Algorithmica (2002) 33: 183–200
DOI: 10.1007/s00453-001-0109-4

Algorithmica
© 2002 Springer-Verlag New York Inc.

Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP[1]

F. Dehne,[2] A. Ferreira,[3] E. Cáceres,[4] S. W. Song,[5] and A. Roncato[6]

Abstract. In this paper we present *deterministic* parallel algorithms for the *coarse-grained multicomputer* (CGM) and *bulk synchronous parallel* (BSP) models for solving the following well-known graph problems: (1) list ranking, (2) Euler tour construction in a tree, (3) computing the connected components and spanning forest, (4) lowest common ancestor preprocessing, (5) tree contraction and expression tree evaluation, (6) computing an ear decomposition or open ear decomposition, and (7) 2-edge connectivity and biconnectivity (testing and component computation). The algorithms require $O(\log p)$ communication rounds with linear sequential work per round (p = no. processors, N = total input size). Each processor creates, during the entire algorithm, messages of total size $O(\log(p)(N/p))$.

The algorithms assume that the local memory per processor (i.e., N/p) is larger than p^{ε}, for some fixed $\varepsilon > 0$. Our results imply BSP algorithms with $O(\log p)$ supersteps, $O(g \log(p)(N/p))$ communication time, and $O(\log(p)(N/p))$ local computation time.

It is important to observe that the number of communication rounds/supersteps obtained in this paper is independent of the problem size, and grows only logarithmically with respect to p. With growing problem size, only the sizes of the messages grow but the total number of messages remains unchanged. Due to the considerable protocol overhead associated with each message transmission, this is an important property. The result for Problem (1) is a considerable improvement over those previously reported. The algorithms for Problems (2)–(7) are the first practically relevant parallel algorithms for these standard graph problems.

Key Words. Coarse grained parallel computing, Graph algorithms.

1. Introduction.

Speedup results for theoretical PRAM algorithms do not necessarily match the speedups observed on real machines [1], [36]. Given sufficient slackness in the number of processors, Valiant's *bulk synchronous parallel* (BSP) approach [39] simulates PRAM algorithms optimally on distributed memory parallel systems. Valiant points out, however, that one may want to design algorithms that utilize local computations and minimize global operations [38], [39]. The BSP approach requires that g (= local computation speed $\div$ router bandwidth) is low, or fixed, even for increasing

[1] This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, FAPESP (Brasil), CNPq (Brasil), PROTEM-2-TCPAC (Brasil), the Commission of the European Communities (ESPRIT Long Term Research Project 20244, ALCOM-IT), DFG-SFB 376 "Massive Parallelität" (Germany), and the Région Rhône-Alpes (France). A preliminary version of this paper was published in the proceedings of the 1997 International Colloquium on Automata, Languages and Programming [5].
[2] School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. frank@dehne.net, www.dehne.net.
[3] CNRS - I3S, INRIA, Sophia-Antipolis, France. ferreira@sophia.inria.fr.
[4] Univ. Federal de Mato Grosso do Sul, Campo Grande, Brasil. edson@dct.ufms.br.
[5] University of São Paulo, São Paulo, Brazil. song@ime.usp.br.
[6] Facolta di Scienze Mat. Fis. e Nat., Mestre, Italia. roncato@dsi.unive.it.

Received July 5, 2000; revised April 16, 2001. Communicated by F. P. Preparata.
Online publication March 11, 2002.

number of processors. Gerbessiotis and Valiant [18] describe circumstances where PRAM simulations cannot be performed efficiently, among others, if the factor g is high. Unfortunately, this is true for most currently available multiprocessors. The parallel algorithms presented in this paper consider this case for graph problems.

As pointed out in [39], the cost of a message also contains a constant overhead cost s. The value of s can be fairly large and the total message overhead cost can have a considerable impact on the speedup observed (see, e.g., [9]). In this paper we use a more practical version of the BSP model, referred to as the *coarse-grained multicomputer* (CGM) model [8]–[11]. It is comprised of a set of p processors $P_1, \ldots, P_p$ with $O(N/p)$ local memory per processor and an arbitrary communication network (or shared memory). All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single h-relation with $h = O(N/p)$, i.e., each processor sends $O(N/p)$ data and receives $O(N/p)$ data. We require that all information sent from a given processor to another processor in one communication round is packed into one long message, thereby minimizing the message overhead. A CGM computation/communication round corresponds to a BSP superstep with communication cost $g(N/p)$ (plus the above "packing requirement").

Finding an optimal algorithm in the CGM model is equivalent to minimizing the number of communication rounds as well as the total local computation time and total message size. This considers all parameters discussed above that are affecting the final observed speedup and it requires no assumption on g. Furthermore, it has been shown that minimizing the number of supersteps also leads to improved portability across different parallel architectures [38], [39], [14]. The above model has been used (explicitly or implicitly) in parallel algorithm design for various problems [4], [8]–[11], [13], [15], [25] and shown very good practical timing results.

In this paper we study deterministic parallel graph algorithms for the CGM and BSP models. We consider the following well-known graph problems:

1. List ranking.
2. Euler tour construction.
3. Computing the connected components and spanning forest.
4. Lowest common ancestor preprocessing.
5. Tree contraction and expression tree evaluation.
6. Computing an ear decomposition or open ear decomposition.
7. 2-Edge connectivity and biconnectivity (testing and component computation).

These problems have been extensively studied for fine-grained parallel networks and for the PRAM (see, e.g., [32]). However, for the practically much more relevant CGM/BSP model there exist, to the best of our knowledge, only a few results on parallel graph algorithms.

For the remainder, let n denote the number of vertices, let m be the number of edges of a given input graph G, and let $N = n + m$. Reid-Miller [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 at that time. More detailed empirical studies and tradeoffs with respects to the communication volume are presented in [35]. The algorithm in [31] required $O(\log n)$ communication rounds. In [12], an improved algorithm was presented which required, with high probability, only

$O(k \log p)$ rounds, where $k \leq \log^* n$. In [14] the list ranking problem is considered as well. Here $O(\log p)$ communication rounds are achieved by a randomized algorithm. Bäumker and Dittrich presented in [3] a 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. Again, both algorithms are randomized.

