DISTRIBUTED SORTING ALGORITHMS FOR MULTI-CHANNEL BROADCAST NETWORKS

Eli Gafni
John Marberg

February 1986
CSD-860055

Distributed Sorting Algorithms for Multi-Channel Broadcast Networks †

Eli Gafni
John M. Marberg

Computer Science Department
University of California
Los Angeles, CA 90024

Technical Report CSD-860055
February 1986
Revised November 1986

Abstract

A multi-channel broadcast network is a distributed computation model in which p independent processors communicate over a set of p shared broadcast channels. Computation proceeds in synchronous cycles, during each of which the processors first write and read the channels, then perform local computation. Performance is measured in terms of the number of cycles used in the computation. In this paper we investigate the problem of sorting p bit strings of uniform length m, where each string is initially located at a different processor in the broadcast network. The underlying assumption is that the transmission of each bit requires a separate cycle. We develop a sorting approach that first reduces the length of the strings without affecting their relative order, then proceeds using only the shorter strings. A sequence of three successively improved algorithms based on this approach are presented, the most efficient of which runs in $O(m+p\log p)$ cycles. By showing a lower bound of $\Omega(m)$ cycles, we prove that the algorithm is optimal for sufficiently large m. Our results can be used to improve by a factor of $\log p$ the solution of the multiple identification problem presented by Landau, Yung and Galil [Land85].

† Research supported in part by NSF grant DCR-84-51396 and by IBM grant D840622.

1. Introduction

Local area network architectures that use multiple broadcast channels have recently been proposed [Chou83, Marh85, Mars82a, Mars82b] as an alternative to single-channel Ethernet-like networks [Metc76]. In environments where messages are generated in real time, splitting a single channel into multiple channels of narrower bandwidth results in reduction of channel contention among processors at the expense of longer transmission time. It has been shown in [Mars83] that for high communication rates the reduced contention dominates the increased transmission time, and the overall message delay is decreased. Thus, multi-channel architectures seem to be viable, and it becomes of interest to investigate the complexity of algorithms in such architectures.

A *Multi-Channel Broadcast (MCB)* network [Marb85] is a general computation model for the design of distributed algorithms using broadcast communication. It consists of a collection of independent processors which communicate by means of multiple shared broadcast channels. Computation proceeds in synchronous *cycles*, during each of which the processors first write and read the channels, then perform local computation. The complexity measure is the number of cycles used by the computation.

In this paper we study the problem of sorting in the MCB model. The underlying assumption is that each bit to be transmitted requires a separate cycle. This is called *bit communication*. We have previously investigated the sorting problem under *uniform communication*, where each atomic unit of information (e.g., input element) can be transmitted in one cycle [Marb85].

Our sorting approach is tailored specifically for bit communication. Each of the elements to be sorted is encoded into a short representation called *signature*, such that the signatures have the same relative order as the original elements. The sorting then proceeds efficiently using the signatures.

Following is a summary of our results. Let p elements, each of length m bits, be distributed in a network consisting of p processors and p channels. We develop a sequence of three successively improved sorting algorithms, the most efficient of which has a complexity of $O(m+p\log p)$ cycles. By showing a lower bound of $\Omega(m)$ cycles, we prove that the algorithm is optimal for sufficiently large m.

Landau, Yung and Galil [Land85] investigate a model which is equivalent to ours. It consists of a fully connected synchronous point-to-point network, where in each cycle each node is capable of sending one bit over all outgoing links and receiving one incoming bit. The model is applied in solving the *multiple identification* problem, in which each of p processors contains a string of m bits, and needs to identify all the processors which have the same string as itself. The approach taken in [Land85] is to sort the strings, then use the sorted order to form groups of processors with identical strings. The sorting is implemented by emulation of the AKS network [Ajta83], which takes $O(m \log p)$ cycles. The total complexity of the solution is $O(m \log p + p)$ cycles. By replacing the AKS emulation with our sorting method, we are able to improve the upper bound on the multiple identification problem by a factor of $\log p$, which is optimal.

