Parallel Graph Algorithms

S. W. Song*
Universidade de São Paulo
Instituto de Matemática e Estatística
Rua do Matão, 1010
05508-900 São Paulo, SP, Brazil
e-mail: song@ime.usp.br

1 Introduction

Many applications in parallel machines that appeared in the literature are restricted to trivially parallelizable problems with low communication requirements. In real machines communication time is often much greater than computation time. Therefore for non-trivial problems, such as many graph problems, theoretically efficient parallel algorithms for PRAM or fine grained network models often give disappointing speedups when implemented on real machines.

The CGM (Coarse Grained Multicomputer) model was proposed [15, 16, 17] to be an adequate model of parallelism sufficiently close to existing parallel machines with distributed memory. It is a simple model but nevertheless intends to give a reasonable prediction of performance when parallel algorithms on this model are implemented. CGM algorithms are expected to have theoretical complexity analyses close to actual times observed in real implementations.

In his excellent book [32], Reif presents the *synthesis approach* to algorithm design. With this approach, a family of related algorithms can be synthesized from some very basic algorithms. He presents a family of graph problems for which sophisticated parallel algorithms are synthesized from parallel algorithms for simpler basic problems. For instance, to solve the open ear decomposition problem, we can first solve the lowest common ancestor problem, which in turn can be based on the range minima problem and the list ranking problem. It is our purpose to consider some of the very basic problems such as list ranking, determination of the connected components of a graph, and range minima. All the presented algorithms in [32] however are based on the PRAM model. This text considers the CGM model for which there exist, to the best of our knowledge, only a few results on parallel graph algorithms.

In Section 2 we present the CGM model. In Section 3 we consider the basic list ranking problem and give two algorithms: one random algorithm and another deterministic algorithm. In Section 4 we present the problem of determining the connected components of a graph. In Section 5 we present a very efficient algorithm for the range minima problem. At the end of each of these sections we give simulation or experimental results obtained on real parallel machines. In Section 6 we mention the lowest common ancestor problem and observe that it can be based on the range minima problem. Finally in Section 7 we conclude with a summary of the presented results.

*Supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) Proc. No. 98/06138-2, CNPq Proc. No. 52 3778/96-1, and CNPq/NSF Collaborative Research Program Proc. No. 680037/99-3.

The current text is based on results presented in several recent conferences and journal articles. The probabilistic algorithm for list ranking has appeared in ASIAN'96 [18] and its deterministic version in ICALP'97 [12]. The connected components algorithm has been presented in ICALP'97. The parallel algorithm for range minima has appeared in the proceedings of IRREGULAR'99 [27]. This text is therefore the result of many researchers.

2 The Coarse Grained Multicomputer (CGM)

The CGM (Coarse Grained Multicomputer) model [15, 16, 17] is similar to Valiant's BSP (Bulk-Synchronous Parallel) model [35]. However it uses only two parameters, n and p, where n is the size of the input and p the number of processors each with $O(n/p)$ local memory. The term "coarse grained" means the local memory size is considerably greater than $O(1)$. Usually we require $n/p > p$.

All processors are connected by a router that can deliver messages in a point to point fashion. A CGM algorithm consists of an alternating sequence of *computation rounds* and *communication rounds* separated by barrier synchronizations.

A computation round is equivalent to a computation superstep in the BSP model. In this computation round, we usually use the best possible sequential algorithm in each processor to process its data locally. A communication round consists of a single h-relation with $h \leq n/p$, that is, each processor can exchange a total of $O(n/p)$ data with other processors in one communication round.

In the CGM model the communication cost of a parallel algorithm is modeled by the number of communication rounds. The objective is to design algorithms that require a small number of communication rounds. Many algorithms for graph and geometric problems [12, 15, 16, 17] require only a constant or $O(\log p)$ rounds. As p is usually small compared to n the resulting CGM algorithm is expected to be efficient when implemented in practice. Observe that PRAM algorithms frequently are designed for $p = O(n)$ and each processor receives a small amount of input data. The CGM model considers a more realistic case of $p << n$.

Notice that the CGM model is particularly suitable in cases where the overall computation speed is considerably larger than the overall communication speed, and the problem size is considerably larger than the number of processors, which is usually the case in practice.

3 List Ranking

Consider a linear linked list consisting of a set S of n nodes and, for each node $x \in S$, a pointer $(x \rightarrow next(x))$ to its successor, $next(x)$, in the list. Let $\lambda \in S$ be the last list element and $next(\lambda) = \lambda$. The list ranking problem consist of computing for each $x \in S$ the distance of x to λ, referred to as $dist(x)$. We assume that, initially, every processor stores n/p nodes and, for each of these nodes the pointer $(x \rightarrow next(x))$ to the next list element. See Figure 1. As output we require that every processor stores for each of its n/p nodes $x \in S$ the value $dist(x)$.

A trivial sequential algorithm solves the list ranking problem in optimal linear time by traversing the list. Several PRAM list ranking algorithms have been proposed [23, 30]. The first optimal $O(\log n)$ EREW PRAM algorithm is due to Cole and Vishkin [13]. Another optimal deterministic algorithm is given by Anderson and Miller [4]. Parallel list ranking algorithms using randomization were proposed by Miller and Reif [25, 26]. The algorithms use $O(n)$ processors. The optimal algorithm by Anderson and Miller [5] improves this by using an optimal number of processors. A

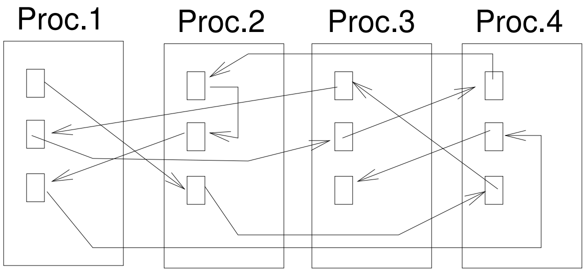

Fig. 1: A Linear linked list stored in a distributed memory multiprocessor.

$O(\sqrt(n))$ time mesh algorithm is described in [6].

3.1 A Randomized Parallel Algorithm for List Ranking

Consider a linear linked list with a set S of n nodes. In this section we will show that if we select $\frac{n}{p}$ random elements (pivots) of S then, with high probability, these pivots will split S into sublists whose maximum size is bound by $3p\ln(n)$.

Lemma 1 *$xk \leq n$ randomly chosen elements of S (pivots) partition list S into sublists S_i such that the size of the largest sublist is at most $\frac{n}{x}$ with probability at least $1 - 2x(1-\frac{1}{2x})^{xk}$.*

Proof. (Analogous to [11]) Assume that the nodes of S are sorted by their rank. This sorted list can be viewed as $2x$ segments of size $\frac{n}{2x}$. If every segment contains at least one pivot (chosen element), then $\max_{1\leq j\leq xk}|S_j| \leq \frac{n}{x}$. Consider one segment. Since the pivots are chosen randomly, the probability that a specific pivot is not in the segment is $(1-\frac{1}{2x})$. Since xk pivots are selected independently, the probability that none of the pivots are in the segment is $(1-\frac{1}{2x})^{xk}$. Therefore, even assuming mutual exclusion, the probability that there exists a segment which contains no pivot is at most $2x(1-\frac{1}{2x})^{xk}$. Hence, every segment contains at least one pivot with probability at least $1-2x(1-\frac{1}{2x})^{xk}$. □

Corollary 1 *$xk \leq n$ randomly chosen pivots partition list S into $xk+1$ sublists S_i such that there exists a sublist S_i of size larger than $c\frac{n}{x}$ with probability at most $\frac{2x}{c}(1-\frac{c}{2x})^{xk} \leq \frac{2x}{c}e^{-\frac{1}{2}ck}$.*

Lemma 2 *Consider $xk \leq n$ randomly chosen pivots which partition S into $xk+1$ sublists S_i, and let $m = \max_{0\leq i\leq xk}|S_i|$. If $k \geq \ln(x) + 2\ln(n)$ then Prob$\{m > c\frac{n}{x}\} \leq \frac{1}{n^c}$, $c > 2$.*

Proof. Corollary 1 implies that Prob$\{m > c\frac{n}{x}\} \leq \frac{2x}{c}e^{-\frac{1}{2}ck}$.

We observe that, for $c > 2, \ln(x) + 2\ln(n) \leq k \;\Rightarrow\; \frac{2}{c}\ln(\frac{2x}{c}) + 2\ln(n) \leq k$

$\Rightarrow\; \ln(\frac{2x}{c}) + c\ln(n) \leq \frac{ck}{2} \;\Rightarrow\; \frac{2x}{c}n^c \leq e^{\frac{ck}{2}} \Rightarrow$ Prob$\{m > c\frac{n}{x}\} \leq n^{-c}$ □

Theorem 1 *$\frac{n}{p}$ randomly chosen pivots partition S into $\frac{n}{p}+1$ sublists S_j with $m = \max_{0\leq j\leq p}|S_j|$ such that Prob$\{m \geq c3p\ln(n)\} \leq \frac{1}{n^c}, c > 2$*

Proof. Let $x = \frac{n}{3p\ln(n)}$, $k = \ln(x) + 2\ln(n) = 3\ln(n) - \ln(3p\ln(n))$.

Then $xk = \frac{n}{p}\frac{3\ln(n)-\ln(3p\ln(n))}{3\ln(n)} \leq \frac{n}{p}$, and Theorem 1 follows from Lemma 2. □

3.1.1 A Simple Algorithm Using a Single Random Sample

We present a simple list ranking algorithm which requires, with high probability, at most $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ communication rounds. This algorithm is based on a single random sample of nodes.