We improve these results by giving the first *deterministic* algorithms for list ranking and computing connected components in $O(\log p)$ rounds. Algorithms with $O(1)$ communication rounds have been presented for various Computational Geometry problems [9]–[12], [17], but the graph problems studied in this paper have considerably less "internal structure" which could be exploited to obtain such solutions. It is not known whether solutions with $O(1)$ communication rounds exist for these graph problems. Note that, in practice, the number of processors is usually fixed. In contrast to the previous deterministic results, the improved number of communication rounds obtained in this paper, $O(\log p)$, is *independent* of n. This is of considerable practical relevance. With growing problem size, the number of messages remains unchanged. Only the sizes of these messages grow, linear with respect to the growth in problem size. Due to the considerable protocol overhead associated with each message transmission, this is an important property. In fact, our experience in implementing parallel algorithms on standard commercial machines indicates that this property is, in most cases, a crucial ingredient for practically relevant parallel algorithms.

As in [31] we, in general, assume that $N \gg p$ (coarse-grained), because this is usually the case in practice. More precisely, we assume that $N/p \geq p^\varepsilon$ for some fixed $\varepsilon > 0$, which is true for most commercially available multiprocessors.

In Section 3 of this paper we use a technique called *deterministic list compression* to obtain a deterministic list ranking algorithm with $O(\log p)$ rounds. The connected components algorithm is presented in Section 4.1. It uses a technique called *accelerated cascading*. That is, it simulates an existing PRAM algorithm for the same problem but stops the execution of this algorithm after $O(\log p)$ rounds and then finishes the computation with a different (new) CGM algorithm. In Sections 4.2–4.5 we present *deterministic* parallel CGM/BSP algorithms with $O(\log p)$ communication rounds for solving Problems 4–7, respectively. All algorithms require linear sequential work per round and each processor creates, during the entire algorithm, messages of total size $O(\log(p)(N/p))$. To our knowledge, these are the only currently known CGM and BSP algorithms for these problems.

Before we proceed with presenting the above-mentioned results we give, in Section 2, an overview of the BSP and CGM models and their relationship to each other.

2. The BSP, CGM, and Related Parallel Computing Models.

The BSP model was introduced in [38], and the CGM model was presented in [9]–[11].

A BSP computer is a collection of processor/memory modules connected by a router that can deliver messages in a point to point fashion between the processors. A BSP-style computation is divided into a sequence of supersteps separated by barrier synchronizations. In *computation supersteps* the processors perform computations on data that was present locally at the beginning of the superstep. In *communication supersteps* data is

exchanged among the processors via the router. A BSP computer has the following parameters: N refers to the problem size, p is the number of processors, L is the minimum time between synchronization steps (measured in basic computation units), and g is the ratio of overall system computational capacity (number of computation operations) per unit time divided by the overall system communication capacity (number of messages of unit size that can be delivered by the router) per unit time. A BSP algorithm with a total of λ supersteps has the following computation and communication costs: The *computation cost* T_{comp} of the algorithm is $T_{\text{comp}} = \sum_{i=1}^{\lambda} w_{\text{comp}}^{i}$, where the ith computation superstep is assigned cost $w_{\text{comp}}^{i} = \max\{L, t_1, ..., t_p\}$. Here t_j is the number of basic computation operations performed by processor j in the ith superstep. The *communication cost* T_{comm} of the algorithm is $T_{\text{comm}} = \sum_{i=1}^{\lambda} w_{\text{comm}}^{i}$, where the ith communication superstep is assigned cost $w_{\text{comm}}^{i} = \max_{j=1}^{p} \{w_{\text{comm},j}^{i}\}$. Here $w_{\text{comm},j}^{i}$ is the communication cost incurred by processor j in the ith superstep. Assuming that processor p_j receives messages of lengths $r_1, \ldots, r_{j'}$ and sends messages of lengths $\{s_1, \ldots, s_{j''}\}$ during the ith superstep, $w_{\text{comm},j}^{i} = \max\{L, g(\sum_{u=1}^{j'} r_i + \sum_{u=1}^{j''} s_i)\}$.

A CGM(N, p) uses only two parameters, N and p, and assumes a collection of p processors with N/p local memory each 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, and the total computation cost T_{comp} is defined analogously. A communication round consists of a single h-relation with $h \leq N/p$. The cost w_{comm}^{i} of each communication round has the same value, referred to as $H_{N,p}$. Therefore, the total communication cost T_{comm} of a CGM algorithm with λ communication rounds is simply $T_{\text{comm}} = \lambda H_{N,p}$. In a recent overview of different BSP and related models, Goodrich [20] referred to the CGM as the *weak-CREW BSP*. The main difference between the BSP and CGM models is that the latter allows only one single type of communication operation, the h-relation, and simply counts the number of h-relations as its main measure of communication cost. Note that every CGM algorithm is also a BSP algorithm but not vice versa. The CGM model aims at designing simple and practical yet theoretically optimal or efficient parallel algorithms for *coarse-grained* parallel systems ($N/p \gg 1$). Algorithms do usually require a lower bound on N/p, e.g., $N/p \geq p$ or $N/p \geq p^{\varepsilon}$. The CGM model targets in particular the case where the overall computation speed is considerably larger than the overall communication speed, which is usually the case. Since the message size is maximal, the CGM model minimizes the message overhead associated with sending a single message (regardless of its length), which is very important in practice.

In summary, the main advantage of the CGM model is that it allows us to model the communication cost of a parallel algorithm by one single parameter, the number of communication rounds, λ. Note that, while λ is the main parameter determining the performance of a CGM algorithm, we will also indicate other parameters like the local computation and total communication when analyzing CGM algorithms (more discussion below).

