paragon with $P = 100$, the goal would be 3.9s, but we do not even come close to this, because we waste $201 \cdot 99 \cdot 2 \cdot 10^{-4} = 4.0$s with the start-ups. Applying the idea of Ranade [15], a problem of size N is reduced to one subproblem of size $2 \cdot N/f$. If again we apply two rounds with $f = 20$, we would need $2 \cdot (60 + 1 + 2) + 3 = 129$ communication rounds. For the same two reductions, the algorithm of this section requires $2 \cdot (25 + 2) + 3 = 57$ communication rounds. The resulting time for the start-ups is $57 \cdot 99 \cdot 2 \cdot 10^{-4} = 1.1$s, which is more reasonable.

5.1 Constant Number of Active Waves

Consider the following simple list-ranking algorithm:

```
Algorithm SPARSE-RULING-SET($N, S$)
1. Select $S$ nodes randomly and uniformly from among the non-final nodes as rulers.
2. Each ruler $p$ initiates a wave: $p$ prepares a packet containing its index and dist($p$) to be sent to succ($p$).
3. Mark all final nodes as rulers.
4. for round $= 0$ to $N/S - 1$ do
   a. Send all packets.
   b. Each node $p$ that receives a packet marks
      the contained data.
   c. if $p$ is a non-ruler then
         $p$ prepares a new packet for succ($p$), adding
         dist($p$) to the second field in the packet.
      else
         A new ruler $p'$ is selected from among
         the unreached non-rulers. $p'$ initiates
         a new wave as was done in Step 2.
```

In every round, S nodes are reached that were not reached before. Thus, all nodes have been reached after N/S rounds. Particularly, all non-initial rulers have been reached by waves from their predecessors, and thus a sublist with links consisting of all rulers can be constructed without further communication. The rest of the algorithm is the same as in independent-set removal: after some more reduction rounds, pointer-jumping or one-by-one cleaning is performed, and finally the excluded nodes are reinserted.

SPARSE-RULING-SET is not much more than S sequential algorithms running in parallel. Different from the sequential algorithm, there is no need to search for the initial nodes, on the other hand all excluded nodes must be reinserted. These operations have comparable complexity.

Lemma 7 *On an interconnection network, an application of* SPARSE-RULING-SET *causes a routing-volume of* $3 \cdot k$.

Proof: Every node is sending and receiving exactly one packet. Such a packet carries the length of the covered path, the index of the ruler and the destination. ☐

Let $H_n = \sum_{i=1}^{n} 1/i$ be the n-th harmonic number.

Theorem 4 *The expected number of rulers selected by* SPARSE-RULING-SET*(N, S) equals* $S \cdot H_{N/S}$.

Proof: Let α_t, $0 \leq t < N/S$, denote the number of rulers that is selected in Round t. $\alpha_0 = S$. Generally, in Round t, there are still $N - t \cdot S$ unreached nodes, and among them there are exactly S rulers. Thus,

$$E(\alpha_{t+1}) = S \cdot \frac{S}{N - t \cdot S}.$$

As expected values may be added, we get the stated result. ☐

We call an algorithm for reducing the size of list-ranking problems *effective*, if for a given number of routing steps it achieves a large reduction.

Corollary 3 *Asymptotically the presented sparse-ruling-set algorithm is twice as effective as earlier versions.*

Proof: In the previous sparse-ruling-set algorithms, $f \cdot \ln f$ rounds were required for a reduction of the problem size from N to $2 \cdot N/f$. If in our algorithm we set $S = N/(f \cdot \ln f)$, a problem of size N is reduced to $N/(f \cdot \ln f) \cdot H_{f \cdot \ln f} \simeq N/f \cdot (1 + \ln \ln f / \ln f)$. □

How can new rulers be selected efficiently? If initially a random selection of N/P nodes is assigned to each of the P PUs, then a PU finds a new ruler by searching locally for the first unreached non-ruler. If all its nodes are exhausted, then no new ruler is selected. In total over all rounds, this selection is $\mathcal{O}(N/P)$ work. One extra round is required:

Lemma 8 *If the new rulers are selected locally, and $S/P \geq 64 \cdot \log P$, then all nodes are reached within $N/S + 1$ rounds, with high probability.*

Proof: As long as there have been S active waves in all previous rounds, there are $N - t \cdot S$ unreached nodes at the beginning of Round t. Among them there are exactly S rulers. Thus, after the sending in Round $N/S - 3$, in each PU an expected $S/(3 \cdot P)$ new rulers have to be selected out off its expected $4/3 \cdot S/P$ unreached non-rulers. This is possible with high probability. We show that the second number is at least $2/3 \cdot S/P$, with probability $1 - 1/P^2$.