($\tilde{O}(n)$ denotes $O(n)$ "with high probability". More precisely, $X = \tilde{O}(f(n))$, if and only if $(\forall c > c_0 > 1)$ $Prob\{X \geq cf(n)\} \leq \frac{1}{n^{g(c)}}$ where c_0 is a fixed constant and $g(c)$ is a polynomial in c with $g(c) \to \infty$ for $c \to \infty$ [28].)

Consider a random set $S' \subset S$ of pivots. For each $x \in S$ let $nextPivot(x, S')$ refer to the closest pivot following x in the list S. (Without loss of generality, assume that the last element, λ, of S is selected as a pivot and let $nextPivot(\lambda, S') = \lambda$. Note that for $x \neq \lambda$, $nextPivot(x, S') \neq x$.) Let $distToPivot(x, S')$ be the distance between x and $nextPivot(x, S')$ in list S. Furthermore, let $m(S, S') = \max_{x \in S} distToPivot(x, S')$.

The *modified list ranking problem* for S with respect to S' refers to the problem of determining for each $x \in S$ its next pivot $nextPivot(x, S')$ as well as the distance $distToPivot(x, S')$. The input/output structure for the modified list ranking problem is the same as for the list ranking problem.

Algorithm 1

(1) Select a set $S' \subset S$ of $\tilde{O}(\frac{n}{p})$ random pivots as follows: Every processor P_i makes for each $x \in S$ stored at P_i an independent biased coin flip which selects x as a pivot with probability $\frac{1}{p}$.

(2) All processors solve collectively the *modified list ranking problem* for S with respect to S'.

(3) Using an all-to-all broadcast, the values $nextPivot(x, S')$ and $distToPivot(x, S')$ for all pivots $x \in S'$ are broadcast to all processors.

(4) Using the data received in Step 3, each processor P_i can solve the list ranking problem for the nodes stored at P_i sequentially in time $\tilde{O}(\frac{n}{p})$.

For the correctness of Step 1, we recall the following

Lemma 3 *[28] Consider a random variable X with binomial distribution. Let n be the number of trials, each of which is successful with probability q. The expectation of X is $E(X) = nq$, $Prob\{X > cnq\} \leq e^{-\frac{1}{2}(c-1)^2 nq}$, for any $c > 1$.*

In order to implement Step 2, we simply simulate the standard recursive doubling (also called pointer jumping) technique. From Theorem 1 it follows that, with high probability, $m(S, S') \leq 3p\ln(n)$. Hence, Step 2 requires, with high probability, at most $\log(3p\ln(n)) = \log(3p) + \log\ln(n)$ communication rounds. Step 3 requires 1 communication round, and Step 4 is straightforward. In summary, we obtain

Theorem 2 *Algorithm 1 solves the list ranking problem using, with high probability, at most $1 + \log(3p) + \log\ln(n)$ communication rounds and $\tilde{O}(\frac{n}{p})$ local computation.*

3.1.2 Improving the Maximum Sublist Size

We now present an improved algorithm that solves the list ranking problem by using, with high probability, only $r \le (4k+6)\log(\frac{2}{3}p)+8$ communication rounds and $\tilde{\mathrm{O}}(\frac{n}{p})$ local computation where $k := \min\{i \ge 0 | \ln^{(i+1)} n \le (\frac{2}{3}p)^{2i+1}\}$.

Note that $k < \ln^*(n)$ is an extremely small number (see Table 1).

$p =$	4	8	16	32	64	128	256	512	1024	2048
n	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$	$k;R$
10^{10}	1;18	0;26	0;32	0;38	0;44	0;50	0;56	0;62	0;68	0;74
10^{100}	1;18	1;38	0;32	0;38	0;44	0;50	0;56	0;62	0;68	0;74
10^{1000}	1;18	1;38	1;48	0;38	0;44	0;50	0;56	0;62	0;68	0;74
$10^{(10^4)}$	1;18	1;38	1;48	1;58	0;44	0;50	0;56	0;62	0;68	0;74
$10^{(10^5)}$	1;18	1;38	1;48	1;58	1;68	0;50	0;56	0;62	0;68	0;74
$10^{(10^6)}$	1;18	1;38	1;48	1;58	1;68	1;78	0;56	0;62	0;68	0;74
$10^{(10^7)}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	0;62	0;68	0;74
$10^{(10^8)}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	1;98	0;68	0;74
$10^{(10^9)}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	0;74
$10^{(10^{10})}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{11})}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{12})}$	1;18	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{14})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{16})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{18})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{20})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{30})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{40})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{50})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{60})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{70})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{80})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{90})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118
$10^{(10^{100})}$	2;22	1;38	1;48	1;58	1;68	1;78	1;88	1;98	1;108	1;118

Table 1: Values of k and $R := (4k+6)\log(\frac{2}{3}p)+8$ [upper bound on r] for various combinations of n and p.

The basic idea of the algorithm is that any two pivots should not be closer than $O(p)$ because this creates large "gaps" elsewhere in the list. If two pivots are closer than $O(p)$, then one of them is "useless" and should be "relocated". The non-trivial part is to perform the "relocation" without too much overhead and such that the new set of pivots has a considerably better distribution. The algorithm uses three colors to mark nodes: *black* (pivot), *red* (a node close to a pivot), and *white* (all other nodes).

Algorithm 2

(1) Perform Step 1 of Algorithm 1. Mark all selected pivots *black* and all other nodes *white.*

(2) For $i = 1, \ldots, k$ do

(2a) For each *black* node x, all nodes which are to the right of x (in list S) and have distance at most $\frac{2}{3}p$ are marked *red.* Note: previously *black* nodes (pivots) that are now marked *red* are no longer considered pivots.

(2b) For each *black* node x, all nodes which are to the left of x (in list S) and have distance at most $\frac{2}{3}p$ are marked *red.*

(2c) Every processor P_i makes for each *white* node $x \in S$ stored at P_i an independent biased coin flip which selects x as a new pivot, and marks it *black*, with probability $\frac{1}{p}$.

(2d) Every processor P_i marks *white* every *red* node $x \in S$ stored at P_i.

(3) Let $S' \in S$ be the subset of *black* nodes obtained after Step 2. Continue with Steps 2 – 4 of Algorithm 1.

Observe that Steps 2a and 2b have to be performed in a left-to-right scan, respectively, as if executed sequentially. We can simulate this sequential scanning process in the parallel setting because the number of pivots is bounded by n/p. For Step 2a, we build linked lists of pivots by computing for each of them a pointer to the next pivot of distance at most $2\ p/3$, if any, and the distance. These linked lists of pivots are compressed into one processor and we run on these lists a sequential left-to-right scan to mark pivots red. We return the pivots to their original location and mark every non-pivot red for which there exists a non-red pivot that attempts to mark it red. Step 2b is performed analogously.

Let r be the number of communication rounds required by Algorithm 2. We will now show that, with high probability, $r \leq (4k+6)\log(\frac{2}{3}p) + 8 = \tilde{\mathrm{O}}(k \log p)$.

Let n_i be the maximum length of a contiguous sequence of *white* nodes after the i^{th} execution of Step 2b, and define $n_0 = n$. Let S_i be the set of *black* nodes after the i^{th} execution of Step 2c, $1 \leq i \leq k$, and let S_0 be the set of *black* nodes after the execution of Step 1. Note that, in Step 3, $S' = S_k$. Define $m_i = m(S_i)$ for $0 \leq i \leq k$.

Lemma 4 *With high probability, the following holds:*
(a) $n_0 = n$ *and* $n_i \leq 3p\ln(n_{i-1}), 1 \leq i \leq k$
(b) $m_i \leq 3p\ln(n_i), 0 \leq i \leq k$

Proof. It follows from Theorem 1 that, with high probability, $n_0 = n$ and $m_0 \leq 3p\ln(n)$ and, for a fixed $1 \leq i \leq k$ $n_i \leq m_{i-1}$ and $m_i \leq 3p\ln(n_i)$.
Since $k \leq \ln^*(n)$ and $\log^*(n)\frac{1}{n^c} \leq \frac{1}{n^{c-\epsilon}}, \epsilon > 0$, the above bounds for n_i and m_i hold, with high probability, for all $1 \leq i \leq k$. □

Lemma 5 *With high probability, for all* $1 \leq i \leq k$,
(a) $n_i \leq 3p(2\ln(3p) + \ln^{(i)}(n))$
(b) $m_i \leq 6p\ln(3p) + 3p\ln^{(i+1)}(n)$

Proof.
(a) Applying Lemma 4 we observe that

$$\begin{aligned}
n_1 &\leq 3p\ln(n) \\
n_2 &\leq 3p\ln(3p\ln(n)) = 3p(\ln(3p) + \ln\ln(n)) \\
n_3 &\leq 3p\ln(n_2) \leq 3p(\ln(3p) + \ln(\ln(3p) + \ln\ln(n))) \\
&\leq 3p(\ln(3p) + \ln\ln(3p) + \ln\ln\ln(n)) \\
n_4 &\leq 3p\ln(n_3) \leq 3p(\ln(3p) + \ln\ln(3p) + \ln\ln\ln(3p) + \ln\ln\ln\ln(n)) \\
&\vdots \\
n_i &\leq 3p(2\ln(3p) + \ln^{(i)}(n))
\end{aligned}$$

(b) It follows from Lemma 4 that $m_i \leq 3p\ln(n_i) \leq 3p\ln(3p(2\ln(3p) + \ln^{(i)}(n))) \leq 3p(\ln(3p) + \ln(2) + ln^{(2)}(3p) + ln^{(i+1)}(n)) \leq 6p\ln(3p) + 3p\ln^{(i+1)}(n)$. $\square$

Theorem 3 *With high probability, Algorithm 2 solves the list ranking problem with $r \leq (4k+6)\log(\frac{2}{3}p) + 8 = \tilde{O}(k\log p)$ communication rounds and $\tilde{O}(\frac{n}{p})$ local computation.*