Previous definitions of the CGM model (e.g., [9]) distinguished between the costs of an h-relation and the cost of sorting. However, due to the recent results in [20] these are equivalent for $N/p \geq p^{\varepsilon}$. Also, it is not necessary to distinguish between

balanced and unbalanced h-relations. In both cases each processor sends/receives $O(h)$ data but in the balanced case the data exchanged between any two processors is always $O(h/p)$ whereas it may vary in the unbalanced case. It has been shown in [2] that an unbalanced h-relation can be simulated by $O(1)$ balanced h-relations. Another possible case of imbalance occurs when some processors send/receive less than $O(h)$ data. This problem has been studied in [22] (E-BSP model) but is not a topic of this paper. Another related model is the QSM [19] where communication is performed via a shared memory with emphasis on memory contention, i.e., simultaneous accesses to the same shared memory cell. The main difference between QSM and CGM is that the latter allows the use of only one single communication scheme, the h-relation, for resolving memory contention. Consult [20] for an overview of the different BSP related models.

The CGM model has recently attracted considerable interest in the parallel algorithms community. Several researchers have used it to design parallel algorithms for various problem areas (see, e.g., [8] and various CGM articles in the recent ACM SPAA and IPPS/SPDP proceedings). For most parallel processing implementations, the number of h-relations required is the overriding factor determining the performance because the protocol overhead associated with each message is usually substantial. In the extreme case this may of course not be true. A few extremely large messages may require more time than a larger number of very small messages. Therefore, it is also useful to study the total message size per processor over all rounds, i.e., the sum of the sizes of all message sent by a processor during the entire computation. For all algorithms presented in this paper, the total message size per processor over all rounds is $O(\log(p)(N/p))$.

In the remainder of this paper we design and analyze our parallel graph algorithms in the CGM model. The relationship to the BSP model is given by the following:

OBSERVATION 1. *A CGM algorithm with λ rounds and computation cost T_{comp} corresponds to a BSP algorithm with λ supersteps*, *communication cost $O(g\lambda(N/p))$*, *and the same computation cost T_{comp}*.

3. List Ranking. Let L be a linked list of length n represented by a vector $s[1\cdots n]$. For each $i \in \{1\cdots n\}$, $s[i]$ is a pointer to the list element following i in the list L. We refer to i and $s[i]$ as *s-neighbors*. The last element λ of the list L is the one with $s[\lambda] = \lambda$. The distance between i and j, $d_L(i, j)$, is the number of nodes in L between i and j plus one (i.e., the distance is zero if and only if $i = j$, 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 λ, referred to as $rank_L(i) = d_L(i, \lambda)$.

On a $CGM(n, p)$, each processor initially stores n/p list elements of L with their respective pointers. Figure 1 shows an example of a linked list with $n = 24$ elements stored on a *CGM* with $p = 4$ processors, $n/p = 6$ elements per processor.

In the remainder of this section we present a *deterministic* CGM list ranking algorithm which requires $O(\log p)$ rounds.

We need the following definitions. An *r-ruling set* L' of L is defined as a subset of *selected* list elements of L 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. For each $i \in L$ let $s_{L'}[i]$ be the next selected element

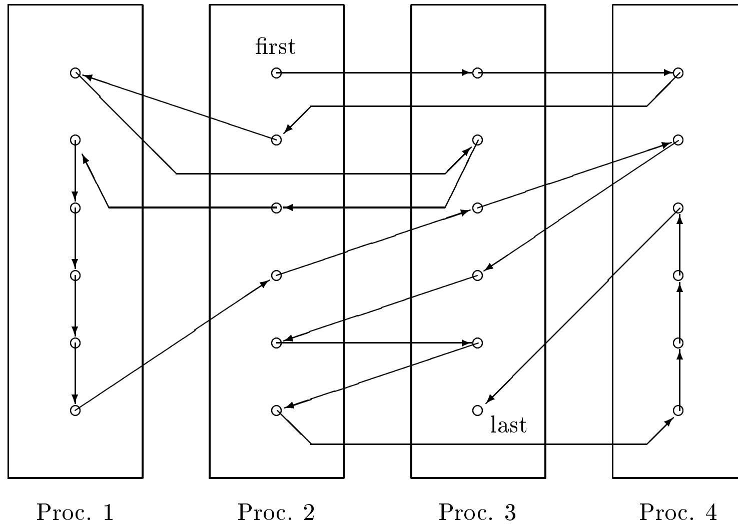

Fig. 1. A linked list L stored on a $CGM(n, p)$.

$j \in L'$ with respect to the order implied by L. We represent an r-ruling set L' again as a linked list where each element $i \in L'$ is assigned a pointer to $s_{L'}[i]$ and a value $w(i) = d_L(i, s_{L'}[i])$ representing the distance between i and $s_{L'}(i)$. See Figure 2 for an illustration. The weighted list ranking problem on L' with weights $w(\cdot)$ refers to computing for each $i \in L'$ the sum of the weights $w(j)$ of all nodes $j \in L'$ between i and λ.

Algorithm 1 outlines the top-level of our CGM list ranking method. An illustrating example is given in Figure 3.

Algorithm 1. CGM List Ranking

Input: A linked list L of length n stored on a $CGM(n, p)$, $n/p \geq p^{\varepsilon}$. Each processor stores n/p list elements $i \in L$ and their respective pointers $s[i]$.
Output: For each list element i its rank $rank_L(i)$ in L.

1. The CGM computes an $O(p^2)$-ruling set R of size $|R| = O(n/p)$ as described in Algorithm 2 below.
2. R is broadcast to all processors. This broadcast is implemented as $O(\log p)$ communication rounds where the number of processors storing R is initially one and then doubled in each communication round.

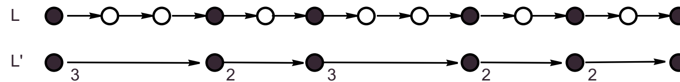

Fig. 2. A list L and a 3-ruling set L'.

