SORTING AND SELECTION IN MULTI-CHANNEL BROADCAST NETWORKS

John M. Marberg
Eli Gafni

January 1985
CSD-850002

Sorting and Selection in Multi-Channel Broadcast Networks†

John M. Marberg
Eli Gafni

Computer Science Department
University of California
Los Angeles, CA 90024

Abstract

A multi-channel broadcast network is a distributed computation model in which processors communicate by sending broadcast messages over a set of shared broadcast channels. Computation proceeds in synchronous cycles, during each of which the processors first write and read the channels, then perform some local computation. We discuss the complexity of sorting and selection by rank in sets distributed among the processors of such a network. Given a set of n elements and a network with p processors and k channels, we show a tight bound of size $\Theta(n)$ on the number of messages and $\Theta(\max\{\frac{n}{k}, n_{\max}\})$ on the number of cycles required for sorting, where $n_{\max}$ is the maximum number of elements in any processor. Similarly, we show a tight bound of $\Theta(p\log\frac{kn}{p})$ messages and $\Theta(\frac{p}{k}\log\frac{kn}{p})$ cycles for selection. We first discuss lower bounds, then describe efficient algorithms which achieve such bounds in a wide range of cases.

1. Introduction

Local area network architectures which use multiple broadcast channels have recently been proposed [Chou83, Mars82a, Mars82b] as an alternative to single-channel Ethernet-like networks [Metc76]. In environments where messages are generated in real time, multiple channels reduce the 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 various algorithms in such architectures.

† This research was supported by an IBM Faculty Development Award.

Algorithms which use broadcasting as the basic mode of communication have received little attention in the literature. In this paper we describe a general computation model for the design of such algorithms. The model consists of a collection of independent processors connected by multiple shared broadcast channels. We call this model a *Multi-Channel Broadcast (MCB)* network.

Complexity in the MCB model is measured in terms of the number of *cycles* required by the algorithm, where each cycle at each processor consists of reading and writing a pair of channels, then performing some arbitrary local computation. We assume the existence of a global mechanism that synchronizes the beginning of each cycle. The issue of collision resolution is avoided in the design of algorithms in the MCB model by requiring that such algorithms be collision-free, i.e., that no two processors attempt to write on the same channel in the same cycle.

The main contribution of this paper is efficient broadcast algorithms for sorting and selection by rank in distributed sets. First, we give lower bounds on the number of cycles and the number of messages required to perform sorting and selection in the MCB model. We then develop efficient algorithms which achieve the lower bounds in a wide range of cases, thus proving that the bounds are tight. The following is a summary of our results. Assume an MCB with p processors and k channels, $k \leq p$, and a set of n elements distributed among the processors. The complexity of sorting is $\Theta(n)$ messages and $\Theta(\max\{\frac{n}{k}, n_{\max}\})$ cycles, where $n_{\max}$ is the maximum number of elements in any processor. The complexity of selecting the element of rank d, where $\frac{\epsilon n}{2} \leq d \leq \lceil \frac{n}{2} \rceil$ for some constant $0 < \epsilon \leq 1$, is $\Theta(p \log \frac{kn}{p})$ messages and $\Theta(\frac{p}{k} \log \frac{kn}{p})$ cycles. The tight bounds on messages and cycles are achieved simultaneously.

Earlier research in the area of broadcast algorithms includes the work of Dechter and Kleinrock [Dech81, Dech84] on the IPBAM model. This model differs from our model in two aspects. First, it only provides a single channel, and second, it allows concurrent write access to the channel and assumes the existence of a global collision resolution mechanism. The IPBAM model was used in the design of algorithms for extrema finding, merging and sorting. In our model, these problems are solved without the need for concurrent write access. Levitan [Levi82] discusses a model called BPM, which has identical properties to the IPBAM.

Santoro and Sidney [Sant82, Sant83] discuss a broadcast model called Shout-Echo, in which a basic communication activity consists of one processor broadcasting a message (shout) and receiving a reply (echo) from all other processors. In the MCB model, on the other hand, each message transmission is a separate communication activity, involving the sending processor and any number of receiving processors. Moreover, multiple disjoint communication

activities may proceed simultaneously on separate channels. The Shout-Echo model was used in the design of algorithms for selection.

The remainder of this paper is organized as follows. In Section 2 we define the computation model. Section 3 explains our notation. The lower bounds are shown in Section 4. Sections 5-7 present sorting algorithms, and Section 8 the selection algorithm. Concluding remarks are given in Section 9.

2. The Multi-Channel Broadcast Model

The *multi-channel broadcast (MCB)* network model consists of a collection of independent processors which communicate by sending broadcast messages over a set of shared broadcast channels. A given configuration of the network with p processors and k channels is denoted MCB(p, k). It is always assumed that $p \geq k$, whereas in practical cases usually $p >> k$. Each processor has a unique identifier, taken from a totally ordered set of p identifiers known to all processors. Similarly, each channel has a unique identifier known to all processors.

The computation proceeds in synchronous cycles. We assume the existence of a global mechanism for cycle synchronization. During each cycle, a processor may access two channels - one channel for the purpose of writing and the other for reading. A cycle consists of the following three steps at each processor.

1. Write one channel.
2. Read one channel.
3. Perform arbitrary local computation.

The width of a slot on the channels is one cycle. Thus, a message written on a channel in a given cycle is received only by the processors reading the channel in that cycle. Processors reading a channel can detect that the channel is empty, i.e., that no processor has written on the channel during that cycle. To avoid the issue of collision resolution in the design of our algorithms, we require that the algorithms be collision-free. In other words, in terms of the computation model, concurrent access to the same channel by more than one processor is allowed only for reading. If more than one processor attempts to write on the same channel in the same cycle, the computation fails.

The channel access rules of the MCB model resemble those of the Concurrent-Read Exclusive-Write (CREW) model [Snir83]. However, in the CREW model, communication is achieved by means of shared memory, and the input is usually in the shared memory as well. Broadcast channels, in contrast, are "memoryless", thus both the input and the computation are fully distributed. Also, in the MCB model the number of channels does not exceed the

number of processors, whereas in the CREW model an arbitrary size shared memory may be assumed.

Computational complexity in the MCB model is measured in terms of the total number of cycles and the total number of broadcast messages required by the computation. A message consists of at most $O(\log\beta)$ bits, where β is the value of the largest parameter or datum involved in the computation.