Other related research in the area of distributed algorithms with broadcast communication includes [Chan83, Dech86, Levi82]. Dechter and Kleinrock [Dech86] investigate a broadcast model called IPABM (Ideal Parallel Broadcast Model). This model differs from ours in two aspects. First, it provides only a single channel, and second, it allows concurrent-write access to the channel, whereas in our model exclusive-write access is used. The IPABM model is applied in the design of algorithms for extrema finding, merging and sorting. Levitan [Levi82] discusses a model called BPM (Broadcast Protocol Multiprocessor) which has essentially the same properties as the IPABM.

Chandra, Furst and Lipton [Chan83] analyze multi-party protocols for evaluation of binary predicates. The mode of communication is broadcasting in round-robin fashion on a single channel. Tight bounds are given for some specific protocols, however the results are mainly of theoretical value since they depend on Ramsey-like counting arguments.

The remainder of this paper is organized as follows. In Section 2 we define the MCB model. In Section 3 we discuss our sorting approach. Sections 4, 5, and 6 present the sorting algorithms. Section 7 shows the lower bounds.

2. The Multi-Channel Broadcast Network Model

The *Multi-Channel Broadcast (MCB)* network model consists of p independent processors which communicate over p shared broadcast channels. Each processor and each channel have a unique identifier known to all processors. We denote the processors as $P_1, P_2, \ldots, P_p$, and the channels as $C_1, C_2, \ldots, C_p$. Processor P_i has write-access only to channel C_i. Read-access, on the other hand, is unrestricted, i.e., any processor can read any channel. (In

[Marb85] we use a more general version of the model, in which there are p processors and k channels, $k \leq p$, and processors have both read- and write-access to all channels.)

Computation proceeds in synchronous *cycles*. We assume the existence of a global mechanism to synchronize the beginning of each cycle. A cycle consists of the following two phases at each processor.

1. COMMUNICATION: Write your channel and read one other channel.
2. PROCESSING: Perform local computation.

The information written on a channel during a given cycle is received only by the processors reading the channel in that cycle. The complexity measure is the number of cycles used in the computation.

The capability of a processor to write and read two different channels simultaneously in the same cycle is assumed for convenience in algorithm design. It can be shown that limiting each processor to access a single channel per cycle does not decrease the power of the model.

An algorithm for the MCB network is said to be *oblivious* [Cook86] if the processors that read each given channel in each given cycle are known in advance, independent of the particular instance of the input. In other words, a processor can determine which channel to read in any given cycle simply as a function of the number of cycles that have elapsed form the beginning of the algorithm (and perhaps the general parameters of the problem). Two of the three sorting algorithms presented in this paper are oblivious.

3. The Signature Approach

The sorting problem we are studying is the following: p elements, each a string of length m bits, are distributed in an MCB consisting of p processors and p channels; the task is to rearrange the elements in the network so that the element at processor P_i will be greater or equal to the element at P_{i+1}.

The MCB network can easily emulate a sorting network for p elements, such as AKS [Ajta83] or the bitonic network [Batc68]. However, under the assumption of bit communication this becomes inefficient when the elements are long, since $\Theta(m)$ cycles are needed to emulate

each stage of the sorting network. AKS emulation, for example, requires a total of $\Theta(m\log p)$ cycles.

Our goal is to develop a sorting method which does not entail repeated transmission of long elements. To accomplish this, we separate the task of computing the position of each element in the sorted order from the actual rearrangement of the elements in the network.

The idea is to compute for each element a short encoding called *signature*, such that the signatures have the same relative order as the original elements. The sorted order is then found efficiently using the signatures.