Proof. With high probability, the total number of communication rounds in Algorithm 2 is bounded by $2k\log(\frac{2}{3}p) + \log(m_k) + 1$
$\leq 2k\log(\frac{2}{3}p) + \log(6p) + \log\ln(3p) + \log(3p) + \log\ln^{(k+1)}(n) + 1$
$\leq (2k+3)\log(\frac{2}{3}p) + \log 9 + \log 4.5 + \log\ln^{(k+1)}(n) + 1$
$\leq (2k+3)\log(\frac{2}{3}p) + \log\ln^{(k+1)}(n) + 8$
$\leq \log((\frac{2}{3}p)^{2k+3}) + \log\ln^{(k+1)}(n) + 8 \leq 2\log((\frac{2}{3}p)^{2k+3}) + 8$
[if (*) $\ln^{(k+1)}(n) \leq (\frac{2}{3}p)^{2k+3}$] $\leq (4k+6)\log(\frac{2}{3}p) + 8 = \tilde{\mathrm{O}}(k\log p)$
Condition (*) is true because we selected $k = \min\{i \geq 0 | \ln^{(i+1)} n \leq (\frac{2}{3}p)^{2i+1}\}$. Note that this bound is not tight. $\square$

3.1.3 Simulation and Experimental Results

We simulated the behavior of Algorithm 2. In particular, we simulated how the above method improves the sample by reducing the maximum distance, m_i, between subsequent pivots. We examined the range of $4 \leq p \leq 2048$ and $100,000 \leq n \leq 1,500,000$ as shown in Table 2 and applied Algorithm 2 for each n, p combination shown 100 times with different random samples. Table 2 shows the values of k and the upper bound R on the number of communication rounds required according to Theorem 3. We then measured the maximum distance, m_k^{obs}, observed between two subsequent pivots in the sample chosen at the end of the algorithm, as well as the number, r^{obs}, of communication rounds actually required. Each of the numbers shown is the worst case observed in the respective 100 test runs.

According to Theorem 3, for the range of test data used, the number of communication rounds in our algorithm should not exceed 78. This is an upper bound, though. The actual number of communication rounds observed in Table 2 is 25 in the worst case. The number of rounds observed is usually around 30% of the upper bound according to Theorem 3. We also observe that for a given p (i.e. in a vertical column), the values of m_k^{obs} and r^{obs} are essentially stable and show no monotone increase or decrease with increasing n.

$p=$	4	8	16	32	64	128	256	512	1024	2048
n	k	k	k	k	k	k	k	k	k	k
	R	R	R	R	R	R	R	R	R	R
	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}	m_k^{obs}
	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}	r^{obs}
100,000	1	1	1	1	1	0	0	0	0	0
	18	38	48	58	68	50	56	62	68	74
	28	59	119	238	409	1400	2421	5900	9136	17158
	8	13	16	19	22	12	13	14	15	16
500,000	1	1	1	1	1	1	0	0	0	0
	18	38	48	58	68	78	56	62	68	74
	32	72	117	264	474	860	3150	6144	11179	21552
	8	14	16	20	22	25	13	14	15	16
1,000,000	1	1	1	1	1	1	0	0	0	0
	18	38	48	58	68	78	56	62	68	74
	40	69	127	264	440	851	3406	7924	11861	21552
	9	14	16	20	22	25	13	14	15	16
1,500,000	1	1	1	1	1	1	0	0	0	0
	18	38	48	58	68	78	56	62	68	74
	33	89	172	270	551	903	3893	6120	11938	23631
	9	14	17	20	23	25	13	14	15	16

Table 2: k, $R := (4k+6)\log(\frac{2}{3}p)+8$, m_k^{obs} and r^{obs} For various combinations of n and p, where m_k^{obs} and r^{obs} are the observed *worst case* values of m_k and r, respectively. (For each shown combination of n and p, the m_k^{obs} and r^{obs} shown are the worst case values observed during 100 test runs.)

NProc/n	4	8	16	64	128	256	512	1024	2048	4096	16384	65536	131072	262144
4	5	6	7	7	8	8	8	9	9	9	9	10	9	9
8	-	6	7	8	9	10	10	10	10	10	10	10	10	11

Table 3: No. of communication rounds on the PowerXplorer (worst values in 5 runs.)

In Table 3 we show the actual number of communication rounds needed by Algorithm 1 on the Parsytec PowerXplorer machine, with 16 nodes (each with a PowerPC601 processor and a T805 transputer).

3.1.4 Applications

The problem of list ranking is a special case of computing the suffix sums of the elements of a linked list. The above algorithm can obviously be generalized to compute prefix or suffix sums for associative operators. List ranking is a very popular tool for obtaining numerous parallel tree and graph algorithms [6, 7]. An important application outlined in [6] is to use list ranking for applying Euler tour techniques to tree problems: for an undirected forest of trees, rooting every tree at a given vertex chosen as root, determining the parent of each vertex in the rooted forest, computing the preorder (or postorder) traversal of the forest, computing the level of each vertex, and computing the number of descendants of each vertex. All these problems can be easily solved with one or a small constant number of list ranking operations.

3.1.5 Conclusion

We presented a randomized parallel list ranking algorithm for distributed memory multiprocessors, using the coarse grained multicomputer model. The algorithm requires, with high probability, $r \leq (4k+6)\log(\frac{2}{3}p) + 8 = \tilde{O}(k \log p)$ communication rounds. For all practical purposes, $k \leq 2$. Therefore, we expect that our result will have considerable practical relevance.

3.2 A Deterministic Algorithm for List Ranking

We revisit the important list ranking problem. Reid-Miller's [31] presented an empirical study of parallel list ranking for the Cray C-90. The paper followed essentially the CGM/BSP model and claimed that this was the fastest list ranking implementation so far. The algorithm in [31] required $O(\log n)$ communication rounds. In [18], an improved algorithm was presented which required, with high probability, only $O(k \log p)$ rounds, where $k \leq \log^* n$. In [19], $O(\log p)$ communication rounds are achieved by a randomized algorithm.

We improve these results by giving the first *deterministic* algorithm for list ranking using $O(\log p)$ rounds. This improvement is an important step towards the ultimate goal, a deterministic algorithm with only $O(1)$ communication rounds. In fact, it is an open problem whether this is possible for these graph problems. In contrast to the previous deterministic results, the improved number of communication rounds obtained, $O(\log p)$, is *independent* of n and grows only very slowly with respect to p. Hence, for most practical purposes, the number of communication rounds can be considered as constant. We expect that this will be of considerable practical relevance.

As in [31] we will, in general, assume that $n >> p$ (coarse grained), because this is usually the case in practice. Note, however, that our results for the list ranking problems hold for arbitrary ratios $\frac{n}{p}$. Goodrich [21] calls such algorithms *fully scalable*.

3.2.1 The Basic Idea

Let L be a list represented by a vector s such that $s[i]$ is the node following i in the list L. The last element l of the list L is the one with $s[l] = l$. The distance between i and j, $d_L(i,j)$, is the number of nodes between i and j plus 1 (i.e. the distance is 0 iff $i = j$, and it is one if and only if one node follows the other). The list ranking problem consists of computing for each $i \in L$ the distance between i and l, referred to as $rank_L(i) = d_L(i,l)$.

For our algorithm, we need the following definitions. A *r-ruling set* is defined as a subset of selected list elements that has the following properties: (1) No two neighboring elements are selected. (2) The distance of any unselected element to the next selected element is at most r.

An overview of our CGM list ranking algorithm is as follows. First, we compute a $O(p^2)$-ruling set R with $|R| = O(n/p)$ and broadcast R to all processors. More precisely, the $O(p^2)$-ruling set R is represented as a linked list where each element i is assigned a pointer to the next element j of R with respect to the order implied by L as well as the distance between i and j in L. Then, every processor sequentially performs a list ranking of R, computing for each $i \in R$ its distance to the last element of L. All other list elements have at most distance $O(p^2)$ from the next element of R in the list. Their distance is determined by simulating standard PRAM pointer jumping until the next element of R is reached.

All steps, except for the computation of the $O(p^2)$-ruling set R, can be easily implemented in $O(\log p)$ communication rounds.

In the remainder of this section we introduce a new technique, called *deterministic list compression*, which will allows us to compute a $O(p^2)$-ruling set in $O(\log p)$ communication rounds.

The basic idea behind *deterministic list compression* is to have an alternating sequence of *compress* and *concatenate* phases. In a compress phase, we select a subset of list elements, and in a *concatenate* phase we use pointer jumping to work our way towards building a linked list of selected elements.

For the compress phase, we apply the *deterministic coin tossing* technique of [14] but with a different set of labels. Instead of the memory address used in [14], we use the number of the processor storing list item i as its label $l(i)$. During the computation, we select sequentially the elements of R in the sublists of subsequent nodes in L which are stored at the same processor. The term "subsequent" refers to successor with respect to the *current* value of s.

Note that there are at most p different labels, and subsequent nodes in those parts of L that are not processed sequentially have different labels. We call list element $s[i]$ a *local maximum* if $l(i) < l(s[i]) > l(s[s[i]])$. We apply deterministic coin tossing to those parts of L that are not processed sequentially.

The naïve approach of applying this procedure $O(\log p)$ times would yield a $O(p^2)$-ruling set, but unfortunately it would require more than $O(\log p)$ communication rounds. Note that, when we want to apply it for a second, third, etc. time, the elements selected previously need to be linked by pointers. Since two subsequent elements selected by deterministic coin tossing can have distance $O(p)$, this may require $O(\log p)$ communication rounds, each. Hence, this straightforward approach requires a total of $O(\log^2 p)$ communication rounds.