Let $p' \geq p$ and $k' \geq k$. It is easy to show that one cycle of an MCB(p', k') can be simulated in $O((\frac{p'}{p})^2 \frac{k'}{k})$ cycles on an MCB(p, k), using only $O(\frac{p'}{p})$ messages per each original message. The idea is to simulate $\lceil \frac{p'}{p} \rceil$ processors on each processor and $\lceil \frac{k'}{k} \rceil$ channels on each channel, and repeat each message $\lceil \frac{p'}{p} \rceil$ times. This can be used to make various assumptions on p and k without loss of generality (w.l.g.). For example, we may assume that p is a power of 2, that k divides p, etc. (if not, an MCB with such properties can be simulated on the existing MCB at no added complexity).

3. Notation and Definitions

The processors of an MCB(p, k) are denoted $P_1, P_2, \ldots, P_p$. The channels are denoted $C_1, C_2, \ldots, C_k$. W.l.g. we assume k divides p.

Let N be a collection of elements distributed arbitrarily among the processors. W.l.g. we may assume that N is a set, i.e., that all elements in N are distinct. If not, we can replace each element ξ in P_i with the triple (ξ, i, j_ξ) where j_ξ is a unique index within P_i, and use lexicographic order among the triples. The subset of elements at processor P_i is denoted N_i. The sizes of the sets are $|N|=n$ and $|N_i|=n_i$. We assume $n \geq p$ and $n_i > 0$, whereas in practical cases usually $n >> p$.

The term n_i^+ denotes the partial sum $n_1+n_2+\cdots+n_i$. For convenience, we also define $n_0^+=0$. $n_{\max}$ and $n_{\max 2}$ denote the largest and second largest, respectively, among the n_i's. If $n_i=n_j$ for all $i \neq j$, we say that the distribution of N among the processors is *even*. Otherwise, the distribution is *uneven*.

We denote by $N[j]$ the j'th largest element in N. $N[j_1, j_2]$ where $j_1 \leq j_2$ denotes the set $\{N[j_1], N[j_1+1], \ldots, N[j_2]\}$. Alternatively, $N[j_1, j_2]$ can be interpreted as a segment of the sorted list N from rank j_1 to j_2. $N[\lceil \frac{n}{2} \rceil]$ is called the median of N. $N_i[j]$, $N_i[j_1, j_2]$, and the median of N_i are defined similarly. We will use the terms list and set interchangeably in reference to N and N_i.

Selection is defined as identifying $N[d]$, the d'th largest element in N, for a given rank d. W.l.g. we may assume $1 \leq d \leq \lceil \frac{n}{2} \rceil$ (if not, reverse the sorting order and select the element of rank $n-d+1$).

Sorting is defined as rearranging the distribution of N among the processors so that $N_i = N[n_{i-1}^+ + 1, n_i^+]$. In other words, after the sorting the number of elements in each processor remains the same as before, however the elements in P_i are now larger than the elements in P_{i+1}.

Throughout the paper we use log to denote logarithm of base 2.

4. Lower Bounds

4.1. Lower Bounds on Selection

The following theorem is a generalization of a result by Frederickson [Fred83].

Theorem 1. The number of messages required to select the median of n elements distributed among p processors is $\Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{max})$.

Proof. The discussion is limited to comparison-based algorithms. We devise an adversary that, given the cardinalities n_i and a selection algorithm, provides input sets N_i such that the algorithm requires $\Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{max})$ messages when executed on that input.

The adversary is free to make each element arbitrary large or small, as long as the relative order in each N_i is maintained consistently. Initially, none of the elements has a fixed magnitude. The adversary follows the execution of the algorithm, fixing the magnitude of elements as the algorithm proceeds. Elements not yet fixed are candidates for the median. Fixed elements are made either "very small" or "very large", in the sense that they are smaller or larger than all the remaining candidates in the network. By keeping an equal number of very small and very large elements at all times, such elements cannot be selected as the median. Total order is maintained among the fixed elements of all processors by making each new very small element (very large element, resp.) larger (smaller) than all the existing very small (very large) elements.

W.l.g. assume n_i is even. Let $n_{i_1}, n_{i_2}, \ldots, n_{i_p}$ be a non-increasing order among the n_i's. The processors are divided into disjoint pairs $(P_{i_1}, P_{i_2}), (P_{i_3}, P_{i_4}), \ldots$, with P_{i_p} excluded from any pair if p is odd. Denote a typical pair (P_a, P_b). The adversary initializes the smallest $\frac{n_a - n_b}{2}$ elements in P_a to be very small, and the largest $\frac{n_a - n_b}{2}$ elements in P_a to be very large. Thus, P_a and P_b have an equal number of candidates. Also, if p is odd, $\frac{n_{i_p}}{2}$ elements in P_{i_p} are made very small, and the remaining elements in P_{i_p} are made very large.

Whenever a message is sent which contains a candidate of P_a that is smaller or equal (larger, resp.) than the median of the candidates in P_a, the adversary fixes that candidate and all those smaller (larger) than it in P_a to be very small (very large), and an equal number of candidates in P_b to be very large (very small). When the message contains a candidate of P_b, the same action with the roles of P_a and P_b reversed is taken. No action is taken by the adversary if the message does not contain a candidate. Concurrent messages are serialized in some arbitrary order.

Let $2m$ be the combined number of candidates in a given pair of processors immediately prior to the sending of a message containing a candidate of that pair. It can be seen that at most $m+1$ candidates, all of them in the given pair, are eliminated by the adversary as a result of that message. Thus, at least $\sum_{j=1}^{\lfloor \frac{p}{2} \rfloor} \log 2n_{i_{2j}} \geq \frac{1}{2} \sum_{j=2}^{p} \log 2n_{i_j} = \Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{\max})$ messages are needed to reduce the number of candidates in the network to one. ■

Clearly, a lower bound on messages in the MCB model immediately implies a lower bound, smaller by a factor of k, on cycles.

Corollary 1. The number of cycles required to select the median of n elements is $\Omega(\frac{1}{k} \sum_{i=1}^{p} \log 2n_i - \frac{\log 2n_{\max}}{k})$. ■

We now generalize the lower bounds for an arbitrary rank d.