We present a sequence of three algorithms, A, B and C, based on this approach. Each algorithm improves upon the previous one by using a more efficient technique to sort the signatures. Algorithm A uses bottom-up processing on a tree, running in a total of $O(m+p\log^2 p)$ cycles. Algorithm B combines a tree with a bitonic network, and runs in $O(m+p\log p\ \text{loglog}\, p)$ cycles. Algorithm C employs divide and conquer, and has a complexity of $O(m+p\log p)$ cycles.

Before we describe the algorithms, we need the following definitions. $[a_1, a_2, \ldots, a_l]$ denotes a list of l elements. Given two lists of equal length, $X=[x_1, x_2, \ldots, x_l]$ and $Y=[y_1, y_2, \ldots, y_l]$, we use (X, Y) to denote the list of pairs $[(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)]$.

Let $Z=[z_1, z_2, \ldots, z_l]$ be a list of l not necessarily distinct elements from a totally ordered domain, and let $z_{i_1} \geq z_{i_2} \geq \cdots \geq z_{i_l}$ be a nonincreasing order among the elements. The list $[i_1, i_2, \ldots, i_l]$ is called the *sorting permutation* of Z.

The *rank* of z_i in Z, denoted $R(z_i, Z)$, is defined as the number elements in Z that are strictly larger than z_i. We use $R(Z)$ to denote the list $[R(z_1, Z), R(z_2, Z), \ldots, R(z_l, Z)]$, called the *rank list* of Z. Computing $R(Z)$ is called *ranking*.

In the next three sections we present the sorting algorithms. Following our approach, each algorithm consists of three phases: (1) signature computation; (2) signature sorting; and (3) element transfer.

4. Algorithm A

4.1. Signature Computation

Without loss of generality assume that p divides m. Let e_i denote the element initially at processor P_i. Let $e_{i,j}$, $1 \le j \le p$, be a substring of e_i of length $\frac{m}{p}$, starting at position $(j-1)\frac{m}{p}+1$. We call $e_{i,j}$ the j'th *component* of e_i. We can view the input as a square matrix of components $\{e_{i,j} \mid 1 \le i,j \le p\}$, with the i'th row located at processor P_i.

The signatures are obtained in the following manner. Let B_j denote the j'th column of the component matrix. We compute for each component $e_{i,j}$ the rank $r_{i,j}=R(e_{i,j}, B_j)$. All ranks are in the range 0 to $p-1$, so we view them as strings of $\lceil \log p \rceil$ bits.[1] The signature of element e_i is the concatenation of the ranks of its components, i.e., the string $r_{i,1} r_{i,2} \cdots r_{ip}$. Thus, each signature consists of $p \lceil \log p \rceil$ bits.

It can be verified that the signatures have the same relative order as the original elements. This follows from the fact that ranking preserves order. Notice that when $m < p \log p$, the signature is actually longer than the element itself. This has no bearing on the performance of the algorithm.

To implement phase 1, we first transpose the component matrix, thereby moving each column B_j to processor P_j. Then, the ranks in each column are computed locally, yielding a transposed matrix of ranks. As will be seen in the description of phase 2, there is no need to "re-transpose" the ranks and explicitly form the signatures.

The transpose operation is implemented using the following oblivious communication protocol. Row i of the component matrix is initially at processor P_i. There are $p-1$ steps, during each of which one component from each row is moved to the corresponding column. To achieve maximum concurrency, different positions in different rows are used in each step. Formally, in step t, $0 \le t \le p-2$, processor P_i sends component $e_{i,\,(i+t) \bmod p+1}$ to processor $P_{(i+t) \bmod p+1}$ over channel C_i. Each step takes $\frac{m}{p}$ cycles, for a total of $\frac{m}{p}(p-1)=O(m)$ cycles.

[1] Throughout the paper we use "log" to denote logarithm of base 2.

4.2. Signature Sorting

The second phase computes the sorting permutation using the signatures. This could be done in $O(p\log^2 p)$ cycles by straightforward emulation of the AKS sorting network. However, due to the large constants involved, the AKS approach is impractical. Instead, we use the following method, which achieves the same complexity but is considerably more practical.

