# Techniques for Designing Efficient Parallel Graph Algorithms for SMPs and Multicore Processors

Guojing Cong[1] and David A. Bader[2,⋆]

[1] IBM TJ Watson Research Center, Yorktown Heights, NY, 10598
gcong@us.ibm.com
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332
bader@cc.gatech.edu

**Abstract.** Graph problems are finding increasing applications in high performance computing disciplines. Although many regular problems can be solved efficiently in parallel, obtaining efficient implementations for irregular graph problems remains a challenge. We propose techniques for designing and implementing efficient parallel algorithms for graph problems on symmetric multiprocessors and chip multiprocessors with a case study of parallel tree and connectivity algorithms. The problems we study represent a wide range of irregular problems that have fast theoretic parallel algorithms but no known efficient parallel implementations that achieve speedup without serious restricting assumptions about the inputs. We believe our techniques will be of practical impact in solving large-scale graph problems.

**Keywords:** Spanning Tree, Minimum Spanning Tree, Biconnected Components, Shared Memory.

## 1 Introduction

Graph theoretic problems arise in several traditional and emerging scientific disciplines such as VLSI design, optimization, databases, and computational biology. There are plenty of theoretically fast parallel algorithms, for example, optimal PRAM algorithms, for graph problems; however, in practice few parallel implementations beat the best sequential implementations for arbitrary, sparse graphs. Much previous algorithm design effort aimed at reducing the complexity factors with work-time optimality as the ultimate goal. In practice this often yields complicated algorithms with large hidden constant factors that frustrate attempt of implementation on real parallel machines [1,2].

Modern symmetric multiprocessors (SMPs) and chip multiprocessors (CMPs) are becoming very powerful and common place. Most of the high performance computers are clusters of SMPs and/or CMPs. PRAM algorithms for graph problems can be emulated much easier and more efficiently on SMPs than on distributed memory platforms because shared memory allows for fast, concurrent access to an irregular datastructure that is often difficult to partition well for distributed memory systems. Unfortunately,

⋆ This work is supported in part by NSF Grants CAREER CCF-0611589, NSF DBI-0420513, ITR EF/BIO 03-31654, IBM Faculty Fellowship and Microsoft Research grants, and DARPA Contract NBCH30390004.

I. Stojmenovic et al. (Eds.): ISPA 2007, LNCS 4742, pp. 137–147, 2007.

emulation – even with aggressive algorithm engineering efforts – oftentimes does not produce parallel implementations that beat the best sequential implementation. There remains a big gap between the algorithmic model and SMPs. Architectural factors including memory hierarchy and synchronization cost are hard to encompass into a parallel model, yet they greatly affect the performance of a parallel algorithm. For many parallel models, it is hard to predict for a given problem whether an $O(\log n)$ algorithm runs faster than an $O(\log^2 n)$ algorithm on any real architecture. New design paradigm that considers these factors are called for.

We have conducted extensive study on the design of practical parallel algorithms for graph problems that run fast on SMPs. Our parallel implementations for the spanning tree, minimum spanning tree (MST), biconnected components, and other problems, achieved for the first time good parallel speedups for sparse, arbitrary inputs on SMPs [3,4,5]. In this paper we present techniques that are proven to be effective either by our experimental study or by theoretic analysis on realistic models for fast graph algorithm design. These techniques address factors of modern architectures that are transparent to most parallel models but crucial to performance. For example, we propose asynchronous algorithms for reducing the synchronization overhead and I/O efficient PRAM simulation for designing cache-friendly parallel algorithms with irregular graph inputs. The problems we study are fundamental in graph theory. We believe our techniques can help build efficient parallel implementations for a wide range of graph algorithms on SMPs, and future manycore systems.

The rest of the paper is organized as follows. The remainder of Section 1 summarizes previous studies of experimental results and our new results. Sections 2, 3 and 4 discuss the impact of limited number of processors, synchronization and multiple levels of memory hierarchy on performance, respectively, and present parallel techniques for the corresponding scenarios. Section 5 presents an example for reducing the constants in algorithmic overhead. In Section 6 we conclude and give future work. Throughout the paper, we use $n$ and $m$ to denote the number of vertices and the number of edges of an input graph $G = (V, E)$, respectively.