Notice, however, that if two selected elements are at distance $\Theta(p)$ at a given moment, then it is unnecessary to further apply deterministic coin tossing in order to reduce the number of selected elements. The basic approach of our algorithm is therefore to interleave pointer jumping and deterministic coin tossing operations with respect to our new labeling scheme. More precisely, we will have only one pointer jumping step between subsequent deterministic coin tossing steps, and such pointer jumping operations will not be applied to those list elements that are pointing to selected elements.

This concludes the high level overview of our *deterministic list compression* techniques. The following describes the algorithm in detail.

Algorithm 3 CGM Algorithm for computing a p^2-ruling set.

Input: A linked list L and a vector s where $s[i]$ is the node following i in the list L. L and s are stored on a p processor CGM with total $O(n)$ memory.

Output: A set of selected nodes of L (which is a p^2-ruling set).

(1) Mark all list elements as *not selected.*

(2) FOR EVERY list element i IN PARALLEL:
IF $l(i) < l(s[i]) > l(s[s[i]])$ THEN mark $s[i]$ as *selected.*

(3) Sequentially, at each processor, process the sublists of subsequent list elements which are stored at the same processor. For each such sublist, mark every second element as *selected.* If a sublist has only two elements, and not both neighbors have a smaller label, then mark both elements of the sublist as *not selected.*

(4) FOR $k = 1 \ldots \log p$ DO

(4.1) FOR EVERY list element i IN PARALLEL:
IF $s[i]$ is not selected THEN set $s[i] := s[s[i]]$.

(4.2) FOR EVERY list element i IN PARALLEL:
IF (i, $s[i]$ and $s[s[i]]$ are selected) AND NOT ($l(i) < l(s[i]) > l(s[s[i]])$) AND ($l(i) \neq l(s[i])$) AND ($l(s[i]) \neq l(s[s[i]])$) THEN mark $s[i]$ as *not selected*.

(4.3) Sequentially, at each processor, process the sublists of subsequent selected list elements which are stored at the same processor. For each such sublist, mark every second selected element as *not selected*. If a sublist has only two elements, and not both neighbors have a smaller label, then mark both elements of the sublist as *not selected*.

(5) Select the last element of L.

We first prove that the set of elements selected at the end of Algorithm 3 is of size at most $O(n/p)$.

Lemma 6 *After the k^{th} iteration in Step 4, there are no more than two selected elements among any 2^k subsequent elements of the original list L.*

We now prove that subsequent elements selected at the end of Algorithm 3 have distance at most $O(p^2)$. We first need the following.

Lemma 7 *After every execution of Step 4.3, the distance of two subsequent selected elements with respect to the current pointers (represented by vector s) is at most $O(p)$.*

Lemma 8 *After the k-th execution of Step 4.3, two subsequent elements with respect to the current pointers (represented by vector s) have distance $O(2^k)$ with respect to the original list L.*

Lemma 9 *No two subsequent selected elements have a distance of more than $O(p^2)$ with respect to the original list L.*

In summary, we obtain

Theorem 4 *The list ranking problem for a linked list with n vertices can be solved on a CGM with p processors and $O(\frac{n}{p})$ local memory per processor using $O(\log p)$ communication rounds and $O(\frac{n}{p})$ local computation per round.*

The proofs can be found in [12] and will be omitted here.

4 Connected Components and Spanning Forest

Consider an undirected graph $G = (V, E)$ with n vertices and m edges. Each vertex $v \in V$ has a unique label between 1 and n. Two vertices u and v are connected if there is an undirected path of edges from u to v. A connected subset of vertices is a subset of vertices where each pair of vertices is connected. A *connected component* of G is defined as a maximal connected subset.

Bäumker and Dittrich [8] present a randomized connected components algorithm for planar graphs using $O(\log p)$ communication rounds. They suggest an extension of this algorithm for general graphs with the same number of communication rounds.

In this section, we study the problem of computing the connected components of G on a CGM with p processors and $O(\frac{n+m}{p})$ local memory per processor. We improve the previous result by giving the first *deterministic* algorithm for connected components problem using $O(\log p)$ rounds.

We introduce a new technique, called *clipping*, which refers to the idea of taking a PRAM algorithm for the same problem but running it for only $O(\log p)$ rounds and then finishing the computation with some other $O(\log p)$ rounds CGM algorithm. (See also JáJá's *accelerated cascading* technique for the PRAM [23].)

Steps 1 and 2 of Algorithm 4 simulate Shiloch and Vishkin's PRAM algorithm [34], but for $\log p$ phases only. Each vertex v has a pointer to a vertex $parent(v)$ such that the $parent(v)$ pointers always form trees. The trees are also referred to as a *supervertices*. A tree of height one is called a star. An edge (u, v) is *live* if $parent(u) \neq parent(v)$. Shiloch and Vishkin's PRAM algorithm merges supervertices along live edges until they equal the connected components. When simulated on a CGM or BSP computer, Shiloch and Vishkin's PRAM algorithm results in $\log n$ communication rounds or supersteps, respectively.

Our CGM algorithm requires $O(\log p)$ rounds only. It simulates only the first $\log p$ iterations of the main loop in the PRAM algorithm by Shiloch and Vishkin and then completes the computation in another $\log p$ communication rounds (Steps 3 - 7).

Algorithm 4 CGM Algorithm for Connected Component Computation

Input: An undirected graph $G = (V, E)$ with n vertices and m edges stored on a p processor CGM with total $O(n + m)$ memory.

Output: The connected components of G represented by the the values $parent(v)$ for all vertices $v \in V$.

(1) FOR all $v \in V$ IN PARALLEL DO $parent(v) := v$.

(2) FOR $k := 1$ to $\log p$ DO

(2.1) FOR all $v \in V$ IN PARALLEL DO $parent(v) := parent(parent(v))$.

(2.2) FOR every live edge (u, v) IN PARALLEL DO (simulating concurrent write)

(a) IF $parent(parent(v)) = parent(v)$ AND $parent(parent(u)) = parent(u)$ THEN { IF $parent(u) > parent(v)$ THEN $parent(parent(u)) := parent(v)$ ELSE $parent(parent(v)) := parent(u)$ }

(b) IF $parent(u) = parent(parent(u))$ AND $parent(u)$ did not get new links in steps 2.1 and 2.2(a) THEN $parent(parent(u)) := parent(v)$

(c) IF $parent(v) = parent(parent(v))$ AND $parent(v)$ did not get new links in steps 2.1 and 2.2.1 THEN $parent(parent(v)) := parent(u)$

(2.3) FOR all $v \in V$ IN PARALLEL DO $parent(v) := parent(parent(v))$.

(3) Use the Euler Tour algorithm in [12] to convert all trees into stars. For each $v \in V$, set $parent(v)$ to be the root of the star containing v. Let $G' = (V', E')$ be the graph consisting of the supervertices and live edges obtained. Distribute G' such that each processor stores the entire set V' and a subset of $\frac{m}{p}$ edges of E'. Let E_i be the edges stored at processor i, $0 \leq i \leq p - 1$.

(4) Mark all processors as *active*.

(5) FOR $k := 1$ to $\log p$ DO

(5.1) Partition the active processors into groups of size two.

(5.2) FOR each group P_i, P_j of active processors, $i < j$ IN PARALLEL DO

(a) processor P_j sends it's edge set E_j to processor P_i.

(b) processor P_j is marked as *passive*.

(c) processor P_i computes the spanning forest (V', E_s) of the graph $SF = (V', E_i \cup E_j)$ and sets $E_i := E_s$.

(6) Mark all processors as *active* and broadcast E_0.

(7) Each processor i computes sequentially the connected components of the graph $G'' = (V', E_0)$. For each vertex v of V' let $parent'(v)$ be the smallest label $parent(w)$ of a vertex $w \in V'$ which is in the same connected component with respect to $G'' = (V', E_0)$. For each vertex $u \in V$ stored at processor P_i set $parent(u) := parent'(parent(u))$. (Note that $parent(u) \in V'$.)

Lemma 10 [34] *The number of different trees after iteration k of Step 2 is bounded by $(\frac{2}{3})^k n$.*

We obtain

Theorem 5 *Algorithm 4 computes the connected components and spanning forest of a graph $G = (V, E)$ with n vertices and m edges on a CGM with p processors and $O(\frac{n+m}{p})$ local memory per processor, $\frac{n+m}{p} \geq p^\epsilon$ ($\epsilon > 0$), using $O(\log p)$ communication rounds and $O(\frac{n+m}{p})$ local computation per round.*

Proof. Steps 1 and 2 simulate the PRAM algorithm by Shiloch and Vishkin [34] but with only $\log p$ iterations of the main loop in Step 2 instead of the $O(\log n)$ iterations in Shiloch and Vishkin's original algorithm. Step 3 converts all trees into stars. It follows from Lemma 10 that the graph $G' = (V', E')$ has at most $O(\frac{n}{p})$ vertices. Hence, V' can be broadcast to all processors. E' can still be of size $O(m)$ and is distributed over the p processors. Recall that E_i refers to the edges in E' stored at processor i. We note that the spanning forest of $(V', E_i \cup E_j)$ for two sets E_i, E_j is of size $O(|V'|) = O(\frac{n}{p})$. Hence, each of the spanning forests computed in Step 5 can be stored in the local memory of a single processor. Steps 3-6 merge pairs of spanning forests until, after $\log p$ rounds, the spanning forest of G' is stored at processor P_0. In Step 7, all processors update the partial connected component information obtained in Step 2.

The PRAM simulations in Steps 2.1 - 2.3 require $O(1)$ rounds, each.

Hence, the entire algorithm requires $O(\log p)$ communication rounds. The local computation per round is obviously bounded by $O(\frac{n+m}{p})$.