Theorem 2. Let d be an integer, $p \leq d \leq \lceil \frac{n}{2} \rceil$. Let s be the number of processors for which $n_i \geq \frac{d}{p}$. Let $n_{i_1}, n_{i_2}, \ldots, n_{i_p}$ be a non-increasing order among the n_i's. The number of messages required to select the d'th largest element is $\Omega((s-1)\log \frac{2d}{p} + \sum_{j=s+1}^{p} \log 2n_{i_j})$.

Proof. We use the same adversary argument as in Theorem 1. The only difference is in the initial choice of candidates. Consider a typical pair of processors (P_a, P_b). For $b \geq s+1$, all the n_b elements in in P_b and an equal number of elements in P_a are made candidates. In the remaining pairs, candidates are chosen so that the total number of candidates in the network will not exceed $2d$, and each processor will have at least $\frac{d}{p}$ candidates, with an equal number of candidates in both processors of a pair. Let $2m$ denote the initial number of candidates. Among the remaining $n-2m$ elements, an arbitrary but consistent set of $d-m$ elements are made very large, and the others very small. Clearly, the median of the candidates is $N[d]$. Let m_i denote the initial number of candidates in P_i. Similar to Theorem 1, the number of messages required is at least $\sum_{j=1}^{\lfloor \frac{s}{2} \rfloor} \log 2m_{i_{2j}} \geq \frac{1}{2} \sum_{j=2}^{p} \log 2m_{i_j} = \Omega((s-1)\log \frac{2d}{p} + \sum_{j=s+1}^{p} \log 2n_{i_j})$. ■

Corollary 2. The number of cycles required to select the d'th largest element, $p \leq d \leq \lceil \frac{n}{2} \rceil$, is $\Omega(\frac{s-1}{k} \log \frac{2d}{p} + \frac{1}{k} \sum_{j=s+1}^{p} \log 2n_{i_j})$. ■

4.2. Lower Bounds on Sorting

We begin by showing a lower bound on the number of messages.

Theorem 3. The number of messages required to sort n elements distributed among p processors is $\Omega(n-n_{\max}+n_{\max 2})$.

Proof. As in selection, we consider only comparison-based algorithms. Given the input N and the cardinalities n_i, we will devise sets N_i such that $\Omega(n-n_{\max}+n_{\max 2})$ messages are required. Regardless of how the sorting is performed, each element must be directly compared with its immediate predecessor and successor in the final order. The sets N_i are constructed so that for sufficiently many disjoint comparisons the two elements to be compared are in different processors, thus requiring at least one message per comparison.

Let q_l denote the number of processors for which $n_i \geq l$. Consider the distribution where $N_i[j] = N[i + \sum_{l=1}^{j-1} q_l]$. In other words, the elements are distributed going circularly over all processors that have not yet reached their element capacity, and placing one element at a time in the sorted order in each processor. It can be easily seen that in this distribution no two immediate neighbors in $N[1, n-(n_{\max}-n_{\max 2})]$ are in the same processor. Thus, $\lfloor \frac{n-(n_{\max}-n_{\max 2})}{2} \rfloor = \Omega(n-n_{\max}+n_{\max 2})$ messages are required to complete the sorting. ■

Corollary 3. The number of cycles required to sort n elements is $\Omega(\frac{n-n_{max}+n_{max2}}{k})$. ■

We now show a different lower bound on the number of cycles.

Theorem 5. The number of cycles required to sort n elements distributed among p processors is $\Omega(\min\{n_{max}, n-n_{max}\})$.

Proof. Let P_{max} denote a processor for which $n_i=n_{max}$. Assume first that $n_{max}\leq\frac{n}{2}$ (i.e., $\min\{n_{max}, n-n_{max}\}=n_{max}$). Consider a distribution where for all $1\leq j\leq n_{max}$, $N[2j]$ is in P_{max} whereas $N[2j-1]$ is in some other processor. Using an argument similar to Theorem 3, at least n_{max} messages are required to compare the elements in $N[1, 2n_{max}]$. However, since P_{max} is involved in all these messages (either as sender or as receiver), the number of cycles required is also at least n_{max}. A similar argument shows that when $n_{max}>\frac{n}{2}$, at least $n-n_{max}$ cycles are required, and the result follows. ■

5. Sorting Algorithms for Even Distributions

The sorting algorithms described here are based on Leighton's Columnsort [Leig84], which in turn is a generalization of the Odd-Even sorting algorithm [Knut73]. Columnsort is useful for our purposes because of the well-behaved fashion in which elements are moved around, which helps overcome the "bottleneck" of only k channels. We begin by describing Columnsort. We then give a simple implementation of the algorithm for the case where the distribution is even and $p=k$, then show how to generalize it for $p>k$. In Section 6 we present several improvements to the basic algorithm, and in Section 7 we further generalize the algorithm for uneven distributions.

5.1. Columnsort

Let $n=m\cdot k$. We view the input N as a matrix of size $m\times k$, or alternatively, as a set of k columns of length m. The algorithm uses four transformations on the matrix, which are described informally below (also see Figure 1 for examples). Assume k divides m.

Transpose — Take the elements of the matrix column after column beginning with column 1, and store them row after row beginning with row 1.

Un-Diagonalize — Take the elements diagonal after diagonal, i.e., in the (column, row) order (1,1), (2,1), (1,2), (3,1), (2,2), (1,3), . . . , (k,m), and store them column after column beginning with column 1.

Up-Shift — Viewing the matrix as a linear list in lexicographical order by (column, row), shift each element $\lfloor \frac{m}{2} \rfloor$ positions in the ascending direction. The last $\lfloor \frac{m}{2} \rfloor$ elements are shifted circularly to the beginning of the list.

Down-Shift — This is similar to up-shift, except that the direction of shift is reversed.

At the end of the Columnsort algorithm, the elements are stored in descending order of magnitude, column after column beginning with column 1. The algorithm consists of 8 phases.

Phase 1. Sort each column in descending order.
Phase 2. Transpose the matrix.
Phase 3. Sort each column.
Phase 4. Un-diagonalize the matrix.
Phase 5. Sort each column.
Phase 6. Up-shift the matrix.
Phase 7. Sort each column except column 1.
Phase 8. Down-shift the matrix.

Proof of correctness of the algorithm is given in [Leig84]. The algorithm works only if the dimensions of the matrix satisfy the inequality $m \geq k(k-1)$, the reason being that otherwise the transformations, which are the crux of the algorithm, are not effective. In other words, in order to use k columns, the number of input elements, n, must be at least $k^2(k-1)$. Also, the transformations require that k divide m.