We begin by ranking the signatures. Let Z denote the corresponding rank list. Since ranking preserves order, the ranks in Z have the same relative order as the original elements. We thus obtain the sorting permutation from Z.

We now describe how to compute Z. Let s_i denote the signature of element e_i. Breaking each signature into two parts of equal length, let s_i^+ denote the left part (the most significant bits) and let s_i^- denote the right part (the least significant bits). Let S, S^+, and S^- denote the lists comprising all the s_i, s_i^+, and s_i^-, respectively.

The reader may verify the correctness of the formula $Z = R(S) = R((R(S^+), R(S^-)))$, where the order among rank pairs is determined lexicographically. This suggests the following bottom-up tree computation of Z. We divide each signature into p equal substrings, or components. Now, consider a full binary tree with p leaves (i.e., all the leaves are in at most two adjacent levels), where the j'th leftmost leaf contains a list comprising the j'th component of every signature. Moving bottom-up in the tree, we combine at each parent the lists of its two children, as follows. Let G^+ and G^- denote the lists of the left and right child, respectively. The parent is assigned the list $R((G^+, G^-))$. It is easy to see that the root contains the rank list Z.

The j'th leaf list is actually the j'th column in the rank matrix of phase 1. This column is located in P_j. We implement each parent in the same processor that implements its left child. The root is thus in processor P_1. To evaluate a parent node, G^- is sent from the processor implementing the right child to the processor implementing the parent. The latter then computes $R((G^+, G^-))$ locally. All nodes in the same level in the tree can be processed in parallel. There are $\lfloor \log p \rfloor$ levels, and each level takes $p\lceil \log p \rceil$ cycles to transmit the lists G^-. The total cost of the tree is therefore $O(p\log^2 p)$ cycles.

It remains to obtain the sorting permutation from Z. Since Z is in P_1, this can be done locally. Finally, P_1 broadcasts the permutation to all processors. The total cost of phase 2 is $O(p\log^2 p)$ cycles.

4.3. Element Transfer

In the third and final phase of the algorithm, the elements are transferred to their destination processors according to the sorting permutation. Let P_{d_i} be the destination of element e_i. Since the permutation has been broadcast, P_{d_i} knows the index i. e_i can therefore be sent directly from P_i to P_{d_i} over channel C_i. All the processors proceed in parallel. The total cost of the transfer is $O(m)$ cycles.

The reader may observe that the transfer protocol just described is not oblivious, since the id of the channel to be read by each processor depends on the sorting permutation. On the other hand, phases 1 and 2 are oblivious. We now give an oblivious transfer protocol which is as efficient as the non-oblivious one.

The protocol consists of three steps. First, the elements are divided into p equal components and transposed, similar to phase 1. In fact, if storage availability permits the processors to save the component lists B_j in phase 1, this step is redundant. Second, each P_j locally rearranges B_j according to the sorting permutation. That is, if the destination of element e_i is processor P_{d_i}, then $e_{i,j}$ is placed in position d_i in the list. Finally, the rearranged components are transposed a second time. It can be verified that this effectively accomplishes the transfer. The cost is $O(m)$ cycles.

4.4. Complexity

Summing up the costs of all three phases, the complexity of algorithm A is $O(m+p\log^2 p)$ cycles. If m is sufficiently large, it dominates the complexity. In Section 7 we show that in this case the algorithm is optimal.

Notice that the algorithm beats AKS emulation when $m>p\log p$. This illustrates the difference between bit communication and uniform communication.

5. Algorithm B

In phase 2 of algorithm A, as the computation gets closer to the root of the tree, more and more processors become idle. The idea of algorithm B is to modify the tree computation in order to increase processor utilization, thereby improving the performance. We now describe how this is accomplished.