We observe that the algorithm is easily extended to report the entire spanning forest for G within the same time bounds. □

5 Parallel Range Minima

Given an array of n real numbers $A = (a_1, a_2, \ldots, a_n)$, define $MIN(i, j) = \min\{a_i, \ldots, a_j\}$. The range minima problem consists of preprocessing array A such that queries $MIN(i, j)$, for any $1 \leq i \leq j \leq n$, can be answered in constant time.

We present an algorithm, under the CGM(n,p) model, that solves the problem requiring a constant number of communication rounds and $O(\frac{n}{p})$ computation time. In the PRAM model, there is an optimal algorithm of $O(\log n)$ time [23]. This algorithm, however, is not immediately transformable to the CGM model. Berkman et al. [10] describe an optimal algorithm of $O(\log\log n)$ time for a PRAM CRCW. This algorithm motivates the CGM algorithm presented in this paper.

Range minima is a basic problem and appears in other problems. In [23], range minima is used in the design of a PRAM algorithm for the problem of Lowest Common Ancestor (LCA). This algorithm uses twice the Euler-tour problem. The idea of this algorithm can be used in the CGM model, by using the same sequence of steps. In [12], the Euler-tour problem in the CGM model is discussed. Its implementation reduces to the list ranking problem, that uses $O(\log p)$ communication rounds in the CGM model [12]. Applications of the LCA problem, on the other hand, are found in graph problems (see [32]) and in the solution of some other problems. Besides the LCA problem, the range minima problem is used in several other applications [3, 20, 29].

Definition 1 *Consider an array $A = (a_1, a_2, \ldots, a_n)$ of n real numbers. The prefix minimum array is the array $P = (\min\{a_1\}, \min\{a_1, a_2\}, \ldots, \min\{a_1, a_2, \ldots, a_n\})$. The suffix minimum array is the array $S = (\min\{a_1, a_2, \ldots, a_n\}, \min\{a_2, \ldots, a_n\}, \ldots, \min\{a_{n-1}, a_n\}, \min\{a_n\})$.*

Definition 2 *The lowest common ancestor of two vertices u and v of a rooted tree is the vertex w that is an ancestor of u and v and that is farthest from the root. We denote $w = LCA(u, v)$.*

Definition 3 *The lowest common ancestor problem consists of preprocessing a rooted tree such that queries of the type $LCA(u, v)$, for any vertices u and v of the tree, can be answered in constant sequential time.*

5.1 Sequential Algorithms

The algorithm for the range minima problem, in the CGM model, uses two sequential algorithms that are executed using local data in each processor. These algorithms are described in the following.

5.1.1 Algorithm of Gabow et al.

This sequential algorithm was designed by Gabow et al. [20] and uses the *Cartesian tree* data structure, introduced in [36]. An example of Cartesian tree is given in figure 2. In this algorithm we need also an algorithm for the lowest common ancestor problem.

Definition 4 *Given an array $A = (a_1, a_2, \ldots, a_n)$ of n distinct real numbers, the Cartesian tree of A is a binary tree whose nodes have as label the values of array A. The root of the tree has as label $a_m = \min\{a_1, a_2, \ldots, a_n\}$. Its left subtree is a Cartesian tree for $A_{1,m-1} = (a_1, a_2, \ldots, a_{m-1})$, and its right subtree is a Cartesian tree for $A_{m+1,n} = (a_{m+1}, \ldots, a_n)$. The Cartesian tree for an empty array is the empty tree.*

Algorithm 5 Range Minima (Gabow et al.)

1. Build a Cartesian tree for A.

2. Use a linear sequential algorithm for the LCAproblem using the Cartesian tree.

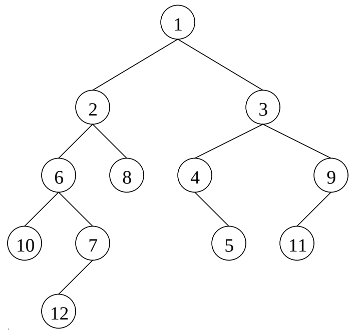

Fig. 2: Cartesian tree corresponding to the array $(10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9)$.

The construction of the Cartesian tree takes linear time [20]. Linear sequential algorithms for the LCA problem can be found in [9, 22, 33]. Thus, any query $MIN(i, j)$ is answered as follows. From the recursive definition of the Cartesian tree, the value of $MIN(i, j)$ is the value of the LCA of a_i and a_j. Thus, each range minima query can be answered in constant time through a query of LCA in the Cartesian tree. Therefore, the range minima problem is solved in linear time.

5.1.2 Sequential Algorithm of Alon and Schieber

In this section we describe the algorithm of Alon and Schieber [2] of $O(n \log n)$ time. In spite of its non linear complexity, this algorithm is important in the description of the CGM algorithm, as will be seen in section 5.2. Without loss of generality, we consider n to be a power of 2.

Algorithm 6 Range Minima (Alon and Schieber)

1. Construct a complete binary tree T with n leaves.
2. Associate the elements of A to the leaves of T.
3. For each vertex v of T we compute the arrays P_v and S_v, the prefix minimum and the suffix minimum arrays, respectively, of the elements of the leaves of the subtree with root v.

Tree T constructed by algorithm 6 will be called *PS-tree*. Figure 3 illustrates the execution of this algorithm. Queries of type $MIN(i, j)$ are answered as follows. To determine $MIN(i, j)$, $1 \leq i \leq j \leq n$, we find $w = LCA(a_i, a_j)$ in T. Let v and u be the left and right sons of w, respectively. Then, $MIN(i, j)$ is the minimum between the value of S_v in the position corresponding to a_i and the value of P_u in the position corresponding to a_j. PS-tree is a complete binary tree. In [23, 32] it is shown that a LCA query in complete binary trees can be answered in constant time.

5.2 The CGM Algorithm for the Range Minima Problem

From the sequential algorithms seen in the previous section, we can design a CGM algorithm for the range minima problem. This algorithm is executed in $O(\frac{n}{p})$ time and uses $O(1)$ communication rounds. The major difficulty is how to store the required data structure among the processors so that the queries can be done in constant time, without violating the limit of $O(\frac{n}{p})$ memory, required by the CGM model.

Given an array $A = (a_1, a_2, \ldots, a_n)$, we write $A[i] = a_i$, $1 \leq i \leq n$, and $A[i \ldots j] =$ subarray $(a_i, \ldots, a_j)$, $1 \leq i \leq j \leq n$.

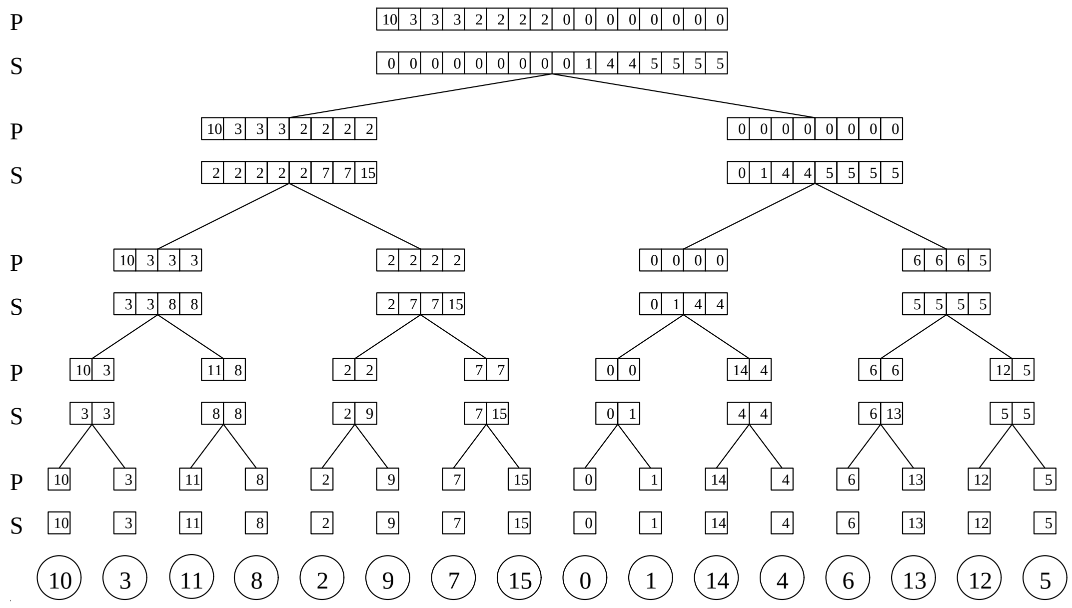

Fig. 3: PS-tree generated by the algorithm of Alon and Schieber for the array $(10, 3, 11, 8, 2, 9, 7, 15, 0, 1, 14, 4, 6, 13, 12, 5)$.

The idea of the algorithm is based on how queries $MIN(i, j)$ can be answered. Each processor stores $\frac{n}{p}$ contiguous positions of the input array. Thus, given $A = (a_1, a_2, \ldots, a_n)$, we have subarrays $A_i = (a_{i\frac{n}{p}+1}, \ldots, a_{(i+1)\frac{n}{p}})$, for $0 \leq i \leq p-1$. Depending on the location of a_i and a_j in the processors, we have the following cases:

1. if a_i and a_j are in a same processor, the problem domain reduces to the subarray stored in that processor. Thus, we need a data structure to answer this type of queries in constant time. This data structure is obtained in each processor by Algorithm 5 (section 5.1.1).

2. if a_i and a_j are in distinct processors $p_{\bar{i}}$ e $p_{\bar{j}}$ (without loss of generality, $\bar{i} < \bar{j}$), respectively, we have two subcases:

   (a) if $\bar{i} = \bar{j} - 1$, a_i and a_j are in neighbor processors, $MIN(i, j)$ corresponds to the minimum between the minimum of a_i through the end of array $A_{\bar{i}}$ and the minimum of the beginning of $A_{\bar{j}}$ to a_j. These minima can be determined by Algorithm 5. To determine the minimum of the minima we need one communication round.

