Practical Parallel Algorithms for Minimum Spanning Trees

Frank Dehne* Silvia Götz†

Abstract

We study parallel algorithms for computing the minimum spanning tree of a weighted undirected graph G with n vertices and m edges. We consider an input graph G with $m/n \geq p$, where p is the number of processors. For this case, we show that simple algorithms with data-independent communication patterns are efficient, both in theory and in practice. The algorithms are evaluated theoretically using Valiant's BSP model of parallel computation and empirically through implementation results.

1. Introduction

Computing a minimum spanning tree (MST) is one of the most studied problems in combinatorial optimization [19]. Formally, an MST of a given undirected connected Graph $G = (V, E)$ with vertices $V = \{0, \ldots, n-1\}$ and weighted edges E, $|E| = m$, can be defined as an acyclic subgraph of G which connects all vertices in V with the least total weight. The *density* k of a graph is the ratio between m and n. In this paper we consider input graphs with *sufficient* density, i.e., graphs that comply with $m/n \geq p$, where p is the number of processors in the parallel machine.

It has been pointed out (e.g., [28, 13]) that communication is the main bottleneck for the performance of parallel algorithms. Therefore, various parallel computation models, accounting for communication differently, have been proposed. Most of them support a coarse-grained approach, i.e., the memory/processor ratio is "large". Examples include the Bulk Synchronous Parallel (BSP) model [28], the Coarse-Grained Multicomputer (CGM) model [14], and the LogP model [13].

For the design and analysis of our parallel MST algorithms we employ Valiant's BSP model [28]. A BSP computer models a distributed memory machine and consists of a set of p processors, a router with computation/communication throughput g, and facilities for barrier synchronization. A synchronization can occur every L time units. A BSP algorithm consists of a sequence of supersteps which are separated by barrier synchronization. Messages sent in one superstep cannot be received until the next superstep. Each superstep i incurs a cost of $w_i + gh_i + L$, where w_i is the maximal number of local operations performed by any of the processors, and h_i is the maximal number of messages received or sent by any of the processors. The total cost of the algorithm is given by $W + gH + LT$, where $W = \sum_i w_i$, $H = \sum_i h_i$, and T is the number of supersteps.

In many parallel machines, the latency/synchronization cost L is considerably higher than the bandwidth parameter g, caused by the high startup cost for messages (e.g., Intel Paragon [21]). Therefore, special attention should be paid to the number of supersteps.

Several papers have been published describing graph algorithms for the BSP and similar models [7, 4, 15, 1], but very little effort has been put into experimental validation of these algorithms. Cáceres *et al.* note in [7] that graph problems "have considerably less 'internal structure' " than many other problems studied. This results in highly data-dependent communication patterns and makes it difficult to achieve communication efficiency.

In this paper, we study communication-efficient MST algorithms using the BSP model and show how they behave in practice. We show that graphs with sufficient density allow for a very simple and efficient MST computation on the BSP model, using $O(\log p)$ supersteps which is an important improvement over PRAM simulations requiring $\Omega(\log n)$ supersteps. One reason for this is that the MST algorithms for these graphs do not require data-dependent communication patterns. Our experiments also show that, although the BSP model is useful for designing parallel algorithms, it does not always rank algorithms correctly in terms of practical performance. Proofs and a more comprehensive discussion of these and other parallel MST algorithms can be found in [17].

Known Results. The MST problem has been extensively studied in the sequential setting (see [19] for a survey up

*School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6. Research partially supported by the Natural Sciences and Engineering Research Council of Canada. Email: dehne@scs.carleton.ca.

†Department of Mathematics and Computer Science, University of Paderborn, 33095 Paderborn, Germany. Research was supported by the Studienstiftung des Deutschen Volkes (German National Scholarship Foundation). Email: sylvie@uni-paderborn.de.

