Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP ⋆

Extended Abstract

E. Cáceres[1], F. Dehne[2], A. Ferreira[3], P. Flocchini[4], I. Rieping[5], A. Roncato[6], N. Santoro[7], and S. W. Song[8]

[1] Univ. Federal de Mato Grosso do Sul, Campo Grande, Brasil, *edson@dct.ufms.br*
[2] Carleton Univ., Ottawa, Canada, *dehne@scs.carleton.ca*
[3] ENS Lyon, Lyon, France, *ferreira@lip.ens-lyon.fr*
[4] Univ. de Montreal, Montreal, Canada, *flocchin@iro.umontreal.ca*
[5] Univ. Paderborn, Paderborn, Germany, *inri@uni-paderborn.de*
[6] Facolta di Scienze Mat. Fis. e Nat., Mestre, Italia, *roncato@dsi.unive.it*
[7] Carleton Univ., Ottawa, Canada, *santoro@scs.carleton.ca*
[8] Univ. of São Paulo, São Paulo, Brazil, *song@ime.usp.br*

Abstract. In this paper, we present *deterministic* parallel algorithms for the coarse grained multicomputer (CGM) and bulk-synchronous parallel computer (BSP) models which solve 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), and (8) cordal graph recognition (finding a perfect elimination ordering). The algorithms for Problems 1-7 require $O(\log p)$ communication rounds and linear sequential work per round. Our results for Problems 1 and 2 hold for arbitrary ratios $\frac{n}{p}$, i.e. they are *fully scalable*, and for Problems 3-8 it is assumed that $\frac{n}{p} \geq p^{\epsilon}$, $\epsilon > 0$, which is true for all commercially available multiprocessors. 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 Problem 1 is a considerable improvement over those previously reported. The algorithms for Problems 2-7 are the first practically relevant deterministic parallel algorithms for these problems to be used for commercially available coarse grained parallel machines.

⋆ Research 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).

1 Introduction

The Models: Speedup results for theoretical PRAM algorithms do not necessarily match the speedups observed on real machines [2] [31]. Given sufficient slackness in the number of processors, Valiant's BSP approach [34] 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 [33] [34]. The BSP approach requires that g (= local computation speed / router bandwidth) is low, or fixed, even for increasing number of processors. Gerbessiotis and Valiant [17] describe circumstances where PRAM simulations can not 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 [34], 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. [8]). We are therefore also using a more practical version of the BSP model, referred to as the *coarse grained multicomputer* model (CGM) [8], [9], [10]. 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. In the BSP model, a computation/communication round is equivalent to a superstep with $L = \frac{n}{p}g$ (plus the above "packing requirement").

Finding an optimal algorithm in the coarse grained multicomputer model (CGM) is equivalent to minimizing the number of communication rounds as well as the total local computation time. 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 ([33] [34] [13]). The above model has been used (explicitly or implicitly) in parallel algorithm design for various problems ([4], [8], [9], [14], [12], [22], [10]) and shown very good practical timing results.

The 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)
8. cordal graph recognition, finding a perfect elimination ordering

These problems have been extensively studied for the PRAM (see e.g. [28]) and for fine-grained parallel network models of computation (see e.g. [1]). 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.

Reid-Miller's [27] 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 [27] required $O(\log n)$ communication rounds. In [11], an improved algorithm was presented which required, with high probability, only $O(k \log p)$ rounds, where $k \leq \log^* n$. In [13], $O(\log p)$ communication rounds are achieved by a randomized algorithm. Bäumker and Dittrich [3] presented 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.

We improve these results by giving the first *deterministic* algorithms for list ranking and computing connected components 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. Algorithms with $O(1)$ rounds have been presented for various Computational Geometry problems [8, 9, 10, 11, 16], but the graph problems studied in this paper have considerably less "internal structure" which could be exploited to obtain such solutions. 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 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 [27] we will, in general, assume that $n >> p$ (coarse grained), because this is usually the case in practice. Note, however, that our results for Problems 1 and 2 hold for arbitrary ratios $\frac{n}{p}$. Goodrich [18] calls such algorithms *fully scalable*. For Problems 3-8 we will assume that $\frac{n}{p} \geq p^{\epsilon}$, $\epsilon > 0$, which is true for all commercially available multiprocessors.

2 List Ranking

Let L be a list represented by a vector s s.t. $s[i]$ 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 [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 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 naive 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 straight forward 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 1 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.

— End of Algorithm —

We first prove that the set of elements selected at the end of Algorithm 1 is of size at most $O(n/p)$.

Lemma 1. *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.*

Proof. Due to space limitations, the proof is omitted. It can be found in the full version of this paper [5].