Consider a random variable X that is the sum of the outcomes of independent Poisson trials with expected value μ. Then Theorem 4.2 from [14] states that

$$Pr[X < (1 - \delta) \cdot \mu] < e^{-\mu \cdot \delta^2 / 2}.$$

In our case $\mu = 4/3 \cdot S/P$, and $\delta = 1/2$. With the given bound on S/P, the bounding is strong enough. How about the required independency? Clearly, for a given list, the probability that a node in a PU has been reached is *not* independent of the earlier decisions. However, the situation we find is exactly the same as when the allocation of a node to a PU were postponed until this node is reached.

So, we may assume that after the sending in Round $N/S - 2$ there remain S unreached nodes. Now for the first time, it may happen that a PU has not enough indices of unreached non-rulers. But, as above, we can show that each PU will forward at least $3/4 \cdot S/P$ waves, with high probability. Thus, in the following round, the expected number of unreached nodes in a PU is less than $S/(4 \cdot P)$ and the expected number of received packets more than $3/4 \cdot S/P$: in every PU the number of waves hitting a ruler will exceed the number of unreached non-rulers, so all of the latter are turned into rulers, and the routing is completed in the next round. □

The condition in Lemma 8 is an obstacle for optimal-time PRAM algorithms. This problem may be overcome, by granting a few extra steps at the end.

5.2 Interlacing Computation and Communication

In SPARSE-RULING-SET, an all-to-all routing is performed in Step 4.a. Instead of this, the all-to-all routing might also be decomposed into $P - 1$ permutations. For example, PU i might send in Step j, $1 \leq j < P$, to PU $(i + j) \bmod P$. After routing each such a permutation, the received data can be processed immediately as in Step 4.b and 4.c, before routing the following packet. If the successor of a newly reached node is residing in the same PU, then a shortcut can be made without waiting. In the original algorithm, a wave is progressing exactly one step after every full communication round, now it is progressing twice as fast.

The first packets sent have expected size S/P^2. Then these sizes increase to reach their maximum of about $e \cdot S/P^2$ at the end of Round 1. Hereafter, the expected packet sizes start to converge.

Lemma 9 *After converging, the expected size of the routed packets equals* $2 \cdot S/P^2$.

Proof: Let $s_{i,h,j}$, $1 \leq h, j < P$, be the expected size of the packet PU i will send in Permutation h at the moment Permutation j is sent. We want to determine $x = s_{i,h,h}$. By performing a permutation, the size of the packet to be routed in this permutation is reset to 0. When a packet is received, on the average, it is equally distributed over the $P - 1$ packets in preparation, adding $x/(P-1)$ to all packets. Thus, $s_{i,j-h,j} = h \cdot x/(P-1)$ (for $h \geq j$, $j - h$ should be replaced by $j - h + P - 1$). Particularly, $\sum_h s_{i,h,j} = x \cdot P/2$. On the other hand, at any time, the expected number of heads of waves in each PU is S/P, thus $\sum_h s_{i,h,j} = S/P$. Equating gives $x = 2 \cdot S/P^2$. □

Theorem 5 *If routing rounds are decomposed in* $P-1$ *permutation routings interlaced with data processing, then all nodes are reached in* $N/(2 \cdot S) + 1$ *rounds, with high probability.*

Proof: Though in the later rounds the packet sizes converge to $2 \cdot S/P^2$, most of them are slightly smaller during the first rounds. It is easy to see that in Permutation j, $1 \leq j \leq P-1$ of the first round the expected size of the packets equals $S/P^2 \cdot (1 + \frac{1}{P-1})^j$. Thus, the sum of the sizes of all packets is approximately $(e-1) \cdot S$. In the following rounds, the sum of the sizes of the packets rapidly converges to $2 \cdot S$. Thus, it can be estimated that after $t = N/(2 \cdot S) - 1$ rounds more than $(2 \cdot t - 1) \cdot S$ nodes have been reached. The remaining nodes are reached in the final two rounds, even if new rulers are selected locally. □

Corollary 4 *Asymptotically, interlacing computation and communication makes* SPARSE-RULING-SET *twice as effective.*