F. Dehne and S. Goetz, "Practical parallel algorithms for minimum spanning trees," in Proc. 17th IEEE Symp. on Reliable Distributed Systems, Advances in Parallel and Distributed Systems, West Lafayette, IN, USA, 1998, pp. 366-371.

to 1984, [22, 8]) with the best result being the linear time randomized algorithm in [22]. The MST problem has also a rich history in parallel computing and a number of PRAM algorithms have been proposed [9, 2, 12, 20, 10, 25]. The best of these results implies the existence of a linear work BSP algorithm using $O(\log n)$ supersteps given sufficient slackness.

Independent of this work, Adler *et al.* [1] also developed BSP algorithms for the MST problem requiring $O(\log p)$ supersteps, and proved a lower bound of $\min(n, m/p)$ on the per-processor communication volume for MST algorithms. They did not provide any implementation results for their algorithms.

There are only a few empirical investigations of sequential [16, 3, 23, 24], fine-grained parallel [3], and coarse-grained parallel [18, 11] MST algorithms. The work in [11] focuses on variants of list-ranking as applied to MST computations. It uses $\Omega(\log n)$ supersteps and its overall running time is not analyzed. The algorithm tested in [18] is only partially documented and its running time is not analyzed.

2. The Algorithms

For sake of exposition, we assume throughout this section that the input graph G is connected. Note that our solutions generalize easily to unconnected graphs. Without loss of generality we assume unique edge weights (make edge weights distinct by numbering the edges). $T_{\text{seq}}(n,m)$ denotes the time to compute sequentially an MST of a graph with n vertices and m edges. We assume a BSP computer with p processors, $p \geq 2$, and a *sufficiently* dense input graph with $m/n \geq p$. Initially, each of the p processors stores the values of n and m, and $\lceil m/p \rceil$ edges which are arbitrarily distributed among the processors. To make the algorithm's description easier, we assume that an input edge $e = \{u,v\}$ is stored as two copies (u,v) and (v,u) on the same processor, which is called the *home* processor of e. Our algorithms consist of two main procedures, *dense BSP Borůvka steps* and *merge steps*. Dense BSP Borůvka steps are a BSP implementation of Borůvka steps adapted to sufficiently dense graphs. Merge steps make use of the locality of data by computing the local MSTs on each processor and combine them along a D-ary tree. We will see that combining the two procedures results in efficient BSP algorithms.

2.1. Dense BSP Boruvka Step

Borůvka's algorithm [6] iterates in so-called *Borůvka steps*. At the beginning of Borůvka's algorithm, each vertex forms a supervertex. A Borůvka step proceeds in 3 steps. **B1.** Already found subtrees of the MST (which form *supervertices*) are enlarged by selecting the minimum-weight edge e incident to each subtree. These edges belong to the MST. **B2.** Selected edges $e = \{u,v\}$ are contracted, i.e., u and v are replaced by a new supervertex v'. The remaining edges incident to u and v are inherited by v'. **B3.** Self-loops are deleted and only the lightest among multiple edges remains.

In each Borůvka step the number of supervertices is reduced by a factor of at least 1/2. In the worst case, this results in $\Theta(\log n)$ iterations which implies $\Omega(\log n)$ supersteps when using at least a constant number of supersteps in each iteration.

We outline in the following our realization of a dense Borůvka step on the BSP model for sufficiently dense graphs ($m/n \geq p$).

BSP-B1. First, each processor computes the local minimum-weight edges (called *candidates*) incident to each supervertex. Then, for each supervertex the candidates are grouped together in segments on consecutive processors in a way that each processor does not hold more than $\lceil m/p \rceil$ elements (*segment scheme* [5]). The minimum of a segment is then computed along a d-ary tree, d to be determined by the analysis. The home processor combines the information for the two edge copies. The establishment of the segment scheme for sufficiently dense graphs can be performed data-independently by assuming wlog that each processor holds a candidate for each supervertex. To reduce the number of supersteps, we allot each processor $n/2^i$ elements in the ith iteration.

BSP-B2. The selected edges are broadcast to all processors, each of which computes sequentially the connected components labels 0 to n', n' being the number of the connected components. Edges can be updated locally. Contrary to the general case, this work does not have to be shared for efficiency reasons since $O(n) \in O(m/p)$.