   (b) if $\bar{i} < \bar{j} - 1$, $MIN(i, j)$ corresponds to the minimum among the minimum of subarray $A_{\bar{i}}[i - \bar{i}\frac{n}{p} \ldots (i+1)\frac{n}{p}]$, the minimum of subarray $A_{\bar{j}}[\bar{j}\frac{n}{p} + 1 \ldots j - \bar{j}\frac{n}{p}]$ and the minima of subarrays $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$. The first two minima are obtained as in the previous subcase. The minima of subarrays $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$ are easily obtained by using the Cartesian tree. The minimum among them corresponds to the range minima problem restricted to the array of minima of the data in the processors. Thus we need a data structure to answer these queries in constant time. As the array of minima contains only p values, this data structure can be obtained by Algorithm 6 (section 5.1.2).

The difficulty of case 2.b is that we cannot construct the PS-tree explicitly in each processor as described in section 5.1.2, since this would require a memory of size $O(p \log p)$, larger than the memory requirement in the CGM model, which is $O(\frac{n}{p})$, with $\frac{n}{p} \geq p$. To overcome this difficulty we construct arrays P' and S' of $\log p + 1$ positions each, described as follows, to store some partial information of the PS-tree T in each processor. Let us describe this construction in processor i, with $0 \leq i \leq p-1$. Let b_i be the value of the minimum of array A_i. Let v be any vertex of T such that the subtree with root v has b_i as leaf and let d_v be the *depth of* v in T, that is the path length from the root to v, as defined in [1]; and l_v be the *level of* v, which is the height of the tree minus the depth of v, as defined in [1] ($l_v = \log p - d_v$, because tree T has height $\log p$). Array P' (respectively, S') contains in position l_v the value of array P_v (respectively, S_v), of the level l_v of T, in the position corresponding to the leaf b_i. In other words, we have $P'[l_v] = P_v[i \bmod 2^{l_v} + 1]$.

Figure 4 illustrates the correspondence between arrays P' and S' stored in each processor and the PS-tree constructed by the algorithm of Alon and Schieber [2]. In Figure 4.b, the bold face positions in arrays S in each level of the tree correspond to the suffix minimum of $b_0 = 3$ in each level. In this way, we obtain array S' in P_0 (figure 4.c).

Algorithm 7 Range Minima (CGM model)

{Each processor receives $\frac{n}{p}$ contiguous positions of array A, partitioned into subarrays $A_0, A_1, \ldots, A_{p-1}$.}

1. Each processor i executes sequentially Algorithm 5 (section 5.1.1).

2. {Each processor constructs an array $B = (b_i)$ of size p, that contains the minimum of the data stored in each processor.}

   2.1. Each processor i calculates $b_i = \min A_i = \min\{a_{i\frac{n}{p}+1}, \ldots, a_{(i+1)\frac{n}{p}}\}$.

   2.2. Each processor i sends b_i to the other processors.

   2.3. Each processor i puts in b_k the value received from processor k, $k \in \{0, \ldots, p-1\} \setminus \{i\}$.

3. Each processor i executes procedures Construct_P' and Construct_S' (see below) .

   {By observing the description of Algorithm 6 (section 5.1.2), $P'[k]$ contains position i of the prefix minimum array in level k, $0 \leq k \leq \log p$.}

Given array B, the following procedure constructs array P' of $\log p + 1$ postitions in processor i, for $0 \leq i \leq p-1$. This procedure constructs P' in $O(p)$ $(= O(\frac{n}{p}))$ time using only local data. The construction of array S' is done in a symmetric way, considering array B in reverse order.

Procedure 1 *PROCEDURE Construct_P'.*

```
P'[0] ← b_i
pointer ← i
inorder ← 2 * i + 1
for k ← 1 until log p do
   P'[k] ← P'[k - 1]
   if ⌊inorder/2^k⌋ is odd
      then for l ← 1 until 2^(k-1) do
               pointer ← pointer - 1
               if P'[k] > B[pointer]
                  then P'[k] ← B[pointer]
```

To simplify the correctness proof of this procedure, we consider p to be a power of 2 and that, in each processor i, array B contains leaves of a complete binary tree, as in the description of the sequential algorithm of section 5.1.2. We do not store an entire array in each internal node of this tree, but only the $\log p + 1$ positions of the prefix minimum array in each level corresponding to position i of array B.

The value of the variable *inorder* of line 3 is the value of the in-order number of the leaf containing b_i, obtained in an in-order traversal of the tree nodes. It can easily be seen that these values, from the left to the right, are the odd numbers in the interval $[1, 2p-1]$.

Theorem 6 *Procedure 1 correctly calculates array P', for each processor i, $0 \le i \le p-1$.*

Proof. We give an idea of the proof. For a processor i, $0 \le i \le p-1$, we prove the following invariant at each iteration of the for loop of line 4: At each iteration k, let T_k be subtree that contains b_i and has root at level k. $P'[k]$ stores the position i of the minimum prefix array of the subarray of B corresponding to the leaves of subtree T_k and the variable *pointer* contains the index, in B, of the leftmost leaf of T_k. □

To determine S' we have a similar theorem with a similar proof.

Lemma 11 *Execution of Procedure 1 in each processor takes $O(\frac{n}{p})$ sequential time.*

Proof. The maximum number of iterations is $\log p$. In each iteration, the value of *pointer* is decremented or remains the same. The worst case is when $i = p-1$. In this case, at each iteration the value of the variable *pointer* is updated. As the number of elements of B is p, the algorithm updates the value of *pointer* at most p times. Therefore, procedure 1 is executed in $O(p) = O(\frac{n}{p})$ sequential time. □

Theorem 7 *Algorithm 7 solves the range minima problem in $O(\frac{n}{p})$ time using $O(1)$ communication rounds.*

Proof. Step 1 is executed in $O(\frac{n}{p})$ sequential time and does not use communication. Step 2 runs in $O(\frac{n}{p})$ sequential time and uses one communication round. By theorem 11, step 3 is executed in $O(\frac{n}{p})$ sequential time and does not require communication. Therefore, algorithm 7 solves the range minima problem in $O(\frac{n}{p})$ time using $O(1)$ communication rounds. □

5.3 Query Processing

In this section, we show how to use the output of algorithm 7 to answer queries $MIN(i,j)$ in constant time. A query $MIN(i,j)$ is handled as follows. Assume that i and j are known by all the processors and the result will be given by processor 0. If a_i and a_j are in a same processor, then $MIN(i,j)$ can be determined by step 1 of algorithm 7. Otherwise, suppose that a_i and a_j are in distinct processors $p_{\bar{i}}$ and $p_{\bar{j}}$, respectively, with $\bar{i} < \bar{j}$. Let $right(\bar{i})$ the index in A of the rightmost element in array $A_{\bar{i}}$, and $left(\bar{j})$ the index in A of the leftmost element in array $A_{\bar{j}}$. Calculate $MIN(i, right(\bar{i}))$ and $MIN(left(\bar{j}), j)$, using step 1. We have two cases:

1. If $\bar{j} = \bar{i} + 1$ then $MIN(i,j) = \min\{MIN(i, right(\bar{i})), MIN(left(\bar{j}), j)\}$.

2. If $\bar{i}+1 < \bar{j}$, then calculate $MIN(right(\bar{i})+1, left(\bar{j})-1)$, using step 3 of the algorithm. Notice that $MIN(right(\bar{i})+1, left(\bar{j})-1)$ corresponds to $\min\{b_{\bar{i}+1}, \ldots, b_{\bar{j}-1}\}$. Thus, $MIN(i,j) = \min\{MIN(i, right(\bar{i})), MIN(right(\bar{i})+1, left(\bar{j})-1), MIN(left(\bar{j}), j)\}$.

The value of $MIN(right(\bar{i})+1, left(\bar{j})-1)$ is obtained using step 3, as follows. Each processor calculates $w = LCA(b_{\bar{i}+1}, b_{\bar{j}-1})$ in constant time. Then determines the level l_w; determines v and u, the left and right sons of w, respectively; determines l, the index of the leftmost leaf in the subtree of u. Processor $\bar{i}+1$ calculates $S_v(2^{l_w-1} - l + \bar{i})$ and sends this value to processor 0. Processor $\bar{j}-1$ calculates $P_u(\bar{j}-l)$ and sends this value to processor 0. Processor 0, finally, calculates the minimum of the received minima. In both cases, processor 0 receives the minima of processors $\bar{i}$ and $\bar{j}$.

Notice finally that the correctness of Algorithm 7 results from the observations presented in this section.