The ranks in each level of the tree can be viewed as components of new signatures whose length is half the length of the signatures in the previous level. Based on this observation, we make the following change. Instead of evaluating the entire tree, we stop at a level where the new signatures are sufficiently short, then switch to emulation of the bitonic sorting network [Batc68] on these signatures. Appending the signature of element e_i with the index i, the effect of the sorting is that each processor P_j knows the j'th index in the sorting permutation. To make the entire permutation public, one processor after another broadcasts the index. Notice that prior to the bitonic sort it is necessary to transpose the current ranks, so that each processor will contain its signature.

Let the tree computation be discontinued after r levels. The length of the signatures is then $O(\frac{p\log p}{2^r})$ bits. There are $O(\log^2 p)$ phases in the bitonic network, so the emulation takes $O(\frac{p\log p}{2^r}\log^2 p)$ cycles. The cost of r tree levels is $O(rp\log p)$ cycles. By choosing $r=2\text{loglog}\,p$, the total cost of phase 2 becomes $O(p\log p\ \text{loglog}\,p)$ cycles.

Phases 1 and 3 are the same as in algorithm A. The total complexity of algorithm B is therefore $O(m+p\log p\ \text{loglog}\,p)$ cycles. Notice that in contrast to the AKS network, the simple structure of the bitonic network results in a very practical algorithm.

6. Algorithm C

The main idea in phase 2 of algorithms A and B is to iteratively reduce the length of the signatures by half. Yet, using the tree mechanism, each reduction step takes the same number of cycles, $p\lceil\log p\rceil$, regardless of the current length of the signatures. If each reduction could be done at a cost which is linear in the current length of signatures, then the complexity of phase 2 would improve to $O(p\log p)$ cycles. This is basically what is achieved in algorithm C.

Phases 1 and 3 of algorithm C are the same as in the previous algorithms, and will not be discussed. In phase 2, instead of a tree, we use a divide and conquer approach resembling radix-exchange sort [Knut73]. The idea is the following. Initially, all p processors comprise one group. We divide the processors into several subgroups, each comprising at most $\lceil \frac{p}{2} \rceil$ processors. The division is such that the input elements in each subgroup occupy successive positions in the sorted order, starting at a given (known) position. From now on, it remains only to determine the order within each subgroup. Moreover, all subgroups can proceed in parallel, independent of each other. Each subgroup computes a new set of signatures, using the same method as in phase 1 but starting from the current signatures. It can be seen that the length of the new signatures is at most half the previous length. The division is then repeated recursively in each subgroup until all subgroups become singletons, at which point each processor knows the position of its input element in the sorted order. To obtain the sorting permutation, one processor after another broadcasts its position.

We now show how to implement each recursive level in linear number of cycles in the current length of the signatures. Consider a group R in a given level of the recursion, consisting of $r = |R|$ processors $P_{i_1}, P_{i_2}, \ldots, P_{i_r}$. The length of the signatures in R is $r\lceil \log p \rceil$ bits. Let the signatures be organized in a transposed matrix of $r \times r$ components (see algorithm A).

Processor P_{i_1}, which contains the first (most significant) column of signature components, divides the processors into subgroups, such that each subgroup comprises all and only those processors which have the same first component. Clearly, the elements in each subgroup belong in successive positions in the sorted order. Moreover, since each signature component is actually a rank, the component value corresponding to a given subgroup is a "pointer" to the position in the sorted order (of the elements of R) where the largest element of the subgroup belongs.

Obviously, there exists at most one subgroup with more than $\lceil \frac{r}{2} \rceil$ processors. Let us call this the "bad" subgroup. P_{i_1} informs group R about the division by broadcasting the processor ids and the corresponding pointer of every subgroup, except the bad subgroup. The processors and pointer of the latter can then be determined by elimination.

It now remains to further divide the bad subgroup. This is done by processor P_{i_2}, using the second most-significant signature component, in a similar way as before. The scheme continues component after component until either the size of the bad group is reduced below $\lceil \frac{r}{2} \rceil$, or all components are exhausted. In the latter case, the bad subgroup can be excluded altogether from the remainder of the recursion since all its input elements are identical.