In order to show that subsequent elements selected at the end of Algorithm 1 have distance at most $O(p^2)$, we need the following lemmas.

Lemma 2. *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 $O(p)$.*

Proof. Due to space limitations, the proof is omitted. It can be found in the full version of this paper [5].

Lemma 3. *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.*

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

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

Proof. Follows from Lemma 2 and Lemma 3.

In summary, we obtain

Theorem 5. *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.*

3 Euler Tour in a Tree

Let $T = (V, E)$ be an undirected tree and $T^* = (V, E^*)$ be a directed graph with $E^* = \{(v, w), (w, v) | \{v, w\} \in E\}$. Thus, T^* is Eulerian because $indegree(v) = outdegree(v)$ for each vertex v. The Euler Tour problem for T consists of computing for T^* a path that traverses each edge exactly once and returns to its starting point, as well as for each vertex its rank in this path.

Theorem 6. *The Euler Tour of a tree T with n vertices can be computed 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.*

Proof. Due to space limitations, the algorithm and proof are omitted. They can be found in the full version of this paper [5].

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.

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 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 JaJa's *accelerated cascading* technique for the PRAM [19].)

Steps 1 and 2 of Algorithm 2 simulate Shiloch and Vishkin's PRAM algorithm [30], 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 2 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 Section 3 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 $parent(u) \in V'$.)

— End of Algorithm —

Lemma 7. [30] *The number of different trees after iteration k of Step 2 is bounded by $(\frac{2}{3})^k n$.*

We obtain

Theorem 8. *Algorithm 2 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. Due to space limitations, the proof is omitted. It can be found in the full version of this paper [5].

5 Other Graph Problems

In the remainder, we summarize our solutions for Problems 4-8. Due to space limitations, the algorithms and proofs are omitted. They can be found in the full version of this paper [5].

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. The problem of preprocessing T in order to answer a query $LCA(u,v)$ quickly for any pair (u,v) is called the *lowest-common-ancestor* (LCA) problem.

Theorem 9. *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.*

Tree Contraction and Expression Tree Evaluation: We observe that the classical tree contraction and expression tree evaluation algorithm of [24] can be easily implemented on a CGM to run in $O(\log p)$ communication rounds.

Observation 1 *Tree contraction and expression tree evaluation on a tree T with n nodes can be performed on a CGM with p processors and $O(\frac{n}{p})$ local memory per processor, $\frac{n}{p} \geq p^{\epsilon}$ ($\epsilon > 0$), using $O(\log p)$ communication rounds and $O(\frac{n}{p})$ local computation per round.*

Open Ear Decomposition and Biconnected Components: 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 $P_1,\ldots,P_r$ 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 $P_1 \cup \ldots \cup P_{i-1}$ 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 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.

Theorem 10. *For a graph $G=(V,E)$ with n vertices and m edges, the ear decomposition, open ear decomposition, as well as its 2-edge connected and biconnected components can be computed on a CGM with p processors and $O(\frac{n+m}{p})$ local memory per processor using $O(\log p)$ communication rounds and $O(\frac{n}{p})$ local computation per round.*

Chordal Graph Recognition: A graph $G=(V,E)$ is *chordal*, if every cycle of length greater than three has a *chord*, i.e., an edge connecting two non-consecutive nodes of the cycle. A *simplicial* node is a node whose neighbors form a clique. Dirac [15] showed that every chordal graph has a simplicial node. It is easy to see that removing an arbitrary node from a chordal graph yields another chordal graph. Therefore, after removing the simplicial node of a chordal graph, the new graph has another simplicial node. Successively removing all simplicial nodes gives an ordering of the nodes of G. This ordering is called *perfect elimination ordering* (*PEO*).

Theorem 11. *Finding the PEO of a given graph $G = (V, E)$ with n vertices and m edges can be solved 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 $O(\log n \log p)$ communication rounds and $O(\frac{n+m}{p})$ local computation per round.*