BSP-B3. Selfloops can be detected completely locally. Without affecting the correctness of the algorithm, we discard the step of removing multiple edges.

Let p' be the number of participating processors, which we call *active*.

Lemma 1 *Given $p' \leq p$ active processors and a graph with $n' \leq n/2^i$ supervertices, $i \geq 0$, and m edges, with a maximum of M' local edges per processor, a dense BSP Borůvka step requires $O(M' + n/2^i + p'/2^{\log n - i})$ space per processor and $W + gH + LT$ time on the BSP model, with $W = O(M' + n/2^i + p'/2^{\log n - i})$, $H = O(n/2^i + p'/2^{\log n - i})$, and $T = O(1)$.*

2.2. Merging of Local MSTs

A simple approach for computing the MST of G is to merge local MSTs along a balanced D-ary tree, $2 \leq D \leq p$. First, each processor computes sequentially the MST of the graph induced by the locally stored edges. In a *merge step*

the edges of D local MSTs, say $F_{i(0)}, \ldots, F_{i(D-1)}$, are gathered by one processor which then computes the MST of the graph induced by $\bigcup_{j=0}^{D-1} F_{i(j)}$. After sending its local MST F_i to another processor, processor P_i will not be needed for further computations and becomes *passive*. The remaining processors stay *active*.

Lemma 2 *Let D be an integer with $2 \leq D \leq p$. Given a graph with n' vertices and m edges, with a maximum of M' local edges per processor, a D-ary merge step requires $O(D \cdot M')$ space per processor and $W + gH + LT$ time on the BSP model, with $W = T_{\text{seq}}(n', DM') + O(DM')$, $H \leq DM'$, and $T = 1$.*

2.3. MST Algorithms for Sufficiently Dense Graphs

A dense BSP Borůvka step reduces the number of supervertices by a factor of at least $1/2$, and a merge step discards a factor of $D - 1/D$ of the total number of edges. Thus, when executing both procedures interleaved, the problem size decreases geometrically, and so does the local computation time and the communication volume. Algorithm BORUVKA_MIXED_MERGE uses this approach.

BORUVKA_MIXED_MERGE

1. Every Processor computes sequentially the MST of the graph induced by the locally stored edges.
2. All processors are active. Do $\lceil \log_D p \rceil$ times (on set of active processors)
   a) Execute a dense BSP Borůvka step.
   b) Execute a D-ary merge step.

Theorem 1 *Let G be a connected undirected weighted graph with n vertices, m edges, and a density of $m/n \geq p$. Algorithm* BORUVKA_MIXED_MERGE *computes the MST of G on the BSP model with $O(m/p + Dn)$ space per processor in $W + gH + LT$ time, with $W = T_{\text{seq}}(n, m/p) + 2 \cdot T_{\text{seq}}(n, Dn) + O(Dn)$, $H = O(Dn)$, and $T = O(\log_D p)$.*

Note that for a constant D, BORUVKA_MIXED_MERGE requires an optimal communication volume of $O(n)$. The work $O(p \cdot T_{\text{seq}}(n, m/p))$ is optimal if $O(p \cdot T_{\text{seq}}(n, m/p)) = O(T_{\text{seq}}(n, m))$.

Other MST algorithms with worse BSP times consisting of dense BSP Borůvka steps and/or merge steps are briefly described in the following. Algorithm DENSE_BORUVKA consists only of dense BSP Borůvka steps and performs in $O((m/p + L)\log n + gn)$ BSP time. By merely using merge steps the MST can be computed in $O(T_{\text{seq}}(n, m/p) + T_{\text{seq}}(n, Dn)\log_D p + gDn\log_D p + L\log_D p)$ time (Algorithm MERGE_MST). A generalization of the idea in [1] is Algorithm BORUVKA_THEN_MERGE which first computes the local MSTs, then executes dense BSP Borůvka steps until $n/log_D^2 p$ supervertices are left, and concludes the computation by merging local MSTs. This results in $O(T_{\text{seq}}(n, m/p) + T_{\text{seq}}(n, Dn) + n\log\log_D p + g(nD/\log_D p + n) + L(\log_D p + \log\log_D p))$ time.