### 1.1 Previous Studies and Results

Different variations of the spanning tree and connected components algorithms based on the "graft-and-shortcut" approach have been implemented (see, e.g., [6,7,8,9]) on a number of platforms including Cray Y-MP/C90, TMC CM-2 and CM-5, and Maspar MP-1. These implementations achieve moderate speedups for regular inputs, and are slower than the best sequential implementation for irregular inputs.

Chung and Condon [10] implemented parallel Borůvka's algorithm on the CM-5. Dehne and Götz [11] studied practical parallel algorithms for MST using the BSP model. The parallel implementations either do not beat the sequential implementation, or are not appropriate for handling large, sparse graphs.

Woo and Sahni [12] presented an experimental study of computing biconnected components on a hypercube. They use an adjacency matrix as the input representation that wastes a huge amount of memory for sparse graphs.

### 1.2 Our Results

We have developed a number of practical parallel algorithms [3,4,5]. Our methodology for designing these parallel algorithms is based upon an in-depth study of available PRAM approaches and techniques as discussed in this paper. Our spanning tree algorithm is the first to achieve good parallel speedups for the notoriously hard arbitrary, sparse instances [3]; our MST algorithm is the only one known that achieves parallel speedups for all test inputs [4]; our biconnected components algorithm filters out edges that are not essential in computing the components, and runs fast with relatively low memory consumption compared with previous implementations [5]. Table 1 summarizes our performance results on SUN Enterprise 4500 with 10 processors. All the input graphs have $1M$ vertices but different number of edges. Comparing against the best sequential implementation, the "prior speedup" column shows the speedups we achieve for our implementation of the best performing algorithms published prior to our study, and the "our speedup" column shows the speedups we achieve for algorithms designed with techniques presented in this paper. We see that prior algorithms run slower than the best sequential implementation with a moderate number of processors, while our new algorithms achieve good parallel speedups.

**Table 1.** Performance comparison of our algorithms and best performing previously published algorithms. Here $1M = 1048576$, $n = 1M$, and $m$ is the number of edges.

| Problem | Input Type | Edges($m$) | Prior speedup | Our speedup |
|---|---|---|---|---|
| Spanning Tree | random graph | $20M$ | 0.36 | 3.1 |
| | Torus | $4M$ | 0.5 | 2.6 |
| Minimum Spanning Tree | random graph | $20M$ | 0.46 | 4.5 |
| | Torus | $4M$ | 2.2 | 3.7 |
| Biconnected Components | random graph | $20M$ | 0.72 | 3.8 |

## 2 Adapting to the Available Parallelism

Nick's Class ($\mathcal{NC}$) is defined as the set of all problems that run in polylog-time with a polynomial number of processors. Whether a problem $P$ is in $\mathcal{NC}$ is a fundamental question. The PRAM model assumes an unlimited number of processors, and explores the maximum inherent parallelism of $P$. Although several massively parallel supercomputers now have thousands of processors, the number of processors available to a parallel program is still nowhere near the size of the problem. Acknowledging the practical restriction of limited parallelism provided by real computers, Kruskal *et al.* [13] argued that non poly-logarithmic time algorithms (e.g., sublinear time algorithms) could be more suitable than polylog algorithms for implementation with practically large input size. We observe this is still true for current problems and parallel machines. In practice if an $\mathcal{NC}$ algorithm does not perform well, it is worthwhile to consider $\mathcal{EP}$ (short for *efficient parallel*) algorithms which by the definition in [13] is the class of algorithms that achieve a polynomial reduction in running time with a poly-logarithmic inefficiency. More formally, let $T(n)$ and $P(n)$ be the running time of a parallel algorithm and the number of processors employed, and $t(n)$ be the running time of the best sequential algorithm, then $\mathcal{EP}$ is the class of algorithms that satisfies

$$T(n) \leq t(n) \text{ and } T(n) \cdot P(n) = O(t(n)).$$

We refer to $\mathcal{EP}$ as a class of algorithms where the design focus is shifted from reducing the complexity factors to solving problems of realistic sizes efficiently with a limited number of processors.