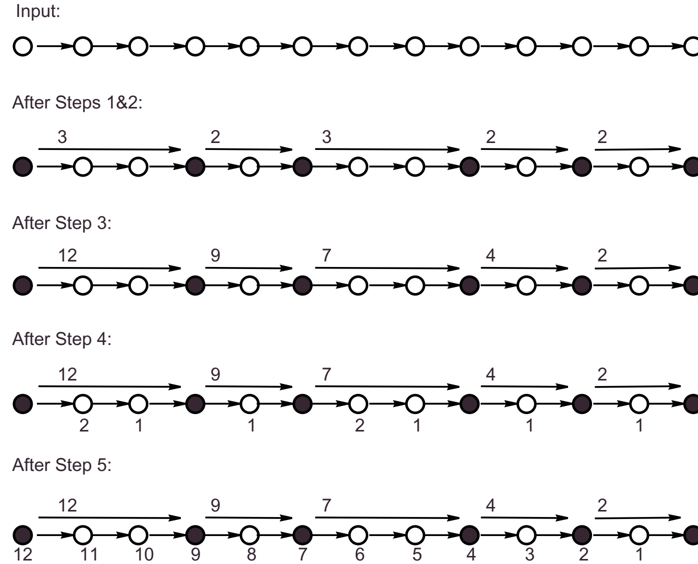

Fig. 3. Illustration of Algorithm 1.

3. Each processor, sequentially, performs weighted list ranking on R with weights $w(\cdot)$, thereby computing for each $j \in R$ its rank $rank_L(j)$ in L.
4. Each list element $i \in L - R$ has at most distance $O(p^2)$, with respect to L, to its next element $s_R[i]$ in R. Using $O(\log p)$ *CGM* sorting steps, the *CGM* simulates $O(\log p)$ pointer jumping steps of the standard PRAM list ranking algorithm, thereby computing for each $i \in L - R$ its distance $d_L(i, s_R[i])$ to its next element $s_R[i]$ in R.
5. Each processor locally computes the ranks of its list elements $i \in L - R$ as follows: $rank_L(i) = d_L(i, s_R[i]) + rank_L(s_R[i])$.

The hard part of the algorithm is the computation of a $O(p^2)$-ruling set R of size $O(n/p)$ which we discuss in detail below. Given such a $O(p^2)$-ruling set, the correctness of Algorithm 1 is straightforward. We also observe that Steps 2–5 can be easily implemented on a *CGM* in $O(\log p)$ communication rounds with $O(n/p)$ local computation per round. Note that in Step 4 we are simulating PRAM pointer jumping steps on the CGM. Each such step can be implemented in $O(1)$ rounds by applying Goodrich's sorting algorithm [20] for $n/p \geq p^{\varepsilon}$.

In the remainder of this section we introduce a new technique, called *deterministic list compression*, which 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.

We define an *s-interval* of length k as a sequence $I = (i_1, \ldots, i_k)$ of list elements with $s[i_j] = i_{j+1}$, $1 \leq j \leq k-1$. The two *neighbors* of the s-interval I are the two list elements n_1, n_2 not in I such that $s[n_1] = i_1$ and $s[i_k] = n_2$. We refer to a maximal s-interval $I = (i_1, \ldots, i_k)$ of list elements which are all stored at the same processor as a *local s-interval*. A maximal s-interval $I = (i_1, \ldots, i_k)$ of list elements where no two subsequent i_j, i_{j+1} are stored at the same processor is called a *nonlocal s-interval*.

For the compress phase we apply the *deterministic coin tossing* technique of [7] but with a different set of labels. Instead of the memory address used in [7], we use the number of the processor storing list element i as its label $l(i)$. A list element $s[i]$ is called a *local maximum* if $l(s^{-1}[i]) < l(i) > l(s[i])$ where $s^{-1}[i]$ is the list element j such that $s[j] = i$. Deterministic coin tossing selects the local maxima with respect to our new label $l(i)$. Note that there are at most p different labels. Therefore, for every nonlocal s-interval, the distance between two selected elements is $O(p)$. Deterministic coin tossing will not select any element of a local s-interval. However, since all elements of a local s-interval are stored at the same processor, we can process them sequentially. Details will be discussed later.

An example is shown in Figure 4. The linked list is the same as in Figure 1. The x-axis represents the ranks of the list elements and the y-axis represents the number of the processor on which each element is stored. The solid circles are the local maxima, i.e., those list elements selected by our modified deterministic coin tossing.

The above procedure selects, within each nonlocal s-interval, list elements with a distance between 2 and $O(p)$ between consecutive selected elements. If we would connect the selected elements by direct links between them, and repeat the procedure on this new linked list, and iterate this $O(\log p)$ times, then we would obtain a $O(p^2)$-ruling set of size $O(n/p)$. However, this would require more than $O(\log p)$ communication rounds. In order to apply deterministic coin tossing for a second, third, etc., time, the previously selected elements 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 approach would require a total of $O(\log^2 p)$ communication rounds.

Notice, however, that if two selected elements are at distance $\Theta(p)$, then it is unnecessary to apply further deterministic coin tossing in order to reduce the number of selected elements. The basic approach of our algorithm is therefore to interleave pointer jumping steps (*concatenate*) and deterministic coin tossing operations with respect to our new labeling scheme (*compress*). More precisely, we will have only one pointer jumping step between subsequent deterministic coin tossing steps, and such pointer jumping

Fig. 4. The linked list elements from Figure 1 labeled by their processor numbers.

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 2. CGM Algorithm for Computing a $O(p^2)$-Ruling Set

Input: A linked list L of length n stored on a $CGM(n, p)$, $n/p \geq p^\varepsilon$. Each processor stores n/p list elements $i \in L$ and their respective pointers $s[i]$.
Output: A set of selected nodes of L representing a $O(p^2)$-ruling set of size $O(n/p)$.