3. Experimental Results

We have implemented the algorithms described in Section 2.3 for sufficiently dense graphs and also investigated some variations. The algorithms compute the minimum spanning forest (MSF) for a possibly unconnected input graph. For computing the local MSF sequentially, the algorithms employ Kruskal's algorithm which works well on a wide variety of inputs [3]. Moreover, Kruskal's algorithm determines the MSF edges in order of increasing weight which enhances all but the first local MSF computation.

We investigated the algorithms' behavior on the Cognitive Computing parallel machine CC-48 by Parsytec Ltd., using 2 to 16 processors. The CC-48 [27] is a 48-node distributed memory computer based on Motorola PowerPC 604 processors which are interconnected by a fat mesh of clos of 2×24 processors. Our programs are written in the programming language C and make use of the Paderborn University BSP-library (PUB) [26].

Experiments were done on following types of input graphs. 1. RANDOM: For each of the m edges, a pair of vertices in $\{0, 1, \ldots, n-1\}^2$ is randomly chosen and assigned a random (integer) weight from $\{0, 1, \ldots, maxW\}$, where $maxW$ denotes the maximum allowable edge weight. The random numbers are generated by the C-function $rand()$. 2. PAIRS: This type of graph is constructed so that in each Borůvka step only pairs (except one triple for an odd number of supervertices) are collapsed. The remaining edges are random ones that are created in the same way as for graph type RANDOM. Their weight is chosen higher than the weights of the edges that cause the pair formation.

The behavior of the algorithms was tested on graphs of size of $n = 1000$, $m = 400000$, and $maxW = 1000000$. The runtimes for RANDOM express the average of 4 RANDOM graphs. For the same size one graph of type PAIRS was tested. All results state the average of at least 5 runs.

After the description of the experimental results (Sections 3.1 to 3.4) follows an assessment of the results in Section 3.5.

3.1. Dense BSP Borůvka Steps

Three versions of dense BSP Borůvka steps were implemented. The first version ("BSP plain") follows the description in Section 2.1 and sends even small messages. The second version ("message packing") groups small messages with the same destination processor into one large message. This saves the high startup cost for many small messages. Rather then using the segment scheme for computing

minimum-weight edges, our third version ("vector") employs, in addition to message packing, a vector computation (parallel prefix computation) provided by the PUB library. [1] The effects of the different dense BSP Borůvka steps are shown on Algorithm DENSE_BORUVKA. Figure 1 depicts the total running times of each version on input RANDOM.

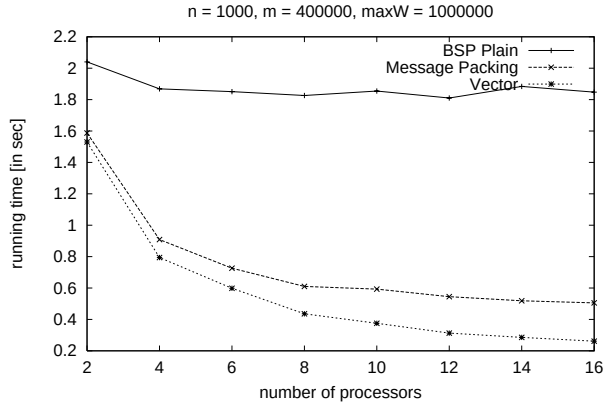

Figure 1. Different dense BSP Borůvka steps.

A significant improvement can be observed by using message packing. In fact, without using message packing, the running times do not decrease noticeably as the number of processors increases. For the RANDOM graphs the running times are reduced by 22.2% (2 processors) to 72.6% (16 processors) through the use of message packing. Hence, the impact of message packing is substantial. Even better results are obtained by the use of the vector function of the PUB library. The improvement over the message packing version is 3.6% for 2 processors to 48.2% for 16 processors. This enhancement shows the superior performance of a machine-dependently optimized algorithm over a generic algorithm. Similar results hold for the PAIRS graph.