Many $\mathcal{NC}$ algorithms take drastically different approaches than their respective sequential algorithms and incur significant parallel overhead. For many algorithms it could take impractically large problem sizes to show any performance advantage. The $\mathcal{EP}$ algorithms, on the other hand, can have advantages coming from limited parallelism, i.e., larger granularity of parallelism, and hence less synchronization. $\mathcal{EP}$ algorithms also tend to work better with distributed-memory environments (e.g., many practical LogP and BSP algorithms [14,15]).

We propose a technique that blends the $\mathcal{EP}$ and $\mathcal{NC}$ approaches so that an algorithm adapts to the number of available processors. The implementation is efficient when there are few processors such as in a multicore chip or a small SMP, and provides enough parallelism when scaling to a larger number of processors. As a case study next we present the design of our MST algorithm – MST-BC.

Many parallel MST algorithms in the literature are based on Borůvka's approach [16,4]. These algorithms run in $O(\log^k n)$ ($k$ is a constant) time with $O(n)$ processors. Performance in general is not good on current SMPs. MST-BC uses multiple, coordinated instances of Prim's sequential algorithm running on the graph's shared data structure. In fact, it marries Prim's algorithm with that of the naturally parallel Borůvka approach. The basic idea of MST-BC is to let each processor simultaneously run Prim's algorithm from different starting vertices. Each processor keeps growing its subtree when there exists a lightweight edge that connects the tree to a vertex not yet in another tree, and the subtree matures when it can grow no further. The lightest incident edges are then found for the isolated vertices (the Borůvka step). Special care is taken to ensure the isolation of mature subtrees that is crucial to the correctness of the algorithm. When all of the vertices have been incorporated into mature subtrees, each subtree is contracted into a supervertex, and the approach is called recursively until only one supervertex remains. We refer interested readers to [4] for details of the algorithm and proof of correctness.

MST-BC is adaptive to the available number of processors. When there are $n$ processors available, each processor can grow only a subtree of one edge in an iteration, MST-BC behaves exactly as parallel Borůvka's algorithm; when there is only one processor available, MST-BC degenerates to Prim's algorithm. The interesting case is that when $p$ ($p \ll n$) processors are available, multiple instances of Prim's algorithm runs in each iteration alternated with Borůvka steps. MST-BC performs well when the graph is large and sparse and the number of processors is small compared with the input size. The technique of adapting to the number of available processors is also applicable in the case of parallel spanning tree.

## 3 Reducing Synchronization Cost

Synchronization is achieved by using barriers or locks. Synchronization is crucial to correctness and performance. In Section 3.1 we present a technique called lazy synchronization that aggressively reduces the amount of synchronization primitives in an

algorithm. Section 3.2 shows how to trade a large number of expensive locking operations with a small number of barrier operations.

### 3.1 Lazy Synchronization

Barriers can be placed after each statement of the program to ensure PRAM-like synchronous execution. For many algorithms, however, this practice will seriously degrade the performance. Barriers are only needed at steps where events should be observed globally.

When adapting a PRAM algorithm to SMPs, a thorough understanding of the algorithm usually suffices to eliminate unnecessary barriers. Reducing synchronization is a pure implementation issue. Lazy synchronization, on the other hand, is an algorithm design technique. Lazy synchronization means much more than inserting a minimum number of synchronization primitives. More importantly, we design algorithms that are as asynchronous as possible. Reduced level of synchronization in general yields greater granularity of parallelism. Lazy synchronization also allows nondeterministic intermediate results but deterministic solutions. For example, in MST-BC, the selection of a light edge is nondeterministic, while we are guaranteed that the answer is one of the possible spanning trees of minimum weight. In a parallel environment, to ensure correct final results oftentimes we do not need to define a total ordering on all the events occurred, and a partial ordering in general suffices. Relaxed constraints on ordering reduce the number of synchronization primitives in the algorithm. Application of lazy synchronization generally involves a careful arrangement of read/write operations to reach consensus with proofs of correctness.