References

1. S.G. Akl, *Parallel Computation*, Prentice Hall, 1997.
2. R.J. Anderson, and L. Snyder, "A Comparison of Shared and Nonshared Memory Models of Computation," in Proc. of the IEEE, 79(4), pp. 480-487.
3. A. Bäumker and W. Dittrich, "Parallel Algorithms for Image Processing: Practical Algorithms with Experiments," *International Parallel Processing Symposium*, IEEE Computer Society Press, 1996, pp. 429 - 433.
4. G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, "A Comparison of Sorting Algorithms for the Connection Machine CM-2.," in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 3-16.
5. E. Cáceres, 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,", on-line Postscript at http://www.scs.carleton.ca/scs/faculty/dehne.html.
6. R. Cole, "Parallel merge sort," SIAM J. Comput., 17(4), 1988, pp. 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 Journal of Computing*, Vol. 17, No. 1, 1988.
8. F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers," in Proc. *ACM 9th Annual Computational Geometry*, pages 298–307, 1993.
9. F. Dehne, A. Fabri, and C. Kenyon, "Scalable and Architecture Independent Parallel Geometric Algorithms with High Probability Optimal Time," in Proc. *6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.
10. F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar, "A randomized parallel 3D convex hull algorithm for coarse grained multicomputers," in Proc. *ACM Symposium on Parallel Algorithms and Architectures* (SPAA'95), pp. 27–33, 1995.
11. F.Dehne, S.W. Song, "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.
12. X. Deng, "A Convex Hull Algorithm for Coarse Grained Multiprocessors," in Proc. 5th International Symposium on Algorithms and Computation, 1994.
13. X. Deng and P. Dymond, "Efficient Routing and Message Bounds for Optimal Parallel Algorithms," in Proc. Int. Parallel Proc. Symp., 1995.

14. X. Deng and N. Gu, "Good Programming Style on Multiprocessors," in Proc. IEEE Symposium on Parallel and Distributed Processing, 1994, pp. 538-543.
15. G.A. Dirac. "On rigid circuit graphs". *Abh. Math. Sem. Univ. Hamburg* 25, 1961, pp. 71–76.
16. A. Ferreira, A. Rau-Chaplin, and S. Ubeda, "Scalable 2d convex hull and triangulation algorithms for coarse-grained multicomputers," in *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing – SPDP'95*, pages 561–569, San Antonio (USA), October 1995. IEEE Press.
17. A.V. Gerbessiotis and L.G. Valiant, "Direct Bulk-Synchronous Parallel Algorithms," in Proc. 3rd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, Vol. 621, 1992, pp. 1-18.
18. M.T. Goodrich, "Communication efficient parallel sorting," ACM Symposium on Theory of Computing (STOC), 1996.
19. Ja'Ja', *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.
20. P. Klein. "Efficient Parallel Algorithms for Chordal Graphs". *Proc. 29th Symp. Found. of Comp. Sci., FOCS* 1989, pp. 150–161.
21. P. Klein. "Parallel Algorithms for Chordal Graphs". In *Synthesis of parallel algorithms*, J. H. Reif (editor). Morgan Kaufmann Publishers, 1993, pp. 341–407.
22. Hui Li, and K. C. Sevcik, "Parallel Sorting by Overpartitioning," in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 46-56.
23. Y. Maon, B. Schieber, U. Vishkin. "Parallel ear decomposition search (EDS) and st-numbering in graphs". *Theoretical Computer Science*, vol. 47, 1986, pp. 277 - 298.
24. G.L. Miller, J.H. Reif, "Parallel tree contraction and its application," IEEE Symp. on Foundations of Computer Science, 1985, pp. 478–489.
25. G. L. Miller, V. Ramachandran. "Efficient parallel ear decomposition with applications", manuscript, MSRI, Berkeley, January 1986.
26. V. Ramachandran. "Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity", in [28], pp. 276 - 340.
27. M. Reid-Miller, "List ranking and list scan on the Cray C-90," in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 104-113.
28. J. H. Reif (editor), *Synthesis of parallel algorithms*, Morgan Kaufmann Publishers, 1993.
29. D.J. Rose, R.E. Tarjan, and G.S. Lueker. "Algorithmic Aspects of Vertex Elimination on Graphs". *SIAM J. Comp.* 5, 1976, pp. 266–283.
30. Y. Shiloch, U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, 3(1), pp. 57–67, 1983.
31. L. Snyder, '"Type architectures, shared memory and the corollary of modest potential," *Annu. Rev. Comput. Sci. 1*, 1986, pp. 289-317.
32. R.E. Tarjan, U. Vishkin, "An efficient parallel biconnectivity algorithm," SIAM J. Comput., 14(4), 1985, pp. 862–874.
33. L. Valiant, "A bridging model for parallel computation," Communications of the ACM, Vol. 33, No. 8, August 1990.
34. L.G. Valiant et al., "General Purpose Parallel Architectures," *Handbook of Theoretical Computer Science*, Edited by J. van Leeuwen, MIT Press/Elsevier, 1990, pp.943-972.
35. H. Whitney. "Non-separable and planar graphs". *Trans. Amer. Math. Soc.* 34, 1932, pp. 339 - 362.