1. Each processor locally marks all its list elements as *not selected*.
2. Using global sort [20], all processors determine for each list element i its two neighbors $s^{-1}[i]$ and $s[i]$. Then each processor locally performs for each local list element i:
   IF $l(s^{-1}[i]) < l(i) > l(s[i])$ THEN mark i as *selected*.
3. Each processor locally determines its local s-intervals. Using global sort, all processors determine the two neighbors of each local s-intervals. Each processor examines locally all of its local s-intervals. For each local s-interval of size larger than two, every second element is marked as *selected*. If a local s-interval has size two and not both neighbors have a smaller label, then both elements of the local s-interval are marked as *not selected*.
4. FOR $k = 1 \cdots \log p$ DO
   4.1. Using global sort, all processors determine for each list element i the current $s[i]$ and $s[s[i]]$. Then each processor locally performs for each local list element i:
   IF $s[i]$ is not selected THEN set $s[i] := s[s[i]]$.
   4.2. Using global sort [20], all processors determine for each list element i its two current neighbors $s^{-1}[i]$ and $s[i]$. Then each processor locally performs for each local list element i:
   IF ($s^{-1}[i]$, i and $s[i]$ are selected) AND NOT ($l(s^{-1}[i]) < l(i) > l(s[i])$) AND ($l(s^{-1}[i]) \neq l(i)$) AND ($l(i) \neq l(s[i])$) THEN mark i as *not selected*.
   4.3. Each processor locally determines its local s-intervals. Using global sort, all processors determine the two neighbors of each local s-interval. Each processor examines locally all of its local s-intervals. For each local s-interval of size larger than two, every second element is marked as *not selected*. If a local s-interval has size two and not both neighbors have a smaller label, then both elements of the local s-interval are marked as *not selected*.
5. The processor storing the last element λ of L marks λ as selected.

We first prove that the set of elements selected at the end of Algorithm 2 is of size at most $O(n/p)$.

LEMMA 1. *After the* kth *iteration in Step* 4, *there are no more than two selected elements in any* s*-interval of length* 2^k *for the original list* L.

PROOF. By induction on k. The lemma is trivial for $k = 1$. Let S be an s-interval of length 2^k for the original list L and let S_1 and S_2 be the first and second half of S, respectively. By assumption, after iteration $k-1$ in Step 4, S_1 and S_2 contain at most two selected elements each. Denote the at most four selected elements, ordered with respect to L, by $e_1, \ldots, e_4$. Note that the distance between these elements, with respect to L, is at most 2^k. Consider now iteration k in Step 4. After Step 4.1, any two selected elements that have distance at most 2^k and no selected element between them (with respect to L) are directly connected by a link which is represented by the current s vector. Hence, $s[e_1] = e_2$, $s[e_2] = e_3$, $s[e_3] = e_4$. Note that there are at most four (nonsymmetric) possible cases for applying Steps 4.2 and 4.3 to $e_1, \ldots, e_4$. If $e_1, \ldots, e_4$ are pairwise distinct, then only Step 4.2 applies. If $e_1, \ldots, e_4$ are all equal, then Step 4.3 applies. If $e_1, \ldots, e_3$ are pairwise distinct and $e_3 = e_4$, then Step 4.2 applies to $e_1, \ldots, e_3$ and Step 4.3 to e_3, e_4. The other possible case is that $e_1 \neq e_2 = e_3 \neq e_4$. Then Step 4.2 applies to e_1 and e_4, and Step 4.3 applies to e_2, e_3. In any case, after Steps 4.2 and 4.3, at most two elements among $e_1, \ldots, e_4$ are still selected. □

We now prove that consecutive elements selected at the end of Algorithm 2 have distance at most $O(p^2)$ in the original list L. We first need the following.

LEMMA 2. *After every execution of Step* 4.3, *the distance* (*with respect to the current vector* s) *of two consecutive selected elements is at most* $O(p)$.

PROOF. Consider two consecutive selected elements e_1 and e_2. There are three possible cases: (1) e_1 and e_2 and all elements between them, with respect to s, have the same label. (2) For e_1 and e_2 and all elements between them, with respect to s, any pair of consecutive elements has different labels. (3) The mixed case where some pairs of consecutive elements have the same label. Note that in the mixed case it is impossible for three or more consecutive elements to have the same label, because one of them would be a selected element (Step 4.3). In Case (1) the distance of e_1 and e_2 is at most two because of Step 4.3. In Case (2), due to our modified labeling scheme, there are at most p different labels. Hence, by the standard argument for *deterministic coin tossing* [7] the distance is at most $O(p)$. Case (3) is equivalent to Case (2) except for a factor of two. □

LEMMA 3. *After the* kth *execution of Step* 4.3, *two* s*-neighbors with respect to the current vector* s *have distance* $O(2^k)$ *with respect to the original list* L.

PROOF. Obvious consequence of the fact that only k pointer jumping operations were so far executed in Step 4.1. □

LEMMA 4. *No two consecutive selected elements have a distance of more than* $O(p^2)$ *with respect to the original list* L.

PROOF. Follows from Lemmas 2 and 3. □

LEMMA 5. *On a CGM with p processors and $O(n/p)$ local memory per processor, $n/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, Algorithm 2 determines a $O(p^2)$-ruling set of size $O(n/p)$ in $O(\log p)$ communication rounds with $O(n/p)$ local computation per round.*

PROOF. The correctness of Algorithm 2 follows from Lemmas 1–4. For $n/p \geq p^{\varepsilon}$, the sorting algorithm in [20] requires $O(1)$ rounds with $O(n/p)$ local computation per round. The communication performed by Algorithm 2 consists of two global sorts for Steps 2 and 3, and $3 \log p$ global sorts for Step 4. All local computation in each round can be performed in linear time. Thus, the claimed time complexity follows. □

In summary, we obtain

THEOREM 1. *The list ranking problem for a linked list with n vertices can be solved on a CGM with p processors and $O(n/p)$ local memory per processor, $n/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, using $O(\log p)$ communication rounds and $O(n/p)$ local computation per round.*

Euler Tour in a Tree. We complete this section with an important application of our list ranking algorithm. Let $T = (V, E)$ be an undirected tree. We assume that the tree T is represented by an adjacency list for each vertex. Let $T^* = (V, E^*)$ be a directed graph with $E^* = \{(v, w), (w, v) | \{v, w\} \in E\}$. T^* is Eulerian because $indegree(v) = outdegree(v)$ for each vertex v. The Euler tour problem for T consists of (1) computing a path that traverses each edge exactly once and returns to its starting point, and (2) computing for each vertex its rank in this path.