Here we take the spanning tree problem as a case study. Our parallel spanning tree algorithm for shared-memory multicores and multiprocessors has two main steps: 1) stub spanning tree, and 2) work-stealing graph traversal. In the first step, one processor generates a stub spanning tree, that is, a small portion of the spanning tree by randomly walking the graph for $O(p)$ steps. The vertices of the stub spanning tree are evenly distributed into each processor's queue, and each processor in the next step will traverse from the first element in its queue. After the traversals in step 2, the spanning subtrees are connected to each other by this stub spanning tree. In the graph traversal step, each processor traverses the graph (by coloring the nodes) similar to the sequential algorithm in such a way that each processor finds a subgraph of the final spanning tree. Work-stealing is used to balance the load for graph traversal (We refer interested readers to [3] for details of work-stealing).

One problem related to synchronization that we have to address is that there could be portions of the graph traversed by multiple processors and be in different subgraphs of the spanning tree. The immediate remedy is to synchronize using either locks or barriers. With locks, coloring the vertex becomes a critical section, and a processor can only enter the critical section when it gets the lock. Although the nondeterministic behavior is now prevented, it does not perform well on large graphs due to an excessive number of locking and unlocking operations.

In our algorithm there are no barriers introduced in graph traversal. As we will show the algorithm runs correctly without barriers even when two or more processors color the same vertex. In this situation, each processor will color the vertex and set as its

parent the vertex it has just colored. Only one processor succeeds at setting the vertex's parent to a final value. For example, using Fig. 1, processor $P_1$ colored vertex $u$, and processor $P_2$ colored vertex $v$, at a certain time they both find $w$ unvisited and are now in a race to color vertex $w$. It makes no difference which processor colored $w$ last because $w$'s parent will be set to either $u$ or $v$ (and it is legal to set $w$'s parent to either of them; this will not change the validity of the spanning tree, only its shape). Further, this event does not create cycles in the spanning tree. Both $P_1$ and $P_2$ record that $w$ is connected to each processor's own tree. When each of $w$'s unvisited children are visited by various processors, its parent will be set to $w$, independent of $w$'s parent.

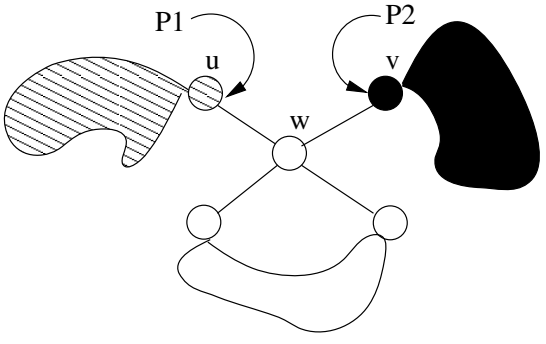


**Fig. 1.** Two processors $P_1$ and $P_2$ see vertex $w$ as unvisited, so each is in a race to color $w$ and set $w$'s parent pointer. The shaded area represents vertices colored by $P_1$, the black area represents those marked by $P_2$, and the white area contains unvisited vertices.

**Lemma 1.** *On a shared-memory parallel computer with sequential memory consistency, the spanning tree algorithm does not create any cycles.*

We refer interested readers to [3] for details of the proof. Note that different runs of the algorithm may produce trees of different topologies, yet each is a correct spanning tree.