5.2. Implementation of Columnsort in the MCB Model

We first describe a simple implementation for the case where $p=k$ and the distribution is even. Column i is implemented in processor P_i, with N_i as the initial column data. Each column is thus of length $m=n_i=\frac{n}{k}$. Phases 1, 3, 5 and 7 are executed locally at each processor, using some efficient sequential sorting algorithm [Knut73]. The remaining phases consist only of traffic over the channels. Since $p=k$, all processors may broadcast simultaneously, with processor P_i using channel C_i. Thus, to move an element from column i to column j, processor P_j must read channel C_i in the correct cycle. It remains to devise an efficient broadcast schedule, so that a maximum number of elements could be moved simultaneously.

We give a schedule for phase 2. Similar schemes can be devised for phases 4, 6 and 8. Let us number the cycles of phase 2 sequentially, beginning with cycle 0. During cycle j, processor P_i sends the element in position $(i+j \bmod m)+1$ in its column, and reads channel $[(i-(j \bmod k)-2) \bmod k]+1$. As can be easily verified, the schedule correctly implements the transpose operation, and the transformation is completed in m cycles.

The actual order in which elements are kept in the target columns during phases 2, 4 and 6 is immaterial because the columns are sorted in the next phase anyway, with the exception of column 1 after phase 6. In the latter case, the effect of not sorting column 1 is that the same elements which are shifted from column k into column 1 in phase 6 are shifted back into column k in phase 8. Thus, P_1 need not keep these elements in any specific order. Alternatively, these elements need not be shifted at all. To similarly avoid having to maintain order during phase 8, an additional sorting phase at each processor, phase 9, is added.

During each cycle of phases 2, 4, 6, and 8, each processor broadcasts one element, and no element needs to be broadcast twice. Thus, the number of messages in each of these phases is $O(mk)=O(n)$, and the number of cycles is $O(m)=O(\frac{n}{k})$. The sorting phases are implemented locally, and thus have no communication cost. The total complexity of the algorithm is therefore $O(mk)=O(n)$ messages and $O(m)=O(\frac{n}{k})$ cycles. Since $n_{\max}=n_{\max 2}$, by Theorem 3 and Corollary 3 the algorithm is optimal. Note that the lower bounds for messages and for cycles are achieved simultaneously.

We now show how to extend the algorithm for the case $p>k$. A simple approach is to augment the algorithm with a preprocessing phase and a postprocessing phase, numbered phase 0 and phase 10, respectively. In phase 0, all elements are collected into k processors. Phases 1-9 then proceed as before, except that only k of the processors are active. In phase 10, the sorted elements are redistributed to all the processors. Notice that this implementation requires an additional $\Omega(\frac{n}{k})$ memory in k of the processors. In Section 6 we give a different implementation which avoids this extra memory.

Phase 0 is implemented as follows. W.l.g. assume k divides p. The processors are divided into k groups of equal size $\frac{p}{k}$. Group j consists of all processors P_i such that $\lceil \frac{ik}{p} \rceil = j$. The highest-numbered processor in the group, $P_{\frac{jp}{k}}$, is the group's "representative". One processor after another in ascending order of processors within the group, all elements of the group are sent to the representative. All groups proceed in parallel, group j using channel C_j. Phase 10 is the inverse of phase 0. The representatives broadcast, and each group member collects its respective elements.

The number of elements in each group is $m=\frac{n}{p}\cdot\frac{p}{k}=\frac{n}{k}$. Thus, phases 0 and 10 each require $O(\frac{n}{k})$ cycles and $O(n)$ messages. Phases 1-9 have the same complexity as before. The total complexity is therefore $O(n)$ messages and $O(\frac{n}{k})$ cycles, and the algorithm remains optimal for $p>k$.

The transformations require that the column length be a multiple of the number of columns. In case $\frac{n}{k}$ is not a multiple of k, all we need to do is pad each column at the end of phase 0 with at most $k-1$ dummy elements to get an appropriate column length. Phases 1-9 then proceed as before. Due to the padding, the elements that go to a given processor after the sorting might not all be in the same column at the end of phase 9. However, since the length of a column is at least $\frac{n}{k}$, the elements of each processor are guaranteed to be in at most two consecutive columns. In phase 10, group representatives must therefore broadcast each element twice, so that processors will be able to receive all their elements without missing any messages. Clearly, the complexity of the algorithm does not change.

As indicated before, inputs of size $n<k^2(k-1)$ cannot be sorted using k columns. To handle inputs of such size, we need to use fewer columns. Clearly, $\lfloor n^{1/3} \rfloor$ columns will do. The processors are divided into groups of equal size $\lceil \frac{p}{\lfloor n^{1/3} \rfloor} \rceil$, with the last group possibly padded with dummy processors containing dummy elements. Each column is now of size $O(n^{2/3})$. The complexity of the algorithm in this configuration is therefore $O(n)$ messages and $O(n^{2/3})$ cycles. In other words, the number of messages remains optimal, whereas the number of cycles becomes suboptimal. In section 6 we provide an improvement which reduces the range of input sizes that exhibit suboptimal performance.

6. Improvements to the Sorting Algorithm

6.1. Sorting with Efficient Memory Utilization

The $\Omega(\frac{n}{k})$ auxiliary memory needed in each representative processor can be avoided with a more efficient implementation. We consider each group of processors as a single virtual processor with a single virtual column, thus avoiding altogether the need for phases 0 and 10. In the sorting phases, virtual columns are sorted as if each group of processors were a separate MCB($\frac{p}{k}$, 1). Below, we describe a single-channel sorting algorithm called Rank-Sort, which works in linear number of messages and cycles and requires only $O(n_i)$ additional storage in each processor. Using this algorithm to sort each virtual column, the total complexity of each sorting phase is $O(\frac{n}{k})$ cycles and $O(n)$ messages. In the transformation phases, all the work of

a virtual processor during a given cycle is carried out by the processor containing the element to be broadcast in that cycle. The element received during the cycle can be stored over the one just sent, thus using only $O(1)$ additional storage in each processor. The total cycle and message complexities of the Columnsort algorithm remain the same as before.