It can be seen that each processor id in group R is broadcast at most once during the above protocol. Thus, the cost of dividing R is $O(r \log p)$ cycles. The new signatures in each subgroup are then computed form the current signatures using the method of phase 1, which also costs $O(r \log p)$ cycles.

Since the size of the groups is reduced by at least half in each recursive level, the recursion terminates after $\lceil \log p \rceil$ levels. All the groups in each level proceed in parallel, so the total cost of phase 2 is $O(\sum_{l=0}^{\log p} (\frac{p}{2^l} \log p)) = O(p \log p)$ cycles. Notice that it is necessary to set a global synchronization point at the end of phase 2, since groups of different sizes proceed through the recursion at a different pace.

The total complexity of algorithm C is $O(m+p \log p)$ cycles. It can be seen that the algorithm is not oblivious, in contrast to algorithms A and B. This is because the division into groups is dependent on the input. It is interesting whether the upper bound established by algorithm B can be improved by means of an oblivious algorithm.

7. Lower Bounds

We now show a lower bound on the complexity of sorting under the assumption of bit communication. Clearly, if elements of length m are to be rearranged in the network, $\Omega(m)$ cycles is a lower bound. Yet, we can prove a stronger result. We can show that the $\Omega(m)$ bound holds even if all we require is that destination processors obtain "pointers" to the elements in the sorted order, without actually transferring the elements. For example, processor P_j could use as pointer the j'th index in the sorting permutation.

Theorem 1. Sorting strings of length m requires $\Omega(m)$ cycles.

Proof. Consider the following problem. Let e_1 be a bit string known only to P_1, and e_2 a bit

string known only to P_2. Given that $e_1 \neq e_2$, we want to determine whether or not $e_1 > e_2$.

Using Yao's lower bound on two-party protocols [Yao79] it can be shown that a solution requires $\Omega(m)$ cycles. The bound holds even if more than two processors are involved in the computation. This is because the contribution of processors other than P_1 and P_2 is warranted only by the information they obtain from P_1 and P_2, and such information may as well be communicated directly between the two processors.

On the other hand, the problem can be solved by sorting, using dummy elements $e_i = 0$ in all processors except P_1 and P_2. To this end, $e_1 > e_2$ if and only if the first index in the sorting permutation is 1. It follows that sorting also requires $\Omega(m)$ cycles. ■

Corollary 1. When m is sufficiently large, algorithms A, B, and C are optimal. Specifically, algorithm C is optimal for $m \geq p \log p$. ■

Another implication of Theorem 1 is that the divide and conquer method used in algorithm C is optimal, in the sense that signatures of length $p \log p$ cannot be sorted in less than $\Omega(p \log p)$ cycles. Thus, algorithm C is the best possible implementation of the signature approach.

We now show a lower bound which holds regardless of the type of communication being used (bit or uniform).

Theorem 2. Sorting requires at least $\log p$ cycles.

Proof. Let us number the cycles of the computation sequentially. The set of input elements that *affect* processor P_i in cycle t, denoted $A_i(t)$, is defined recursively as follows.

1. $A_i(0) = e_i$.
2. $A_i(t+1) = A_i(t) \cup A_j(t)$, where C_j is the channel being read by P_i in cycle $t+1$.

Let t^* denote the last cycle of the computation. Clearly, each processor must eventually be affected by all the input elements. Thus, $A_i(t^*) = p$. Yet, it can be seen from the recursive formulation that $|A_i(t^*)| \leq 2^{t^*}$. It follows that $t^* \geq \log p$. ■

Theorem 2 implies that the AKS emulation, which takes $O(m\log p)$ cycles, is optimal for $m=O(1)$.