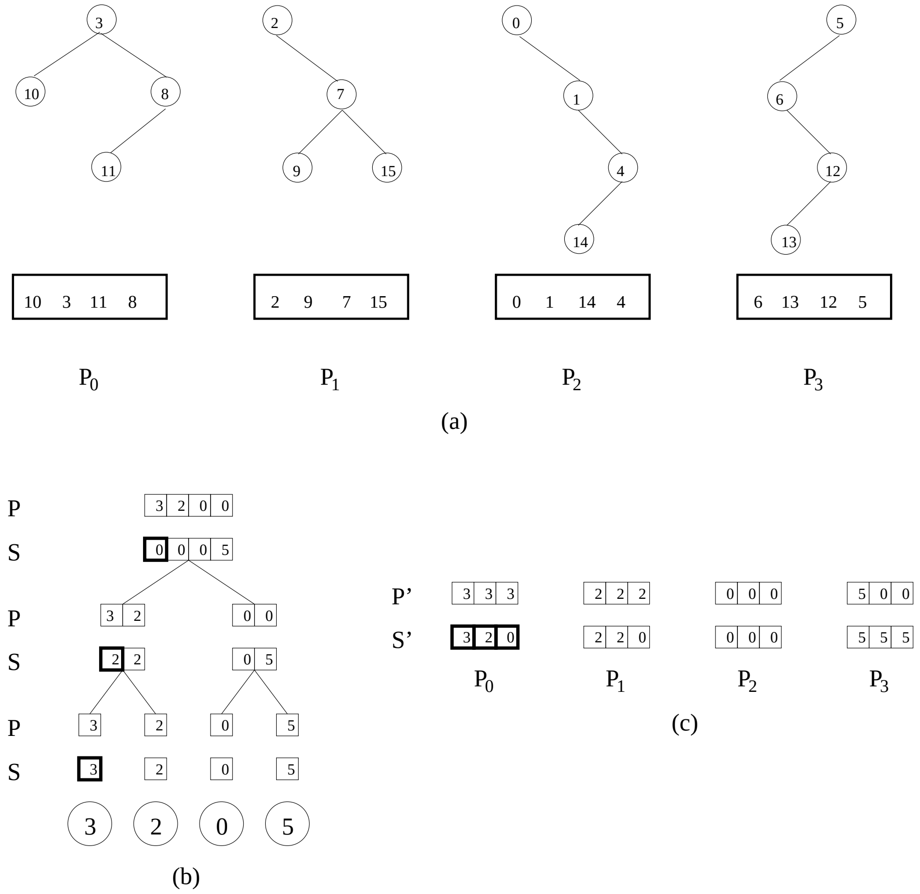

Fig. 4: Execution of algorithm 7 using the array $(10, 3, 11, 8, 2, 9, 7, 15, 0, 1, 14, 4, 6, 13, 12, 5)$. (a) The data distributed in the processors and the corresponding Cartesian trees. (b) PS-trees constructed by the algorithm of Alon and Schieber [2] of section 5.1.2 for the array $(3, 2, 0, 5)$ of the minima of processors. (c) Arrays P' and S' constructed by step 3 of algorithm 7 corresponding to arrays P and S of T.

5.4 Experimental Results

The proposed algorithm has been implemented on a distributed memory parallel machine, the Parsytec PowerXplorer 16/32 Parallel Computing System, with 16 nodes under a two-dimensional topology. Each node consists of one PowerPC601 application processor and one T805 communication processor, and 32 MBytes of local memory. The implementation is done in PVM (parallel virtual machine). We use the interface PowerPVM that implements PVM 3.2. Compared to the implementation with the native system called PARIX, we observe very similar performance results. PVM is chosen for portability.

For the experiments we use the input size $n = 2^k$, with $k = 5, \ldots, 19$, and $p = 2^j$ processors, with $j = 0, \ldots, 4$. The inputs consist of integers randomly distributed among the processors. The input distribution is not important, since the number of operations required in the execution of the algorithm does not depend on how the input data is distributed among the processors.

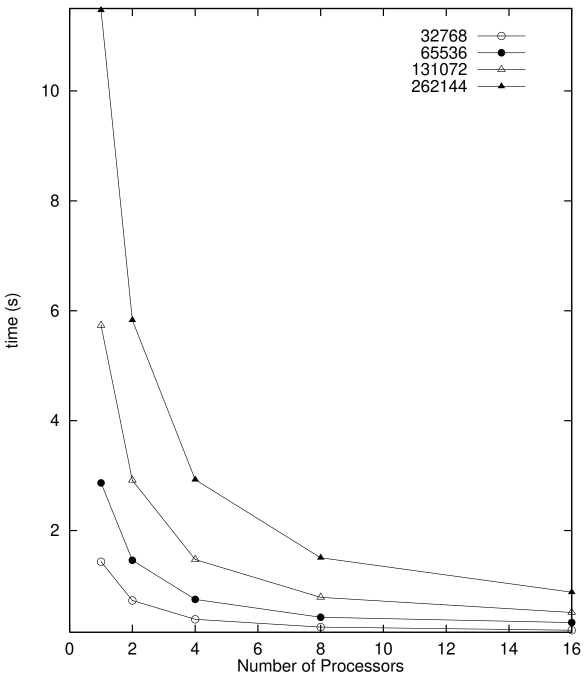

Fig. 5: Maximum times.

There is only one communication round in which the minima of the subarrays in each processor are broadcast to the others. Therefore, the amount of data exchanged depends on p and does not depend on n. Communication thus becomes less significant for large n.

We use the term *speedup* as the quotient between the sequential running time and the parallel running time. For our experiments, the sequential time is obtained by using an optimized sequential algorithm (Gabow et al. [20]) running on a single processor, instead of running the parallel algorithm run on one processor.

For small n ($n < 2^{10}$), no significant gain is obtained due to communication overhead. The effect of communication becomes less influential for $n > 2^{10}$ and the speedup becomes significant for $n > 2^{15}$.

For a given experiment, with fixed n and p, we do not observe significant difference in the running times in different executions. Therefore, we run the experiments only once each to get the results.

Fig. 5 shows the running times (in seconds) for $n = 2^k$ with $k = 15, \ldots, 18$ and using 1, 2, 4, 8 and 16 processors.

Fig. 6 shows the speedup obtained. Notice the speedup curves approximate the optimal curve for large n.

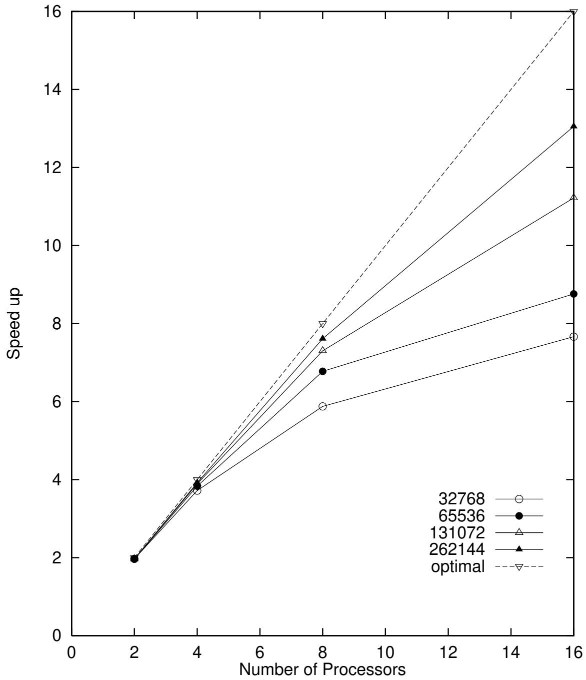

Fig. 6: Speed up.

5.5 Conclusion

In this work we present an algorithm, in the CGM(n, p) model, for the range minima problem that runs in $O(n/p)$ time using $O(1)$ communication rounds. This algorithm can be used to design a CGM algorithm for the LCA problem, based on the PRAM algorithm of JáJá [23], instead of other PRAM algorithms as in [24, 33], which would give less efficient implementations. Many graph algorithms in turn are based on the LCA problem.

In the design of the CGM algorithm, we chose to use the PS-tree to handle case 2(b) instead of merely constructing a Cartesian tree of an array of p numbers. This second approach, though feasible, would increase the local processing time. The small communication overhead, together with efficient local computing, enable us to obtain the final CGM algorithm, which is able to beat the very efficient sequential algorithm, as shown in the experimental results.

The experimental results are very promising, with speedup curves getting closer to the optimal speedup curve for large n, indicating the efficiency and scalability of the proposed algorithm. The efficiency of the proposed algorithm results from the constant number of communication rounds with low communication overhead and $O(n/p)$ computing time.

6 Lowest Common Ancestor

The *lowest common ancestor*, $LCA(u, v)$, of two vertices u and v of a rooted tree $T = (V, E)$ is the vertex w that is an ancestor to both u and v, and is farthest from the root. We address the problem of preprocessing T in order to answer a query $LCA(u, v)$ quickly for any pair (u, v). We refer to this problem as the *lowest-common-ancestor* (LCA) *problem.*

We apply the approach in [23] that uses Euler tour and range-minimum calculation. It consists of the following operations:

(1) compute an Euler tour for T;

(2) find the levels, in T, for all vertices of the Euler tour;

(3) for each vertex v find $l(v)$ and $r(v)$ which denote the leftmost and rightmost) appearances, respectively, of v in the Euler tour;

(4) solve the range-minima problem defined as follows: given a list of numbers $\{b_1, b_2, \ldots, b_n\}$ and an interval $[i, j]$, with $1 \leq i \leq j \leq n$, find the minimum of $\{b_i, \ldots, b_j\}$.

Operation 4 is important because of the following:

Lemma 12 *Consider a rooted tree $T = (V, E)$. Let u and v be two arbitrary distinct vertices of T. Then, the following statements hold:*

- *u is an ancestor of v if and only if $l(u) < l(v) < r(u)$.*
- *u and v are not related (that is, u is not a descendant of v and v is not a descendant of u) if and only if either $r(u) < l(v)$ or $r(v) < l(u)$.*
- *If $r(u) < l(v)$ then $LCA(u, v)$ is the vertex with minimum level contained in the interval $[r(u), l(v)]$.*

Operation 1 can be performed in $O(\log p)$ communication rounds as shown in [12]. The same holds for Operation 2 because it can also be reduced to Euler tour computation.

Consider Operation 3. Given an Euler tour of vertices $a_1, a_2, \ldots, a_n, a_1$. The element $a_i = v$ is the leftmost (rightmost) appearance of v if and only if *level*(a_{i-1}) = *level*(v) - 1 (*level*(a_{i+1}) = *level*(v) - 1, respectively) [23]. This requires the use of indices of the vertices in the Euler tour. Our Euler tour is not given as an array of vertices but rather by pointers to successor vertices in the tour. This is easily solved by considering the index as the rank obtained by list ranking from Section 3.2.1. The rank can be viewed as an index going backwards from the list. After list ranking (in $O(\log p)$ communication rounds), Operation 3 can be completed in $O(1)$ communication rounds.