The Rank-Sort algorithm proceeds as follows. Each processor maintains a rank counter for each of its elements. Initially, the value of all counters is 1. In the first phase, elements are broadcast in arbitrary order (e.g., column by column). After each broadcast, the counters of those elements which are smaller than the one broadcast are incremented by 1. Thus, at the end of the first phase each processor knows the rank of each of its elements. Then, in the second phase, the elements are broadcast in rank order and moved to the appropriate target processor. It is easy to see that the algorithm uses a linear number of cycles and messages, using only $O(n_i)$ auxiliary storage at each processor. Notice also that the algorithm works for arbitrary (even or uneven) distributions.

We may further economize on storage during the sorting phases by replacing the Rank-Sort algorithm with the Merge-Sort algorithm described below. Using this algorithm, the entire Columnsort scheme requires only $O(1)$ auxiliary storage at each processor. The Merge-Sort algorithm, however, is more complicated than Rank-Sort.

The Merge-Sort algorithm proceeds as follows. First, each processor sorts its input list. Then, in repetitive phases, the next largest element in the network is chosen among the top elements of the lists and moved to the appropriate target processor. In order to efficiently keep track of the top elements, the processors maintain a distributed linked list of the top elements sorted in descending order. Each processor knows its own top element and the next smaller one. The latter plays the role of a "pointer" in the linked list. In addition, each processor knows its rank in the distributed list.

Let P_a be the processor at rank 1 in the current phase. P_a sends its top element to the appropriate target processor, and all the processors decrement their rank by 1, thus effectively removing the head of the linked list. To insert the new top element of P_a into the list, P_a broadcasts the element, and all processors compare it with their own top element. Let P_b be the unique processor, if any, which has both a top element that is larger than the new element and a pointer that is smaller or null. The new element is inserted after that of P_b. This is effectively accomplished as follows. All processors with a smaller top element than that of P_a increment their rank by 1. P_b broadcasts its rank incremented by 1 and its current pointer, then changes its pointer to the new element. P_a sets its rank and pointer to those sent by P_b. In case the new element is larger than all the other top elements (i.e., P_b does not exist), which can be detected by silence on the channel when the rank and pointer are supposed to be broadcast, P_a sets its rank to 1. The linked list is initially constructed by broadcasting the top

elements of all processors in some order and inserting them into the list one after the other.

To achieve $O(1)$ auxiliary memory utilization, whenever an element is moved to its target processor, the target processor sends its smallest remaining input element as replacement to the processor at the head of the linked list, who then inserts this element in the proper position in its own input list. With this scheme, no extra storage is needed for output lists, and each processor uses only $O(1)$ auxiliary memory. Clearly, the Merge-Sort algorithm runs in linear number of cycles and messages.

6.2. A Recursive Implementation of Columnsort

As has been noted before, Columnsort works only if m, the length of each column, and k, the number of columns, satisfy the inequality $m \geq k(k-1)$. We have seen that this results in suboptimal cycle performance when $n < k^2(k-1)$. The range of input sizes with suboptimal performance can be effectively reduced by applying the basic algorithm recursively, as shown below. The general approach is to perform the sorting phases of a given level of the recursion by invoking the next level. The purpose of this scheme is to reduce the length of the columns from level to level, so that at the deepest level it will be possible to sort each column in a small number of cycles.

In the following we describe the recursive algorithm. Let $s \geq 1$ be an integer such that $k \geq 4^s$ and $n \geq k^{\frac{3s+6}{3s+2}}$. W.l.g. assume that $n \geq 4k$, and that n, p and k are powers of 4^s (i.e., $n = 4^{sr}$, $p = 4^{sq}$ and $k = 4^{sl}$ for some integers $r \geq q \geq l \geq 1$). Let $\overline{k} = (\frac{k^3}{n})^{\frac{1}{2s}}$ (since we only need to consider the case $n \leq k^3$, clearly $\overline{k} \geq 2$). We apply the recursion to a depth of $s+1$ levels. At any level except the last, a recursive call uses $\overline{k}$ virtual columns. Thus, at level $1 \leq j \leq s$, each virtual column consists of $\frac{n}{\overline{k}^j}$ elements, or, in other words, the lists of $\frac{p}{\overline{k}^j}$ processors (notice that the elements of the column are not collected into one processor). At the last level, level $s+1$, each call uses $\frac{k}{\overline{k}^s} = (\frac{n}{k})^{1/s}$ columns, and hence each column is of length $\frac{n}{k}$.

It can be easily verified using the assumptions on n, p and k given above that for $1 \leq j \leq s$, $\frac{n}{\overline{k}^j} \geq \overline{k}^2$. Thus, the requirement on the length of a column vs. the number of columns is satisfied at each level of the recursion. Since $\overline{k}^s \leq k$, there are sufficiently many channels for all the recursive calls of the same phase at the same level to proceed in parallel. At level j, each call is allocated a separate set of $\frac{k}{\overline{k}^{j-1}}$ channels. During the transformation phases, each virtual column is broken into $\frac{k}{\overline{k}^j}$ segments of $\frac{n}{k}$ elements each, and all segments are broadcast

simultaneously - each segment using a separate channel. The behavior of the processors corresponding to each segment is similar to a virtual processor in the algorithm of Section 6.1. Thus, the number of cycles used in each transformation phase at each level is $O(\frac{n}{k})$. At level $s+1$ each column is of length $\frac{n}{k}$, and thus the number of cycles for each sorting phase at that level is also $O(\frac{n}{k})$. The total cycle complexity of the algorithm is therefore $O(s\frac{n}{k})$. The message complexity is clearly $O(sn)$. The right choice of s will effectively reduce the range of suboptimal input sizes, as shown in the following corollary.

Corollary 5. Given a constant $\epsilon>0$ such that $n\geq k^{1+\epsilon}$ and $k\geq 4^{\lceil\frac{4-2\epsilon}{3\epsilon}\rceil}$, the complexity of sorting n elements distributed evenly in an MCB(p, k) is $\Theta(n)$ messages and $\Theta(\frac{n}{k})$ cycles.

Proof. The lower bounds follow from Theorem 3 and Corollary 3, since $n_{\max}=n_{\max 2}$. For $\epsilon\geq 2$, the upper bounds follow from the discussion in Section 5, since $n=k^{1+\epsilon}\geq k^2(k-1)$. For $\epsilon<2$, the result follows from the discussion above, using $s=\lceil\frac{4-2\epsilon}{3\epsilon}\rceil$. Notice that the tight bounds on messages and cycles are achieved simultaneously. ■

7. Sorting Algorithms for Uneven Distributions

In this section we extend the sorting algorithm of Section 5 to the case of uneven distributions. First, however, we digress to describe a simple algorithm to compute partial sums. We will use this algorithm as a subroutine in the sorting algorithm, as well as in the selection algorithm in Section 8.