3.2. Merge Steps

All algorithms containing merge steps were implemented with merge trees of various degrees. Although $D = 2$ results in the best asymptotical local computation time and communication volume on the BSP model, for most of our tests higher degrees were more beneficial. Higher degrees cause more load in the network, but result in less supersteps. More specifically, in MERGE_MSF using a degree other than 2 reduces the running time by at most 15.6% which was obtained with 14 processors. The value of D allows for optimizations based on machine-dependent parameters like latency and bandwidth. Its impact on the running time is considerable considering that the absolute runtime is small.

[1] Vector computation (instead of segment scheme) was suggested in the algorithm for sufficiently-dense graphs by Adler *et al.* [1].

3.3. Dense MSF Algorithms

The previous sections have given insight into the performance of the different executions of dense BSP Borůvka steps and merge steps. Now, we examine the performance of each algorithm in Section 2.3 on our inputs. The algorithms make use of message packing and the PUB library's vector function. We summarize only the results obtained using the best degree of the merge trees tested.

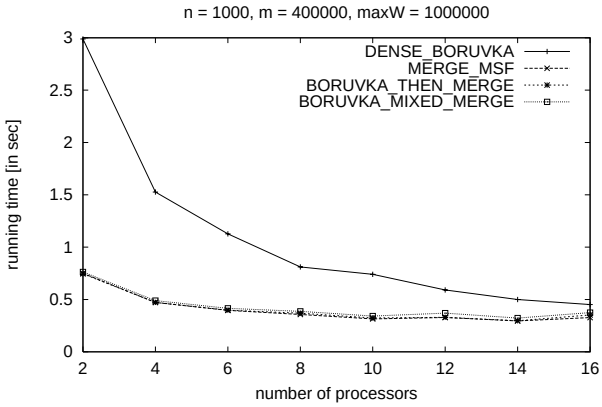

Figure 2. Runtimes on PAIRS

First, we analyze the performance of the algorithms on the input graph of type PAIRS which causes Algorithm DENSE_BORUKVA to have the maximum number of Borůvka steps. As we can see in Figure 2, Algorithm DENSE_BORUKVA performs worst for all numbers of processors tested, whereas almost the same running times were measured for the three algorithms containing merge steps. However, the gap between DENSE_BORUVKA and the other three algorithms becomes smaller with an increasing number of processors. DENSE_BORUVKA requires with 2 processors more than four times the running time used by the other algorithms. For 12 processors, it needs twice as much and with 16 processors additional 38% time.

Among the three algorithms that use merges steps, Algorithm BORUVKA_MIXED_MERGE requires most time, even though it has the best asymptotical BSP cost. The proportional deviation was largest with 16 processors for which its runtime was 14.6% more than the best. For all tested numbers of processors, except 2, Algorithm MERGE_MSF performs best. It improves the running time of BORUVKA_THEN_MERGE often only slightly. We observe that all algorithms which use merge steps have a larger runtime for 16 processors than for 14 processors.

For the average running times of the four RANDOM graphs, a similar behavior was observed, except for higher numbers of processors (shown in Figure 3). As expected, the absolute runtimes are halved for DENSE_BORUVKA.

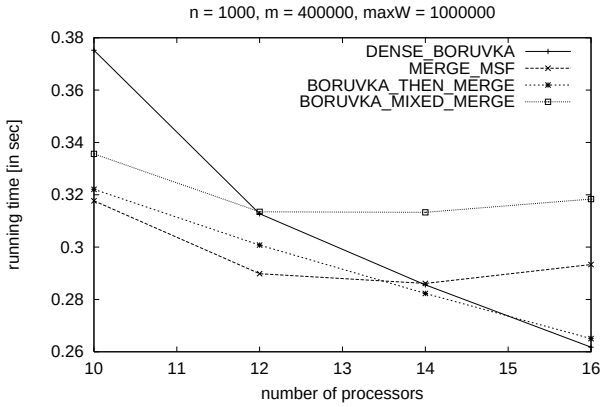

Figure 3. Runtimes on RANDOM (10 – 16 proc.)