THEOREM 2. *The Euler tour problem for a tree T with n vertices can be solved on a CGM with p processors and $O(n/p)$ local memory per processor, $n/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, in $O(\log p)$ communication rounds with $O(n/p)$ local computation per round.*

PROOF. We compute T^* and its adjacency lists by doubling all edges of T and applying sorting [20]. Furthermore, we make the adjacency list for each vertex circular by applying list ranking. For each edge (i, j) in T^* let $next(i, j)$ be the successor of its entry in the respective adjacency list. We now apply the well known method by Tarjan and Vishkin [37] to define an Euler tour ordering on the edges of T^* which assigns to each edge (i, j) as successor the edge $next(j, i)$. Computing $next(j, i)$ for every edge (i, j) reduces to sorting. Finally, we apply our list ranking method described above to determine the rank of each vertex in the Euler tour. □

4. Porting PRAM Algorithms to the CGM/BSP. In this section we present CGM graph algorithms for connected component labeling, lowest common ancestor computation, tree contraction, open ear decomposition, and biconnected component labeling. All algorithms require $O(\log p)$ rounds. They are obtained by porting the respective PRAM algorithms to the CGM, using our CGM list ranking algorithm presented in Section 3.

Our approach for connected component labeling is to simulate the respective PRAM method using sorting [20], but only for $O(\log p)$ rounds, and then finish the computation with a new $O(\log p)$ rounds CGM algorithm based on binary merge. Our solutions for the other problems reduce these tasks to a sequence of list ranking and connected component computations.

4.1. *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.

Algorithm 3 shown below computes the connected components of G on a CGM with p processors and $O((n+m)/p)$ local memory per processor. Step 1 simulates the PRAM algorithm by Shiloch and Vishkin [34] but with only $\log p$ iterations of the main loop instead of the $O(\log n)$ iterations in Shiloch and Vishkin's original algorithm. Step 2 converts all resulting trees into stars. It follows from [34] that the obtained graph $G' = (V', E')$ has at most $O(n/p)$ vertices. Hence, V' can be broadcast to all processors. E' can still be of size $O(m)$ and is distributed over the p processors. E_i refers to the edges of 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 $O(|V'|) = O(n/p)$. Hence, each of the spanning forests computed in Step 3 can be stored in the local memory of a single processor. We merge pairs of spanning forests until, after $\log p$ rounds, the spanning forest of G' is stored at processor P_0. In Step 4 all processors then update the partial connected component information obtained in Step 1.

Algorithm 3. 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, $(n + m)/p \geq p^{\varepsilon}$.
Output: The connected components of G represented by the values *parent*(v) for all vertices $v \in V$.

1. Using sorting [20], simulate $\log p$ iterations of the main loop of the PRAM algorithm by Shiloch and Vishkin [34].
2. Use the Euler tour algorithm in Section 3 to convert all resulting 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 m/p edges of E'. Let E_i be the edges stored at processor i, $0 \leq i \leq p - 1$.
3. Set all processors to *active* mode.
   FOR $k := 1$ to $\log p$ DO
      Partition the active processors into groups of size two.
      FOR each group P_i, P_j of active processors, $i < j$, IN PARALLEL DO
      • Processor P_j sends its edge set E_j to processor P_i.
      • Processor P_j is set to *passive* mode.

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

Set all processors to *active* mode and broadcast E_0.

4. Each processor P_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 $parent(u) \in V'$.)

We obtain

THEOREM 3. *Algorithm* 3 *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* $O((n + m)/p)$ *local memory per processor,* $(n + m)/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, *using* $O(\log p)$ *communication rounds and* $O((n + m)/p)$ *local computation per round.*

4.2. *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 apply the approach in [21] which 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 1 can be performed in $O(\log p)$ communication rounds as shown in Section 3. The same holds for Operation 2 because it can also be reduced to Euler tour computation. We now 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) [21]. 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 using as index the rank obtained by list ranking from Section 3. 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 $rank(r(u)) = i$ and $rank(l(v) = j$. 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

LEMMA 6. *Consider a rooted tree $T = (V, E)$ with n vertices. The LCA problem can be solved on a CGM with p processors and $O(n/p)$ local memory per processor, $n/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, using $O(\log p)$ communication rounds and $O(n/p)$ local computation per round.*

4.3. *Tree Contraction and Expression Tree Evaluation.* We observe that the classical tree contraction and expression tree evaluation algorithm of [27] can be easily implemented on a CGM to run in $O(\log p)$ communication rounds. Recall that the tree contraction algorithm of [27] applies an alternating sequence of $\log n$ *rake* and *compress* operations to contract a tree T into a single node. On a CGM, one can simply apply $\log p$ *rake* and *compress* operations, which require $O(\log p)$ rounds, and compresses the tree into a smaller tree T' of size $O(n/p)$. The tree T' can then be processed sequentially at a single processor. In order to perform expression tree evaluation such that not only the value of the root but the value of every node is calculated, the PRAM method of [27] can be employed for the CGM as well. Let $T = T_1, \dots, T_t = T'$ be the sequence of $\log p$ trees created by the alternating sequence of $\log p$ *rake* and *compress* operations. After T' has been evaluated sequentially, $\log p$ expansion steps recreate the above sequence of trees in reverse order. A node v that is added in an expansion was deleted either by a rake or a compress operation. In both cases its value can easily be computed in $O(1)$ rounds by a local neighborhood operation.

LEMMA 7. *Tree contraction and expression tree evaluation on a tree T with n nodes can be performed on a CGM with p processors and $O(n/p)$ local memory per processor, $n/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, using $O(\log p)$ communication rounds and $O(n/p)$ local computation per round.*

4.4. *Open Ear Decomposition.* We first recall the definition of an *ear decomposition* and *open ear decomposition* (see, e.g., [30]). Consider an undirected graph $G = (V, E)$ with n vertices and m edges. For the remainder, we assume that G is connected. An *ear decomposition* of G is an ordered partition of E into r simple paths $P_1, \dots, P_r$ such that P_1 is a cycle, and, for each $2 \leq i \leq r$, P_i is a simple path with endpoints belonging to $P_1 \cup \cdots \cup P_{i-1}$ but with none of its internal vertices belonging to P_j, $j < i$. The paths P_i are called *ears*. If none of the P_i, $i > 1$, is a cycle, then the decomposition is called an *open ear decomposition*. For an edge e in P_i, let i be the *ear number* of e.