7.1. A Partial-Sums Algorithm

Let $\oplus$ denote a commutative and associative arithmetic operator, such as "+", "max", etc. Let $\{a_1, a_2, \ldots, a_p\}$ be a set of values such that a_i is in P_i. We denote by $a_i^{\oplus}$ the partial sum $a_1\oplus a_2\oplus\cdots\oplus a_i$. The largest partial sum, $a_p^{\oplus}$, is called the total sum. In the following we describe an algorithm to efficiently compute at each P_i the partial sum $a_i^{\oplus}$. The algorithm is based on Vishkin's F&* tree machine [Vish84], which is used for similar purposes in a different model. We first describe the tree machine, then the implementation in our model.

W.l.g. assume $p=2^r$ for some integer r. Consider a synchronous network in the shape of a full binary tree with p leaves. Each node of the tree is a processor, and communication is point-to-point along the edges. Processors are uniquely labeled by the pair $\langle l, j\rangle$, where $l\geq 0$ is the level of the processor in the tree and $j\geq 1$ is a running index from left to right at each level. The leaves are in the lowest level, level 0. Each processor knows which of its incident

links leads to the father, the left son, and the right son, respectively. a_i is initially in leaf processor $\langle 0, i \rangle$. The following computation produces the partial sums $a_i^{\oplus}$ at the leaves.

Assume the local variables L, R, and F at each processor contain the last value received from the left son, right son, and father, respectively. The computation consists of a bottom-up phase followed by a top-down phase. The bottom-up phase starts with the leaves sending a_i to their father. Upon receiving values from both sons, an internal processor sends $L \oplus R$ to its father. When the computation reaches the root, the top-down phase is started with the root sending ω to its left son and L to its right son, where ω denotes the identity value with respect to $\oplus$. Each internal processor, upon receiving a value from the father, sends F to the left son and $F \oplus L$ to the right son. When the computation returns to the leaves, each leaf sets $a_i^{\oplus} = F \oplus a_i$. It is easy to show that the computation works correctly. Note that each leaf also knows the value $a_{i-1}^{\oplus}$.

The implementation on an MCB(p, k) is straightforward. The tree computation is simulated level by level, first bottom up, then top down. A father node is simulated by the same processor that simulates its left son, thus only the messages between father and right son need actually be sent. Let us number the cycles of the algorithm at each level sequentially, beginning with cycle 1. The computation at level l in the bottom-up phase proceeds as follows. The processor simulating node $\langle l, 2j \rangle$ writes on channel $(j-1 \bmod k)+1$ during cycle $\lceil \frac{j}{k} \rceil$. The message is read by the processor simulating node $\langle l+1, j \rangle$. Thus, each level $0 \leq l \leq \log \frac{p}{k} - 1$ in the bottom-up phase can be executed in $\frac{p}{k2^{l+1}}$ cycles. The remaining $\log k$ levels require one cycle each. A similar scheme can be devised for the top-down phase. The total number of cycles is therefore $O(\frac{p}{k} + \log k)$. The total number of messages is clearly $O(p)$.

With an additional p messages and $\frac{p}{k}$ cycles (i.e., at no added complexity) each processor P_i can acquire $a_{i+1}^{\oplus}$ from P_{i+1}. We assume, therefore, that the Partial-Sums algorithm yields at each P_i the values $a_{i-1}^{\oplus}$, $a_i^{\oplus}$ and $a_{i+1}^{\oplus}$. Notice that at the root of the simulated tree $L \oplus R = a_p^{\oplus}$. Thus, if only the total sum is of interest, the bottom-up phase followed by a single broadcast message from P_1 (which simulates the root) suffices.

7.2. The Sorting Algorithm

The implementation of Columnsort in Section 5 relies heavily on the fact that each processor has the same number of elements. In the following we discuss the problems that arise from an uneven distribution of the input, and how they can be solved.

When the distribution is even, it is trivial to divide the processors into groups which form columns of equal length. With an uneven distribution, we instead form groups so that the columns are of "roughly equal" length. After the elements of each group are collected into one processor, the columns are padded to make their lengths exactly equal.

Now consider the task of collecting the elements. In the even case, there is no need to synchronize the write access to the single channel shared by each group - each processor P_i simply waits $\frac{n}{p}(i-1 \bmod k)$ cycles for its turn. In the uneven case, on the other hand, the number of cycles a processor has to wait before writing on the channel depends on the distribution, and thus needs to be explicitly calculated.

We distinguish two subphases in phase 0: group formation, and element collection. Groups are formed so that the number of elements in each group j, denoted m_j, is in the range $\frac{n}{k} \leq m_j \leq \frac{n}{k} + n_{\max} - 1$. Clearly, there are at most k groups, and thus requiring $n \geq k^2(k-1)$ suffices to guarantee that the columns are long enough. The groups are formed one at a time in the following fashion. Using the Partial-Sums algorithm, the processors compute $n_{\max}$ and the partial sums n_i^+. Group 1 is then formed by processors P_i for which $n_i^+ \leq \frac{n}{k} + n_{\max} - 1$. Let P_l be the highest-numbered processor in the group. Clearly, $m_1 = n_l^+$. Processor P_l becomes the group representative and informs the rest of the network of its id and of the number of elements in the group (note that P_l can easily identify itself as the representative by checking the partial sum of the next higher processor). Processors which are not in group 1 subtract m_1 from their partial sum. Group 2 is then formed in a similar fashion using the revised partial sums. The scheme is repeated until all groups are formed. The costs incurred by the group formation phase include two applications of Partial-Sums and the broadcasts from the group representatives, for a total of $O(\frac{p}{k} + k)$ cycles and $O(p)$ messages.

Element collection proceeds similar to the even case, except for the synchronization among processors. The number of cycles each P_i has to wait before writing on the channel is given by $n_i^+ - n_i$, where n_i^+ is the revised partial sum of the processor upon joining a group. After the elements have been collected, each column is padded with dummy elements up to length $\max_j\{m_j\}$. Additional padding may be needed to make the column length, m, a multiple of the number of columns. Clearly, $m = O(\frac{n}{k} + n_{max})$. The cost of the element collection phase is $O(n)$ messages and $O(\frac{n}{k} + n_{\max})$ cycles.