Proof: The interlacing gives a small increase of the number of selected rulers. A refinement of the analysis of Theorem 4 shows that with increasing P this number converges to $(\ln(N/S) + 1) \cdot S$. So, in $N/S + 1$ rounds we now achieve a reduction of the problem size to $(\ln(N/S) + 2) \cdot S/2$, whereas before we achieved a reduction to $(\ln(N/S) + \gamma) \cdot S$. Here γ is the Eulerian number. For large N/S, the ratio converges to two. □

5.3 Experiments

A sequential simulation of SPARSE-RULING-SET with interlaced computation and communication has been programmed in C. This program is identical to a parallel program, except that sending operations have been replaced by memcopies, that all variables have been replaced by fields, and that all instructions have been replaced by loops over the processor numbers. It is available at `http://www.mpi-sb.mpg.de/~jopsi/dprog/prog.html`.

The program shows some additional advantages of the new algorithm. The simplicity of the algorithm leads to a very short and simple code. This does not only mean that it takes less effort to program, debug and optimize, but it also means that less instruction cache is required. Another great advantage is that both new ideas help to reduce buffer sizes. Having a fixed number of active waves, means that we can fix the buffer sizes to the required value. The interlacing means that we need only a small buffer for receiving packets. Small buffers are advantageous because of cache usage, and in order to limit the additional memory requirements. Only the highest level of reinsertion would require large packets. If the available buffer size is insufficient, this routing is (at the expense of a few extra routing operations) divided in several chunks. In addition to the $3 \cdot N$ integers for storing the *succ*, *mast* and *dist* fields, our program requires less than $N/2$ additional storage. Hereby, it is by far the most memory-efficient list-ranking algorithm.

N	S	Rounds	N'
4194304	524288	6	1612992
4194304	262144	10	991936
4194304	131072	18	584288
4194304	65536	34	338336
4194304	32768	66	191216
4194304	16384	130	107696

Table 5.1: Reductions of the problem sizes by SPARSE-RULING-SET with interlaced computation and communication. The columns give N, S, the number of communication rounds, and the resulting problem size.

In Table 5.1 some examples of the efficiency of the new reduction method are given. We can see that with just 20 communication rounds, the problem size is reduced by a factor 7. Unfortunately the Paragon has been turned of, so we could not test this algorithm in the real parallel practice. However, all previous experiments have shown that the performance of the Paragon can be predicted rather accurately. On basis of our sequential simulations we conclude that for $P = 100$, the improved sparse-ruling-sets algorithm definitely should achieve a speed-up of more than 30 for $k = 2^{18}$. Previously speed-up 30 was only achieved for $k = 2^{21}$.

6 Conclusion

Two new algorithms were presented for list ranking. The peeling-off algorithm is simple and versatile. The second algorithm is a simplified and improved version of the sparse-ruling-set algorithm. By strongly reducing the number of performed communication rounds, this most efficient parallel list-ranking algorithm becomes applicable for much smaller problem sizes than before. For practical applications this means a great step forwards.

Bibliography

[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] Bäumker, A. W. Dittrich, F. Meyer auf der Heide, 'Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP-Model,' *Proc. 3rd European Symposium on Algorithms*, LNCS 979, Springer-Verlag, pp. 17–30, 1995.

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

[5] 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. 24th International Colloquium on Automata Languages and Programming*, LNCS 1256, pp. 390–400, Springer-Verlag, 1997.

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

[7] 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.

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

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

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

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

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

[13] 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.

[14] Motwani, R., P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, 1995.

[15] Ranade, A., 'A Simple Optimal List Ranking Algorithm,' *Proc. of 5th High Performance Computing*, Tata McGraw-Hill Publishing Company, 1998.

[16] Reid-Miller, M., 'List Ranking and List Scan on the Cray C-90,' *Journal of Computer and System Sciences*, 53(3), pp. 344–356, 1996.

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

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

[19] Sibeyn, J.F., 'List Ranking on Interconnection Networks,' *Proc. 2nd Euro-Par Conference*, LNCS 1123, pp. 799–808, Springer-Verlag, 1996.

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

[21] Sibeyn, J.F., 'From Parallel to External List Ranking,' *Techn. Rep. MPI-I-97-1021*, Max-Planck Institut für Informatik, Saarbrücken, Germany, 1997. Available at `http://www.mpi-sb.mpg.de/~jopsi/`.

[22] 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.

[23] 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.

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

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

INFORMATIK

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph

MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values