### 3.2 Barriers and Locks

Locks and barriers are meant for different types of synchronizations. In practice, the choice of using locks or barriers may not be very clear. Take the "graft and shortcut" spanning tree algorithm for example. For graph $G = (V, E)$ represented as an edge list, we start with $n$ isolated vertices and $2m$ processors. For edge $e_i = \langle u, v \rangle$, processor $P_i$ ($1 \leq i \leq m$) inspects $u$ and $v$, and if $v < u$, it grafts vertex $u$ to $v$ and labels $e_i$ to be a spanning tree edge. The problem here is that for a certain vertex $v$, its multiple incident edges could cause grafting $v$ to different neighbors, and the resulting tree may not be valid (note that Shiloach-Vishkin's original algorithm is based on priority CRCW [17]). To ensure that $v$ is only grafted to one of the neighbors, locks can be used. Associated with each vertex $v$ is a flag variable protected by a lock that shows whether $v$ has been grafted. In order to graft $v$ a processor has to obtain the lock and check the flag, thus race conditions are prevented. A different solution uses barriers [18] in a two-phase election. No checking is needed when a processor grafts a vertex, but after all processors are done

(ensured with barriers), a check is performed to determine which one succeeds and the corresponding edge is labeled as a tree edge. There is no clear winner between the two synchronization scheme for performance. Whether to use a barrier or lock is dependent on the algorithm design as well as the barrier and lock implementations. Locking typically introduces large memory overhead. When contention among processors is intense, the performance degrades seriously.

## 4 Cache Friendly Design

The increasing speed difference between processor and main memory makes cache and memory access patterns important factors for performance. The number (and pattern) of memory accesses could be the dominating factor of performance instead of computational complexities. For sequential algorithms, there have emerged quite a number of cache-aware and cache-oblivious algorithms that are shown to have good cache performance (see, e.g.[19,20]).

The fact that modern processors have multiple levels of memory hierarchy is generally not reflected by most of the parallel models. As a result, few parallel algorithm studies have touched on the cache performance issue. The SMP model proposed by Helman and JáJá is the first effort to model the impact of memory access and cache over an algorithm's performance [21]. The model forces an algorithm designer to reduce the number of non-contiguous memory accesses. However, it does not give hints to the design of cache-friendly parallel algorithms. Park *et al.* [22] proposed adjacency array representation of graphs that is more cache-friendly than adjacency list, and the access pattern can still be erratic. Simultaneous memory accesses to "random" memory locations determined by the irregular structure of the input make it hard to come up with cache friendly designs.

Chiang *et al.* [23] presented a *PRAM simulation* technique for designing and analyzing efficient external-memory (sequential) algorithms for graph problems. This technique simulates the PRAM memory by keeping a task array of $O(N)$ on disk in $O(scan(N))$ blocks. For each PRAM step, the simulation sorts a copy of the contents of the PRAM memory based on the indices of the processors for which they will be operands, and then scans this copy and performs the computation for each processor being simulated. The following can be easily shown:

**Theorem 1.** *Let A be a PRAM algorithm that uses N processors and $O(N)$ space and runs in time T. Then A can be simulated in $O(T \cdot sort(N))$ I/Os [23].*

Here $sort(N)$ represents the optimal number of I/Os needed to sort $N$ items striped across the disks, and $scan(N)$ represents the number of I/Os needed to read $N$ items striped across the disks. Specifically,

$$sort(x) = \frac{x}{DB} \log_{\frac{M}{B}} \frac{x}{B}$$

$$scan(x) = \frac{x}{DB}$$

where M = # of items that can fit into main memory, B = # of items per disk block, and D = # of disks in the system.

We observe that a similar technique can be applied to the cache-friendly parallel implementation of PRAM algorithms. I/O efficient algorithms exhibit good spatial locality behavior that is critical to good cache performance. We apply the *PRAM simulation* technique to the efficient parallel implementation. Instead of having one processor simulate the PRAM step, we have $p \ll n$ processors perform the simulation concurrently. The simulated PRAM implementation is expected to incur few cache block transfers between different levels. For small input sizes it would not be worthwhile to apply this technique as most of the data structures can fit into cache. As the input size increases, the cost to access memory becomes more significant, and applying the technique becomes beneficial.