Notice that during element collection the groups proceed independently, each group at its own pace. A synchronization point for the beginning of phase 1 can be set at m cycles after the beginning of element collection. Phases 1-9 then proceed as in the even case, except that there may be fewer than k columns. The total complexity of phases 1-9 is $O(m)=O(\frac{n}{k}+n_{\max})$ cycles and $O(n)$ messages (the dummy elements need not be broadcast). In phase 10, due to the padding of columns, group representatives broadcast each element twice, as discussed in the even case. The complexity of phase 10 is also $O(\frac{n}{k}+n_{\max})$ cycles and $O(n)$ messages.

Summing the costs of all phases, the total complexity of the sorting algorithm is $O(\frac{n}{k}+n_{\max})$ cycles and $O(n)$ messages. The following corollary shows that this is optimal in a wide range of cases.

Corollary 6. Given a constant $0<\alpha<1$ such that $n_{\max}\leq\alpha n$ and $n\geq k^2(k-1)$, the complexity of sorting n elements on an MCB(p, k) is $\Theta(n)$ messages and $\Theta(\max\{\frac{n}{k}, n_{\max}\})$ cycles.

Proof. $n_{\max}\leq\alpha n$ implies $n-n_{\max}\geq(1-\alpha)n\geq(1-\alpha)n_{\max}$. The result then follows from the analysis above and from the lower bounds of Theorems 3 and 4 and Corollary 3. Note that the tight bounds on messages and cycles are achieved simultaneously. ■

8. An Algorithm for Selection

A naive approach to selection is to sort all elements, then retrieve the desired element directly by rank. This, however, is inefficient because the extra information provided by sorting comes at a cost and is not really needed. A more promising approach is the following. Reduce the number of candidates for selection by repetitively applying an efficient filtering mechanism. When the number of remaining candidates gets below a specified threshold value, sort the remaining candidates and retrieve the selected element by rank. In the sequel we present a selection algorithm which follows this approach. The algorithm works for arbitrary (even or uneven) distributions. We first describe the algorithm, then prove its correctness and analyze its complexity.

8.1. Description of the Algorithm

Let us denote the set of remaining candidates for selection at each stage of the algorithm by M, with cardinality $|M|=m$. The subset of candidates in each P_i is denoted M_i, with cardinality $|M_i|=m_i$. Initially, $M=N$ and $M_i=N_i$. The median of each M_i is denoted med_i. The threshold value discussed above is denoted m^*, and d is the rank of the element to be selected. As indicated before, we may w.l.g. assume that all elements are distinct. The

algorithm consists of a repetitive filtering phase, followed by a termination phase.

A typical filtering phase proceeds as follows. Each P_i computes med_i using an efficient sequential selection algorithm ([Blum73], for example). If there are no candidates left in P_i, med_i is given a dummy value. Using the sorting algorithm of Section 5, the pairs (med_i, m_i) are sorted in descending order of the first coordinate. We denote the pair located at processor P_i after the sorting as (med'_i, m'_i), to distinguish it from the original pair at that processor. Using the Partial-Sums algorithm of Section 7, the processors compute $m'^{+}_i = m'_1 + \cdots + m'_i$. Let m'^{+}_l be the smallest partial sum such that $m'^{+}_l \geq \lceil \frac{m}{2} \rceil$. We denote the corresponding median, med'_l, as $med_{1/2}$. Intuitively, $med_{1/2}$ is chosen so that sufficiently many candidates are larger than it, and sufficiently many are smaller. P_l broadcasts $med_{1/2}$ to the other processors. Then, using the Partial-Sums algorithm, the processors calculate the total number of candidates that are grater or equal to $med_{1/2}$. Denote this number by $m_{1/2}$. There are three cases.

Case 1. $m_{1/2} = d$

The selected element is $med_{1/2}$, and the algorithm terminates.

Case 2. $m_{1/2} > d$

All the candidates smaller or equal to $med_{1/2}$ are purged and m is set to $m_{1/2}-1$. If $m > m^*$, the filtering phase is repeated, otherwise the termination phase is executed.

Case 3. $m_{1/2} < d$

All the candidates greater or equal to $med_{1/2}$ are purged, m is set to $m-m_{1/2}$, and d is set to $d-m_{1/2}$. Then, similar to case 2, if $m > m^*$ the filtering phase is repeated, otherwise the termination phase is executed.

The termination phase proceeds as follows. First, the processors compute the partial sums m_i^+ of the remaining candidates. Then, one processor after another, the processors send all the remaining candidates to P_1. Similar to the sorting algorithm, processors await their turn to write by counting cycles. The number of cycles P_i has to wait is given by the value m_i^+ just computed. When all candidates have been collected, P_1 sorts the candidates, then selects the element of rank d and broadcasts it to the other processors.

8.2. Correctness and Complexity

We now show that the selection algorithm works correctly. Assume inductively that at the beginning of the current phase the element to be selected has not been purged, and that d is the correct rank of this element among the remaining candidates. This is clearly true at the beginning of the algorithm.

In case 1, the number of candidates grater or equal to $med_{1/2}$ is found to be d. Since w.l.g. we assume that all elements are distinct, the decision to select $med_{1/2}$ is correct. In case 2, since $m_{1/2} > d$, the element we are looking for is greater than $med_{1/2}$. Thus, all candidates smaller or equal to $med_{1/2}$ can be purged. In case 3 a similar argument applies, except that the $m_{1/2}$ candidates being purged are greater than the selected element, so the rank d has to be reduced by the same quantity.

Since at least a few candidates are purged in each of cases 2 and 3, the algorithm will eventually either terminate in case 1 or reach the termination phase. The termination phase clearly selects the correct element, as the remaining candidates are collected into one processor and sorted.

In analyzing the complexity of the algorithm, we must calculate the costs of the filtering and the termination phases, and determine the number filtering phases. The cost of a filtering phase involves the following: (1) sorting the medians; (2) computing $med_{1/2}$; and (3) computing $m_{1/2}$. Using the costs of Columnsort and Partial-Sums under the assumption $p \geq k^2$, the complexity of each filtering phase is $O(\frac{p}{k})$ cycles and $O(p)$ messages. The cost of the termination phase is clearly $O(m^*)$ messages and cycles.