The times for the other algorithms stayed almost in the same range. As before, the algorithms MERGE_MSF, BORUVKA_THEN_MERGE, and BORUVKA_MIXED_MERGE require similar runtimes. Algorithm DENSE_BORUVKA performs much better than for the PAIRS input. With 2 processors, it needs twice as much as the best of the others. This decreases to 18% for 10 processors. Starting from 12 processors, DENSE_BORUVKA is even better than Algorithm BORUKVA_MIXED_MERGE and becomes the best algorithm of all on the RANDOM input for 16 processors. Contrary to MERGE_MSF and BORUVKA_MIXED_MERGE, the runtime of Algorithm BORUVKA_THEN_MERGE decreases steadily achieving the best time of all algorithms with 14 processors. For 16 processors it is beaten by DENSE_BORUVKA with a runtime that is better by 1.2%.

3.4. Speedups

In order to investigate absolute speedups, compared to sequential implementations, we implemented the three standard sequential algorithms: Borůvka's, Kruskal's, and Prim's algorithms. For all inputs tested, Kruskal's algorithm performed best (1.603327 seconds for RANDOM, 1.494378 seconds for PAIRS) and Borůvka's algorithm required the most time.

For both input types we obtained a speedup of better than 2 for 2 processors, and of approximately 3.5 for 4 processors. We believe that if we increase the problem size we also obtain a higher, close to optimal, speedup for other small numbers of processors. The super-linear speedup for 2 processors is probably caused by caching effects. With the fixed-sized input of type PAIRS, the highest speedup (5.4%) was achieved by Algorithm MERGE_MSF for 14 processors. For RANDOM graphs, a speedup of 6.1% could be obtained using Algorithm DENSE_BORUVKA for employing 16 processors.

3.5. Assessment

Recall that the tests and therefore the following assessments are based on fixed-sized inputs. Algorithm MERGE_MSF performed very well for dense graphs and a small number of processors. This is because the algorithm is simple and hides only small constant factors in the big-Oh notation. Similarly good results were obtained by Algorithm BORUVKA_THEN_MERGE. For the RANDOM input BORUVKA_THEN_MERGE was even the only algorithm containing merge steps that had a decreasing runtime from 14 to 16 processors. BORUVKA_MIXED_MERGE performed worst among the algorithms that employ merge steps. BORUVKA_THEN_MERGE has the advantage over BORUKVA_MIXED_MERGE that the number of dense Borůvka steps performed is adjusted to the actual number of supervertices. The three algorithms containing merge steps have the advantage of being able to adapt to the conditions of the underlying parallel machine by varying the degree of the merge tree.

Algorithm DENSE_BORUVKA performs quite badly on small numbers of processors and for the PAIRS input in general. However, its runtime becomes better with an increasing number of processors. For the input of type RANDOM it requires only half the time for the worst case input, which makes it reasonable for more than 10 processors. It even outperforms all other algorithms on the RANDOM input for 16 processors.

For a fixed-size input, it seems that with an increasing number of processors the dense BSP Borůvka steps are more efficient than the merge steps, especially for random inputs. This is because efficiency of merging decreases as the number of processor increases (more local MSFs to merge, less edges can be discarded in one local MSF computation), while the efficiency of Borůvka steps stay the same. This is especially noticeable in RANDOM graphs, where the number of Borůvka steps is small. Algorithm DENSE_BORUVKA is able to adapt to the problem in the number of executed Borůvka steps.

Message packing is extremely important on the CC-48. Juurlink [21] has also noticed the advantage of message packing for the Intel Paragon and Chung and Condon observed it for the CM-5 [11].