Operation 4 also uses indices. Likewise, instead of indices, we utilize the ranks of the vertices of the Euler tour. Given two vertices u and v of T. In order to find the minimum level over the interval $[r(u), l(v)]$, let $\text{rank}(r(u)) = i$ and $\text{rank}(l(v) = j$. To find the required minimum, each of the p processors considers vertices in its local memory with ranks between j and i and finds the minimum level (O(n/p time). The minimum of the resulting p numbers can be found in O(1) communication rounds.

We obtain

Theorem 8 *Consider a rooted tree $T = (V, E)$ with n vertices. The LCA problem can be solved on a CGM with p processors and $O(\frac{n}{p})$ local memory per processor using $O(\log p)$ communication rounds and $O(\frac{n}{p})$ local computation per round.*

7 Concluding Remarks

In the CGM(n, p) model, we are interested in designing parallel algorithms with linear local executing time and that utilize few communication rounds – preferably a constant number. This is because the communication between processors is more expensive than the computing time of the local data.

Several results are presented in this text.

We present a randomized parallel list ranking algorithm for distributed memory multiprocessors. A simple version requires, with high probability, $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ communication rounds (h-relations with $h = \tilde{O}(\frac{n}{p})$) and $\tilde{O}(\frac{n}{p})$ local computation. An improved version requires, with high probability, only $r \leq (4k+6)\log(\frac{2}{3}p) + 8 = \tilde{O}(k\log p)$ communication rounds where $k = \min\{i \geq 0 | \ln^{(i+1)} n \leq (\frac{2}{3}p)^{2i+1}\}$. Note that $k < \ln^*(n)$ is an extremely small number. For $n \leq 10^{10^{100}}$ and $p \geq 4$, the value of k is at most 2. For a given number of processors, p, the number of communication rounds required is, for all practical purposes, independent of n. For $n \leq 10^{10^{100}}$ and $4 \leq p \leq 2048$, the number of communication rounds in our algorithm is bounded, with high probability, by 118. We conjecture that the actual number of communications rounds will not exceed 50.

We then present *deterministic* parallel algorithms for the coarse grained multicomputer (CGM) which solve the list ranking and determination of the connected components of a graph. The algorithms require $O(\log p)$ communication rounds and linear sequential work per round. We view the algorithms presented as an important step towards the final goal of $O(1)$ communication rounds. Note that the number of communication rounds obtained in this paper is independent of n, and grows only very slowly with respect to p. Hence, for most practical purposes, the number of communication rounds can be considered as constant. The result for these two problems constitutes a considerable improvement over those previously reported.

Given an array of n real numbers $A = (a_1, a_2, \ldots, a_n)$, define $MIN(i,j) = \min\{a_i, \ldots, a_j\}$. The range minima problem consists of preprocessing array A such that queries $MIN(i,j)$, for any $1 \leq i \leq j \leq n$, can be answered in constant time. Range minima is a basic problem that appears in many other important problems such as lowest common ancestor, Euler tour, pattern matching with scaling, etc. We present a parallel algorithm under the CGM model (Coarse Grained Multicomputer), that solves the range minima problem in $O(\frac{n}{p})$ time and constant number of communication rounds.

References

[1] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1975.

[2] Alon, N. and Schieber, "Optimal preprocessing for answering on-line product queries," Technical Report TR 71/87, The Moise and Frida Eskenasy Inst. of Computer Science, Tel Aviv University, 1987.

[3] Amir, A., Landau, G. M. and Vishkin, U. "Efficient Pattern Matching with Scaling," *Journal of Algorithms*, Vol. 13, 1992, pp. 2–32.

[4] Anderson, J. R. and Miller, G. L. "Deterministic parallel list ranking", J. H. Reif (ed.), *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing*, Springer Verlag, Lecture Notes in Computer Science, Vol. 319, 1988, pp. 81 –90.

[5] Anderson, J. R. and Miller, G. L. "A simple randomized parallel algorithm for list ranking". *Information Processing Letters*, Vol. 33, No. 5, 1990, pp. 269 – 273.

[6] Atallah, M. J. and Hambrusch, S. E. "Solving tree problems on a mesh-connected processor array," *Information and Control*, Vol. 69, 1986, pp. 168-187.

[7] Baase, S. "Introduction to parallel connectivity, list ranking, and Euler tour techniques". J. H. Reif (ed.) *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publisher, 1993.

[8] Bäumker, A. and Dittrich, W. "Parallel Algorithms for Image Processing: Practical Algorithms with Experiments," *International Parallel Processing Symposium*, IEEE Computer Society Press, 1996, pp. 429 - 433.

[9] Berkman, O. and Vishkin, U. "Recursive Star-Tree Parallel Data Structure," *SIAM J. Comput.*, Vol. 22, No. 2, 1993, pp. 221–242.

[10] Berkman, O. Schieber, B. and Vishkin, U. "Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values", *Journal of Algorithms*, Vol. 14, 1993, pp. 344–370.

[11] Blelloch, G. E., Leiserson, C. E. Maggs, B. M. and Plaxton, C. G. "A Comparison of Sorting Algorithms for the Connection Machine CM-2.," in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 3-16.

[12] Caceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. W. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. Proceedings ICALP'97 - 24th International Colloquium on Automata, Languages, and Programming, P. Degano, R. Gorrieri, A. Marchetti-Spaccamela (editors). Lecture Notes in Computer Science, Vol. 1256 (1997) 390–400.

[13] Cole, R. and Vishkin, U. "Approximate parallel scheduling, Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, Vol. 17, No. 1, 1988, pp. 128 – 142.

[14] Cole, R. and Vishkin, U. "Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time", *SIAM Journal of Computing*, Vol. 17, No. 1, 1988.

[15] Dehne, F., Fabri, A. and Rau-Chaplin, A. Scalable parallel geometric algorithms for coarse grained multicomputers. Proc. ACM 9th Annual Computational Geometry (1993) 298–307.

[16] Dehne, F., Fabri, A. and Kenyon, C. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. Proc. 6th IEEE Symposium on Parallel and Distributed Processing (1994) 586–593.

[17] Dehne, F., Deng, X., Dymond, P., Fabri, A. and Kokhar, A. A. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA'95) (1995) 27–33.

[18] Dehne, F. and Song, S. W. "Randomized parallel list ranking for distributed memory multiprocessors," in Proc. *Second Asian Computing Science Conference*, ASIAN'96, Singapore, Dec. 1996, Springer Lecture Notes in Computer Science 1179, pp. 1–10.

[19] Deng, X. and Dymond, P. "Efficient Routing and Message Bounds for Optimal Parallel Algorithms," in Proc. Int. Parallel Proc. Symp., 1995.

[20] Gabow, H. N., Bentley, J. L. and Tarjan, R. E. "Scaling and Related Techniques for Geometry Problems," *Proc. 16th ACM Symp. On Theory of Computing*, 1984, pp. 135–143.

[21] Goodrich, M. T. "Communication efficient parallel sorting," ACM Symposium on Theory of Computing (STOC), 1996.

[22] Harel, D. and Tarjan, R. E. "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, Vol. 13, No. 2, 1984, pp. 338–335.

[23] JáJá, J. *An introduction to parallel algorithms.* Addison Wesley, 1992.

[24] Lin, R. and Olariu, S. "A Simple Optimal Parallel Algorithm to Solve the Lowest Common Ancestor Problem," *Proc. of International Conference on Computing and Information - ICCI'91*, Lecture Notes in Computer Science, Vol. 497, 1991, pp. 455–461.

[25] Miller, G. L. and Reif, J. H. "Parallel tree contraction part 1: Fundamentals". *Advances in Computing Research*, Vol. 5, 1989, pp. 47 –72.

[26] Miller, G. L. and Reif, J. H. "Parallel tree contraction part 1: Further applications". *SIAM J. Computing*, Vol. 20, No. 6, December 1991, pp. 1128 – 1147.

[27] Mongelli, H. and Song, S. W. "A range minima parallel algorithm for coarse grained multicomputers". IPPS99/Irregular - Sixth International Workshop on Solving Irregularly Structured Problems in Parallel. *Lecture Notes in Computer Science*, Vol. 1586, Jos Rolim et al. (eds), Springer-Verlag. San Juan, Puerto Rico, April 12 - 16, 1999, pp. 1075 -1084.

[28] Mulmuley, K. *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice Hall, New York, NY, 1993.

[29] Ramachandran, V. L. and Vishkin, U. "Efficient parallel triconectivity in logarithmic parallel time," *Proceedings of AWOC'88*, Lecture Notes in Computer Science, Vol. 319, 1988, pp. 33–42.

[30] Reid-Miller, M., Miller, C. L. and Modugno, F. "List ranking and parallel tree compaction". J. H. Reif (ed.) *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publisher, 1993.

[31] Reid-Miller, M. "List ranking and list scan on the Cray C-90," in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 104-113.

[32] Reif, J. H. (ed.) *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, 1993.

[33] Schieber, B. and Vishkin, U. "On Finding Lowest Common Ancestors: simplification and parallelization," *SIAM J. Comput.*, Vol. 17, 1988, pp. 1253–1262.

[34] Shiloch, Y. and Vishkin, U. "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, 3(1), pp. 57–67, 1983.

[35] Valiant, L. G. et al. General purpose parallel architectures. Handbook of Theoretical Computer Science, Edited by J. van Leeuwen, MIT Press/Elsevier (1990) 943–972.

[36] Vuillemin, J. "A unified look at data structures," *Comm. of the ACM*, Vol. 23, 1980, pp. 229-239.