An edge $e \in E$ is a *cut-edge* if e does not lie on a cycle in G. A connected undirected graph G is 2-*edge connected* if it contains no cut-edge. G has an ear decomposition if and only if G is 2-edge connected. A *cut-vertex* is a vertex whose removal leaves G disconnected. G is *biconnected* if it contains at least three vertices and has no cut-vertex. It has been shown that G has an open ear decomposition if and only if it is biconnected [40].

Let T be a spanning tree of G rooted at some node r. Consider the preorder numbering of T with respect to r and let *preorder*(v) be the preorder number of a node v, $0 \leq$

preorder$(v) \le n-1$. For an edge e let *lca*(e) denote the lowest common ancestor of $e = (u, v)$, as defined in Section 4.2.

An edge in $G - T$ is a *nontree edge* with respect to T. Any nontree edge e of G creates a cycle in $T \cup \{e\}$, called the *fundamental cycle* of e with respect to T. For each vertex v consider all fundamental cycles created by nontree edges incident to a descendant of v in T. Let *low*(v) be the minimum preorder number of a node w which lies on any such fundamental cycle. (If no such w exists, then let *low*$(v) = n$.)

The classical $O(\log n)$ time PRAM algorithms for ear decomposition and open ear decomposition [26], [28], [30] consist of a constant number of the following operations:

1. find a spanning tree T for G;
2. find the lowest common ancestor *lca*(e) of every nontree edge $e = (u, v)$;
3. number the vertices of T in preorder from 0 to $n-1$;
4. compute *low*(v) for each vertex v of V;
5. find the connected components of a graph with at most n vertices and m edges;
6. sort at most m numbers.

In the previous sections we have shown that Operations 1, 2, and 5 can be performed in $O(\log p)$ communication rounds, and we can use [20] for Operation 6. We now discuss Operations 3 and 4. Preorder numbering of a tree can be solved by applying the Euler tour technique of Section 3. The preorder number of a vertex in a tree is one plus the number of forward edges found in the Euler tour before encountering the vertex. The computation of $low(v)$ for each vertex v of V can be reduced to tree contraction and lowest common ancestor computation discussed in Sections 4.3 and 4.2, respectively. For each node w of T define as *label*(w) the minimum preorder label of *lca*(e) for all nontree edges incident to w. For each vertex v of V, *low*(v) is the minimum *label*(w) of all nodes w in the subtree of T rooted at v. Hence, tree contraction on T using the *min* operation computes all *low*(v) values.

LEMMA 8. *For a graph $G = (V, E)$ with n vertices and m edges, the ear decomposition, open ear decomposition, set of cut-edges, and set of cut-vertices, if they exist, can be computed on a CGM with p processors and $O((n+m)/p)$ local memory per processor,* $(n+m)/p \ge p^{\varepsilon}$ $(\varepsilon > 0)$, *using $O(\log p)$ communication rounds and $O(n/p)$ local computation per round.*

4.5. *Biconnected Components.* Testing 2-edge connectivity and biconnectivity in $O(\log p)$ rounds is a simple consequence of Lemma 8 and Theorem 3. The algorithms shown in the previous section compute the ear decomposition or open ear decomposition if G is 2-edge connected or biconnected, respectively. If G is not 2-edge connected or biconnected, then we obtain the cut-edges or cut-vertices, respectively, as follows. We observe that cut-edges are tree edges (*parent*$(v), v$) with the property that *low*$(v) >$ *preorder*(v). If G is 2-edge connected, a cut-vertex v can be detected by examining the ear numbers of all edges incident to v. The smallest of those ear numbers will occur twice, while any other of those ear numbers occurs twice if and only if v is a cut-vertex.

LEMMA 9. *The 2-edge connected and biconnected components of a connected undirected graph G with n vertices and m edges can be determined on a CGM with p processors and $O((n+m)/p)$ local memory per processor,* $(n+m)/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$, *using* $O(\log p)$ *communication rounds and* $O((n+m)/p)$ *local computation per round*.

5. Conclusion. In this paper we presented deterministic parallel CGM and BSP algorithms for the following well-known graph problems: (1) list ranking, (2) Euler tour construction in a tree, (3) computing the connected components and spanning forest, (4) lowest common ancestor preprocessing, (5) tree contraction and expression tree evaluation, (6) computing an ear decomposition or open ear decomposition, and (7) 2-edge connectivity and biconnectivity (testing and component computation). The CGM algorithms require $O(\log p)$ communication rounds and linear sequential work per round, assuming local memory $(n+m)/p \geq p^{\varepsilon}$ $(\varepsilon > 0)$ which is true for most commercially available multiprocessors. Our results imply BSP algorithms with $O(\log p)$ supersteps, $O(g \log(p)((n+m)/p))$ communication time, and $O(\log(p)((n+m)/p))$ local computation time.

The number of communication rounds obtained is independent of the problem size and grows only logarithmically with respect to p. It is still an open question whether better algorithms exist (even randomized). Our algorithm for Problem (1) improves significantly on previous results, and our algorithms for Problems (2)–(7) are the first practically relevant parallel algorithms for these standard graph problems.

Acknowledgments. The authors thank P. Flocchini, N. Santoro, and I. Rieping for their helpful discussions on the first version of this paper [5].

References

[1] R. J. Anderson, and L. Snyder, A Comparison of Shared and Nonshared Memory Models of Computation, *Proc. IEEE*, **79**(4) (1993), 480–487.

[2] D. Bader, D. Helman, and J. Jájá, Practical Parallel Algorithms for Personalized Communication and Integer Sorting, *J. Exper. Algorithms*, **1** (1996), http://www.jea.acm.org/1996.