Our experimental studies show that on current SMPs oftentimes memory writes have larger impact than reads on the performance of the algorithm [4]. This is because with the snoopy-cache consistency protocol for current SMPs, memory writes tend to generate more protocol transactions than memory reads and concurrent writes also create consistency issues of memory management for the operating system and the corresponding memory management is of higher cost than reads. We could have incorporated this fact into a parallel model by separating the number of memory reads and writes as different parameters, but then other questions like how to weigh their different importance would emerge. Here the message is that for graph problems that are generally irregular in nature, when standard techniques of optimizing the algorithm for cache performance do not apply, trading memory reads for writes might be an option. For a detailed example, we refer interested readers to the parallel Borůvka's implementation described in [4].

## 5 Algorithmic Optimizations

For most problems, parallel algorithms are inherently more complicated than the sequential counterparts, incurring large overheads with many algorithm steps. Instead of lowering the asymptotic complexities, in many cases we can reduce the constant factors and improve performance. Algorithmic optimizations are problem specific. We demonstrate the benefit of such optimizations with our biconnected components algorithm [5].

We developed a new algorithm that eliminates edges that are not essential in computing the biconnected components. For any input graph, edges are first eliminated before the computation of biconnected components is done so that at most $min(m, 2n)$ edges are considered. Although applying the filtering algorithm does not improve the asymptotic complexity, in practice, the performance of the biconnected components algorithm can be significantly improved.

We say an edge $e$ is *non-essential* for biconnectivity if removing $e$ does not change the biconnectivity of the component to which it belongs. Filtering out *non-essential* edges when computing biconnected components (of course we will place these edges back in later) may produce performance advantages. Recall that the Tarjan-Vishkin algorithm (TV) is all about finding the equivalence relation $R'^{*}_{c}$ [18,16]. Of the three conditions for $R'_c$, it is trivial to check for condition 1 which is for a tree edge and a non-tree edge. Conditions 2 and 3, however, are for two tree edges and checking

involves the computation of *high* and *low* values. To compute *high* and *low*, we need to inspect every nontree edge of the graph, which is very time consuming when the graph is not extremely sparse. The fewer edges the graph has, the faster the *Low-high* step. Also when we build the auxiliary graph, the fewer edges in the original graph means the smaller the auxiliary graph and the faster the *Label-edge* and *connected-components* steps.

Suppose $T$ is a BFS tree, then we have

**Theorem 2.** *The edges of each connected component of $G-T$ are in one biconnected component [5].*

Combining our algorithm for eliminating *non-essential* edges and TV, the new biconnected components algorithm runs in $max(O(d), O(\log n))$ time with $O(n)$ processors on CRCW PRAM, where $d$ is the diameter of the graph. Asymptotically the new algorithm is not faster than TV. In practice, however, we achieve parallel speedups upto 4 with 12 processors on SUN Enterprise 4500 using the filtering technique. This is remarkable, given that the sequential algorithm runs in linear time with a very small hidden constant in the asymptotic complexity.

## 6 Conclusion and Future Work

We present algorithm design and engineering techniques for running parallel algorithms on SMPs, multicore, and manycore processors, with case studies taken from tree and connectivity problems. Our implementations are the first that achieve good speedups over a wide range of inputs. PRAM algorithms provide a good resource of reference when solving problems with parallel computers, yet with real machines, modifications to PRAM algorithms or new designs are necessary for high performance. We discussed the impact of limited number of processors, synchronization and multiple level of memory hierarchies on algorithm designs, and presented techniques that deal with these factors. As these factors we discussed will continue to be crucial to an algorithm's performance on multiprocessor systems, we expect our studies have significant impact on the experimental studies of parallel computing. We are also investigating running parallel graph algorithms on systems with transactional memory.

## References

1. Bader, D.A., Cong, G.: Efficient parallel algorithms for multi-core and multiprocessors. In: Rajasekaran, S., Reif, J. (eds.) Handbook of Parallel Computing: Models, Algorithms, and Applications, CRC press, Boca Raton, USA (2007)
2. Bader, D.A., Cong, G., Madduri, K.: Design of multithreaded algorithms for combinatorial problems. In: Rajasekaran, S., Reif, J. (eds.) Handbook of Parallel Computing: Models, Algorithms, and Applications, CRC press, Boca Raton, USA (2007)
3. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: IPDPS 2004. Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico (2004)

4. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: IPDPS 2004. Proc. 18th Int'l Parallel and Distributed Processing Symp., Santa Fe, New Mexico (2004)
5. Cong, G., Bader, D.: An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In: IPDPS 2005. Proceedings of the 19th International Parallel and Distributed Processing Symposium, Denver, Colorado (2005)
6. Greiner, J.: A comparison of data-parallel algorithms for connected components. In: SPAA-94. Proc. 6th Ann. Symp. Parallel Algorithms and Architectures, Cape May, NJ, pp. 16–25 (1994)
7. Hsu, T.S., Ramachandran, V., Dean, N.: Parallel implementation of algorithms for finding connected components in graphs. In: Bhatt, S.N. (ed.) Parallel Algorithms: 3rd DIMACS Implementation Challenge, October 17-19, 1994. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 30, pp. 23–41 (1997)
8. Krishnamurthy, A., Lumetta, S.S., Culler, D.E., Yelick, K.: Connected components on distributed memory machines. In: Bhatt, S.N. (ed.) Parallel Algorithms: 3rd DIMACS Implementation Challenge, October 17-19, 1994. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 30, pp. 1–21. American Mathematical Society, Providence, RI (1997)
9. Goddard, S., Kumar, S., Prins, J.: Connected components algorithms for mesh-connected parallel computers. In: Bhatt, S.N. (ed.) Parallel Algorithms: 3rd DIMACS Implementation Challenge, October 17-19, 1994. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 30, pp. 43–58 (1997)
10. Chung, S., Condon, A.: Parallel implementation of Borůvka's minimum spanning tree algorithm. In: IPPS'96. Proc. 10th Int'l Parallel Processing Symp., pp. 302–315 (1996)
11. Dehne, F., Götz, S.: Practical parallel algorithms for minimum spanning trees. In: Workshop on Advances in Parallel and Distributed Systems, West Lafayette, IN, pp. 366–371 (1998)
12. Woo, J., Sahni, S.: Load balancing on a hypercube. In: Proc. 5th Int'l Parallel Processing Symp., Anaheim, CA, pp. 525–530. IEEE Computer Society Press, Los Alamitos (1991)
13. Kruskal, C., Rudolph, L., Snir, M.: Efficient parallel algorithms for graph problems. Algorithmica 5, 43–64 (1990)
14. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: 4th Symp. Principles and Practice of Parallel Programming, ACM SIGPLAN, pp. 1–12. ACM Press, New York (1993)
15. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33, 103–111 (1990)
16. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company, New York (1992)
17. Shiloach, Y., Vishkin, U.: An $O(\log n)$ parallel connectivity algorithm. J. Algs. 3, 57–67 (1982)
18. Tarjan, R., Vishkin, U.: An efficient parallel biconnectivity algorithm. SIAM J. Computing 14, 862–874 (1985)
19. Arge, L., Bender, M., Demaine, E., Holland-Minkley, B., Munro, J.: Cache-oblivious priority queue and graph algorithm applications. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montreal, Canada, pp. 268–276. ACM Press, New York (2002)
20. Bender, M., Demaine, E., Farach-Colton, M.: Cache-oblivious search trees. In: FOCS-00. Proc. 41st Ann. IEEE Symp. Foundations of Computer Science, Redondo Beach, CA, pp. 399–409. IEEE Press, Los Alamitos (2000)

21. Helman, D.R., JáJá, J.: Designing practical efficient algorithms for symmetric multiprocessors. In: Goodrich, M.T., McGeoch, C.C. (eds.) ALENEX 1999. LNCS, vol. 1619, pp. 37–56. Springer, Heidelberg (1999)
22. Park, J., Penner, M., Prasanna, V.: Optimizing graph algorithms for improved cache performance. In: IPDPS 2002. Proc. Int'l Parallel and Distributed Processing Symp., Fort Lauderdale, FL (2002)
23. Chiang, Y.J., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., Vitter, J.: External-memory graph algorithms. In: Proceedings of the 1995 Symposium on Discrete Algorithms, pp. 139–149 (1995)