8. Conclusion

We have presented efficient sorting algorithms for the MCB network, thereby demonstrating the power and applicability of the model. The protocols we have developed, in particular the transpose operation and the tree computation, prove to be useful design tools.

We have shown that algorithm C is optimal for $m\geq p\log p$. On the other hand, when $m\leq p$, AKS emulation achieves a better performance of $O(m\log p)$ cycles. An open problem is to bridge the gap between the upper bound $O(p\log p)$ and the lower bound $\Omega(m)$ in the range $p<m<p\log p$. It also remains open whether $O(m\log p)$ is optimal for $m\leq p$.

Acknowledgement

We are indebted to David Cantor for suggesting the idea which led to algorithm C.

References

[Ajta83] Ajtai, M., J. Komlos, and E. Szemeredi, "An $O(N\log N)$ Sorting Network," in *Proceedings 15th ACM Symp. on Theory of Computing*, 1983, pp. 1-9.

[Batc68] Batcher, K.E., "Sorting Networks and their Applications," in *Proceedings AFIPS Spring Joint Computer Conf.*, Vol. **32**, April 1968, pp. 307-314.

[Chan83] Chandra, A.K., M.L. Furst, and R.J. Lipton, "Multi-Party Protocols," in *Proceedings 15th ACM Symp. on Theory of Computing*, 1983, pp. 94-99.

[Chou83] Choudhury, G.L. and S.S. Rappaport, "Diversity ALOHA - A Random Access Scheme for Satellite Communications," *IEEE Trans. Communications* **COM-31**, 3 (March 1983), pp. 450-457.

[Cook86] Cook, S., C. Dwork, and R. Reischuk, "Upper and Lower Bounds for Parallel Random Access Machines Without Simultaneous Writes," *SIAM J. Comput.* **15**, 1 (Feb. 1986), pp. 87-97.

[Dech86] Dechter, R. and L. Kleinrock, "Broadcast Communications and Distributed Algorithms," *IEEE Trans. Computers* **C-36**, 3 (March 1986), pp. 210-219.

[Knut73] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[Land85] Landau, G.M., M.M. Yung, and Z. Galil, "Distributed Algorithms in Synchronous Broadcasting Networks," in *Proceedings 12th Int. Conf. on Automata, Languages and Programming*, 1985, pp. 363-372. To appear in *TCS*.

[Levi82] Levitan, S.P., "Algorithms for a Broadcast Protocol Multiprocessor," in *Proceedings 3rd Int. Conf. on Distributed Computing Systems*, 1982, pp. 666-671.

[Marb85] Marberg, J.M. and E. Gafni, "Sorting and Selection in Multi-Channel Broadcast Networks," in *Proceedings 1985 Int. Conf. on Parallel Processing*, pp. 846-850.

[Marh85] Marhic, M.E., Y. Birk, and F.A. Tobagi, "Selective Broadcast Interconnection: a Novel Scheme for Fiber-Optic Local-Area Networks," *Optics Letters* **10**, 12 (Dec. 1985), pp. 629-631.

[Mars82a] Marsan, M.A., "Multichannel Local Area Networks," in *Proceedings IEEE Fall COMPCON*, 1982, pp. 493-502.

[Mars82b] Marsan, M.A., D. Roffinella, and A. Murru, "ALOHA and CSMA Protocols for Multichannel Broadcast Networks," in *Proceedings Canadian Communications and Energy Conf.*, 1982, pp. 375-378.

[Mars83] Marsan, M.A., P. Camarda, and D. Roffinella, "Throughput and Delay Characteristics of Multichannel CSMA-CD Protocols," in *Proceedings IEEE GLOBECOM*, 1983, pp. 1147-1151.

[Metc76] Metcalfe, R.M. and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM* **19**, 7 (July 1976), pp. 395-403.

[Yao79] Yao, A.C., "Some Complexity Questions Related to Distributive Computing," in *Proceedings 11th ACM Symp. on Theory of Computing*, 1979, pp. 209-213.