[3] A. Bäumker and W. Dittrich, Parallel Algorithms for Image Processing: Practical Algorithms with Experiments, *Proc. International Parallel Processing Symposium*, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 429–433.

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

[5] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song, Efficient Parallel Graph Algorithms for Coarse Grained Multicomputers and BSP, *Proc.* 24*th International Colloquium on Automata, Languages and Programming* (*ICALP* '97), Lecture Notes in Computer Science, Vol. 1256, Springer-Verlag, Berlin, 1997, pp. 390–400.

[6] R. Cole, Parallel Merge Sort, *SIAM J. Comput.*, **17**(4) (1988), 770–785.

[7] R. Cole and U. Vishkin, Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time, *SIAM J. Comput.*, **17**(1) (1988).

[8] F. Dehne (editor), Special Issue on "Coarse Grained Parallel Algorithms," *Algorithmica*, **24**(3/4) (1999).

[9] F. Dehne, A. Fabri, and A. Rau-Chaplin, Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers, *Proc. ACM Symposium on Computational Geometry*, 1993, pp. 298–307.

[10] F. Dehne, A. Fabri, and C. Kenyon, Scalable and Architecture Independent Parallel Geometric Algorithms with High Probability Optimal Time, *Proc. IEEE Symposium on Parallel and Distributed Processing*, 1994, pp. 586–593.

[11] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar, A Randomized Parallel 3D Convex Hull Algorithm for Coarse Grained Multicomputers, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1995, pp. 27–33.

[12] F. Dehne and S. W. Song, Randomized Parallel List Ranking for Distributed Memory Multiprocessors, *Proc. Asian Computing Science Conference*, Singapore, Dec. 1996, Lecture Notes in Computer Science, Vol. 1179, Springer-Verlag, Berlin, 1996, pp. 1–10.

[13] X. Deng, A Convex Hull Algorithm for Coarse Grained Multiprocessors, *Proc. International Symposium on Algorithms and Computation*, 1994.

[14] X. Deng and P. Dymond, Efficient Routing and Message Bounds for Optimal Parallel Algorithms, *Proc. International Parallel Processing Symposium*, 1995.

[15] X. Deng and N. Gu, Good Programming Style on Multiprocessors, *Proc. IEEE Symposium on Parallel and Distributed Processing*, 1994, pp. 538–543.

[16] G. A. Dirac, On Rigid Circuit Graphs, *Abh. Math. Sem. Univ. Hamburg*, **25** (1961), 71–76.

[17] A. Ferreira, A. Rau-Chaplin, and S. Ubeda, Scalable 2D Convex Hull and Triangulation Algorithms for Coarse-Grained Multicomputers, *Proc. IEEE Symposium on Parallel and Distributed Processing*, San Antonio, IEEE Press, New York, 1995, pp. 561–569.

[18] A. V. Gerbessiotis and L. G. Valiant, Direct Bulk-Synchronous Parallel Algorithms, *Proc. Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, Vol. 621, Springer-Verlag, Berlin, 1992, pp. 1–18.

[19] P. B. Gibson, Y. Matias, and V. Ramachandran, The Queue-Read Queue-Write Asynchronous PRAM Model, *Proc. 9th ACM SPAA*, 1997, pp. 72–83.

[20] M. T. Goodrich, Communication Efficient Parallel Sorting, *ACM Symposium on Theory of Computing*, 1996.

[21] J. Jájá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[22] B. H. H. Juurlink and H. A. G. Wijshoff, The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model, *Proc. Euro-Par* '96, vol. II, Lecture Notes in Computer Science, Vol. 1124, Springer-Verlag, Berlin, 1996.

[23] P. Klein, Efficient Parallel Algorithms for Chordal Graphs, *Proc. IEEE Symposium on Foundations of Computer Science*, 1989, pp. 150–161.

[24] P. Klein, Parallel Algorithms for Chordal Graphs, in [32], pp. 341–407.

[25] Hui Li and K. C. Sevcik, Parallel Sorting by Overpartitioning, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1994, pp. 46–56.

[26] Y. Maon, B. Schieber, and U. Vishkin, Parallel Ear Decomposition Search (EDS) and ST-Numbering in Graphs, *Theoret. Comput. Sci.*, **47** (1986), 277–298.

[27] G. L. Miller and J. H. Reif, Parallel Tree Contraction and Its Application, *Proc. IEEE Symposium on Foundations of Computer Science*, 1985, pp. 478–489.

[28] G. L. Miller and V. Ramachandran, Efficient Parallel Ear Decomposition with Applications, Manuscript, MSRI, Berkeley, January 1986.

[29] K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[30] V. Ramachandran, Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity, in [32], pp. 276–340.

[31] M. Reid-Miller, List Ranking and List Scan on the Cray C-90, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1994, pp. 104–113.

[32] J. H. Reif (editor), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, CA, 1993.

[33] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs, *SIAM J. Comput.*, **5** (1976), 266–283.

[34] Y. Shiloch and U. Vishkin, An $O(\log n)$ Parallel Connectivity Algorithm, *J. Algorithms*, **3**(1) (1983), 57–67.

[35] J. F. Sibeyn, Better Trade-Offs for Parallel List Ranking, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 221–230.
[36] L. Snyder, Type Architectures, Shared Memory and the Corollary of Modest Potential, *Annu. Rev. Comput. Sci.*, **1** (1986), 289–317.
[37] R. E. Tarjan and U. Vishkin, An Efficient Parallel Biconnectivity Algorithm, *SIAM J. Comput.*, **14**(4) (1985), 862–874.
[38] L. Valiant, A Bridging Model for Parallel Computation, *Comm. ACM*, **33**(8) (1990).
[39] L. G. Valiant et al., General Purpose Parallel Architectures, in *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen, MIT Press/Elsevier, Cambridge, MA/Amsterdam, 1990, pp. 943–972.
[40] H. Whitney, Non-Separable and Planar Graphs, *Trans. Amer. Math. Soc.*, **34** (1932), 339–362.