4. Concluding Remarks

We conclude that the BSP model gives rise to efficient parallel MST algorithms. The presented algorithms use merging of local MSTs, therefore they could not have been designed under a model that supports only fine-grained parallel computing. Furthermore, the parallel implementations

have the advantage that they can cope with large input data. The sequential implementations could only handle inputs up to 500,000 edges. However, it seems that a large density is necessary to make the algorithms worthwhile. The minimum density of p predicted by the BSP model does not seem to be likely to produce good results for a small number of processors. Another shortcoming of the BSP model, at least for our experiments, is that it did not correctly predict the relative performance of the algorithms tested. However, this is more a problem of asymptotic analysis, since theoretically n, m, and p go to infinity, while in practice p is bounded by a small constant.

5. Acknowledgments

The use of the parallel machine CC-48 was possible through the Paderborn Center for Parallel Computing (PC^2). The authors would also like to thank Patrick Morin and Rolf Wanka for their helpful comments.

References

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutyłowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proc. of ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 27 – 36, 1998.

[2] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[3] R. S. Barr, R. V. Helgason, and J. L. Kennington. Minimal spanning trees: An empirical investigation of parallel algorithms. *Parallel Computing*, 12(1):45–52, 1989.

[4] A. Bäumker and W. Dittrich. Parallel algorithms for image processing: Practical algorithms with experiments. In *Proc. of Intern. Parallel Processing Symp. (IPPS)*, pages 429–433, 1996.

[5] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proc. of European Symp. on Algorithms (ESA)*, pages 17–30, 1995.

[6] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti v Brně (Acta Societ. Scient. Natur. Moravicae)*, 3:37–58 (in Czech.), 1926.

[7] 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. In *Proc. of Intern. Colloquium on Automata, Languages, and Programming (ICALP)*, pages 390 – 400, 1997.

[8] B. Chazelle. A faster deterministic algorithm for minimum spanning trees. In *Proc. of Symp. on Foundations of Computer Science (FOCS)*, pages 22 –31, 1997.

[9] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[10] K. W. Chong. Finding minimum spanning trees on the EREW PRAM. In *Proc. of Intern. Computer Symp. (ICS)*, pages 7 – 14, 1996.

[11] S. Chung and A. Condon. Parallel implementation of Boruvka's minimum spanning tree algorithm. In *Proc. of Intern. Parallel Processing Symp. (IPPS)*, pages 302–308, 1996.

[12] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proc. of SPAA*, pages 243–250, 1996.

[13] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. Santos, K. E. S. R. Subramonian, and T. von Eicken. A practical model of parallel computation. *Communications of the ACM*, 39(11):78 – 85, 1996.

[14] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. of ACM Symp. on Comput. Geometry*, pages 298–307, 1993.

[15] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. To appear in *Intern. Journal of Parallel Programming*.

[16] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[17] S. Götz. Communication-efficient parallel algorithms for minimum spanning tree computations. Diplomarbeit, Dep. of Math. and Comp. Science, University of Paderborn, Germany, 1998. http://www.scs.carleton.ca/~sylvie/work.html.

[18] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. of SPAA*, pages 1–12, 1996.

[19] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[20] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19:383 –410, 1995.

[21] B. Juurlink. Experimental validation of parallel computation models on the Intel Paragon. In *Proc. of IPPS/SPDP*, 1998.

[22] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[23] D. E. Knuth. *The Stanford GraphBase: A Platform for combinatorial Computing*. ACM Press, New York, NY, 1993.

[24] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *Computational Support for Discrete Mathem.*, volume 15 of *DIMACS Series in Discrete Mathem. and Theoretical Comp. Science*, pages 99–117. American Mathematical Society, 1994.

[25] C. Poon and V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *Proc. of Intern. Symp. on Algorithms and Computation (ISAAC)*, pages 212 – 222, 1997.

[26] Heinz Nixdorf Institut, University of Paderborn, AG Meyer auf der Heide. PUB-Library 5.0, Paderborn University BSP-library. http://www.uni-paderborn.de/~bsp.

[27] Paderborn Center for Parallel Computing (PC^2). CC-48 - a parallel PowerPC 604 system. http://www.uni-paderborn.de/pc2/services/systems/cc/.

[28] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.