The analysis of the number of filtering phases is illustrated by Figure 2, which captures the situation at the beginning of a typical filtering phase. The lists of candidates M_i are ordered in descending order of the medians from left to right. The elements in each list are sorted in descending order from top to bottom. Since the medians are ordered, it can be seen that for any given list M_i, half the candidates in M_i and at least half the candidates in each list to the right of M_i are smaller or equal to med_i. Similarly, half the list M_i and at least half of each list to the left of M_i is greater or equal to med_i. This is shown by the encircled areas in Figure 2.

In particular, using the fact that $m_{1/2}$ is the smallest partial sum greater or equal to $\lceil \frac{m}{2} \rceil$, it can be seen that at least $\lceil \frac{m}{4} \rceil$ candidates are smaller or equal to $med_{1/2}$, and at least $\lceil \frac{m}{4} \rceil$ candidates are greater or equal to $med_{1/2}$. Consequently, in each of cases 2 and 3 of the

algorithm, at least one fourth of the remaining candidates are purged. Thus, $O(\log\frac{n}{m^*})$ filtering phases suffice in order to reduce the number of candidates below m^*.

The total complexity of the selection algorithm is therefore $O(m^*+\frac{p}{k}\log\frac{n}{m^*})$ cycles and $O(m^*+p\log\frac{n}{m^*})$ messages. If we choose $m^*=\frac{p}{k}$, the complexity of the algorithm is $O(\frac{p}{k}\log\frac{kn}{p})$ cycles and $O(p\log\frac{kn}{p})$ messages. The following corollary shows that this is optimal for a wide range of cases.

Corollary 7. Let $p\geq k^2$, and let $0<\epsilon\leq 1$ be a constant such that $n\geq\frac{pk}{\epsilon^2}$ and $\frac{\epsilon n}{2}\leq d\leq\lceil\frac{n}{2}\rceil$. Also, let the distribution of the input be such that for at least $\frac{\epsilon p}{2}+1$ processors $n_i\geq\frac{d}{p}$. The complexity of selecting the d'th largest element in an MCB(p, k) is $\Theta(\frac{p}{k}\log\frac{kn}{p})$ cycles and $\Theta(p\log\frac{kn}{p})$ messages.

Proof. $n\geq\frac{pk}{\epsilon^2}$ implies $\log\frac{2d}{p}\geq\log\frac{\epsilon n}{p}\geq\frac{1}{2}\log\frac{kn}{p}$. The result then follows from the analysis above, and from the lower bounds of Theorem 2 and Corollary 2 using $s=\lceil\frac{\epsilon p}{2}\rceil+1$. Note that the tight bounds on cycles and messages are achieved simultaneously. ■

9. Conclusions

In this paper we have presented a model for the design of distributed algorithms using broadcast communication. We have demonstrated the practicality of the model by devising efficient algorithms for the problems of sorting and selection. The algorithms achieve tight performance bounds, both on cycles and on messages simultaneously.

Our results can be applied in other models as well. In [Marb85] we have implemented the selection algorithm in the Shout-Echo broadcast model, improving the previous best upper bound in that model [Rote83] by a factor of $O(\log p)$. The Columnsort algorithm for even distributions can be used in the CREW model, resulting in the same time complexity as the sorting algorithm in [Shil81], and reducing the auxiliary shared memory requirements to p memory cells. In the IPBAM model, the single-channel Merge-Sort algorithm achieves the same complexity as the sorting algorithm in [Dech84], but without the use of concurrent write.

The MCB model can be extended in various ways. For example, by allowing processors to access all channels during each cycle, or by allowing concurrent write access to the channels. As we have seen, such extensions are not needed in order to achieve optimal broadcast algorithms for sorting and selection. It is interesting to characterize the problems for which increasing the power of the model would, or would not, result in more efficient algorithms.

Acknowledgements

The authors wish to thank Rina Dechter for reviewing the manuscript.

References

[Blum73] Blum, M., R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *JCSS* **7**, 4 (Aug. 1973), pp. 448-461.

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

[Dech81] Dechter, R. and L. Kleinrock, "Parallel Algorithms for Multiprocessors Using Broadcast Channel," ATS Working Paper 81002, Computer Science Dept., Univ. of California, Los Angeles, CA, Nov. 1981.

[Dech84] Dechter, R. and L. Kleinrock, "Broadcast Communications and Distributed Algorithms," Technical Report, Computer Science Dept., Univ. of California, Los Angeles, CA, 1984. To appear in *IEEE Trans. on Computers.*

[Fred83] Frederickson, G.N, "Tradeoffs for Selection in Distributed Systems," in *Proceedings 2nd ACM Symp. on Principles of Distributed Computing,* 1983, pp. 154-160.

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

[Leig84] Leighton, T., "Time Bounds on the Complexity of Parallel Sorting," in *Proceedings 16th ACM Symp. on Theory of Computing,* 1984, pp. 71-80.

[Levi82] Levitan, S., "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, "An Optimal Shout-Echo Algorithm for Selection in Distributed Sets," Technical Report, Computer Science Dept., Univ. of California, Los Angeles, CA, March 1985.

[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 Conference,* 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.

[Rote83] Rotem, D., N. Santoro, and J.B. Sidney, "A Shout-Echo Algorithm for Finding the Median of a Distributed Set," in *Proceedings 14th S.E. Conf. on Combinatorics, Graph Theory and Computing,* Boca Raton, FL, 1983, pp. 311-318.

[Sant82] Santoro, N. and J. Sidney, "Order Statistics on Distributed Sets," in *Proceedings 20th Allerton Conf. on Communication, Control and Computing,* Univ. of Illinois at Urbana Champaign, 1982, pp. 251-256.

[Sant83] Santoro, N. and J.B. Sidney, "Communication Bounds for Selection in Distributed Sets," Working Paper 83-39, Faculty of Administration, Univ. of Ottawa, Ottawa, Canada, 1983.

[Shil81] Shiloach, Y. and U. Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model," *J. of Algorithms* **2**, 1 (March 1981), pp. 88-102.

[Snir83] Snir, M., "On Parallel Searching," Tech. Rep. 83-21, Dept. of Computer Science, Hebrew Univ., Jerusalem, Israel, June 1983.

[Vish84] Vishkin, U., "A Parallel-Design Distributed-Implementation (PDDI) General-Purpose Computer," *TCS* **32**, 1 (July 1984), pp. 157-172.

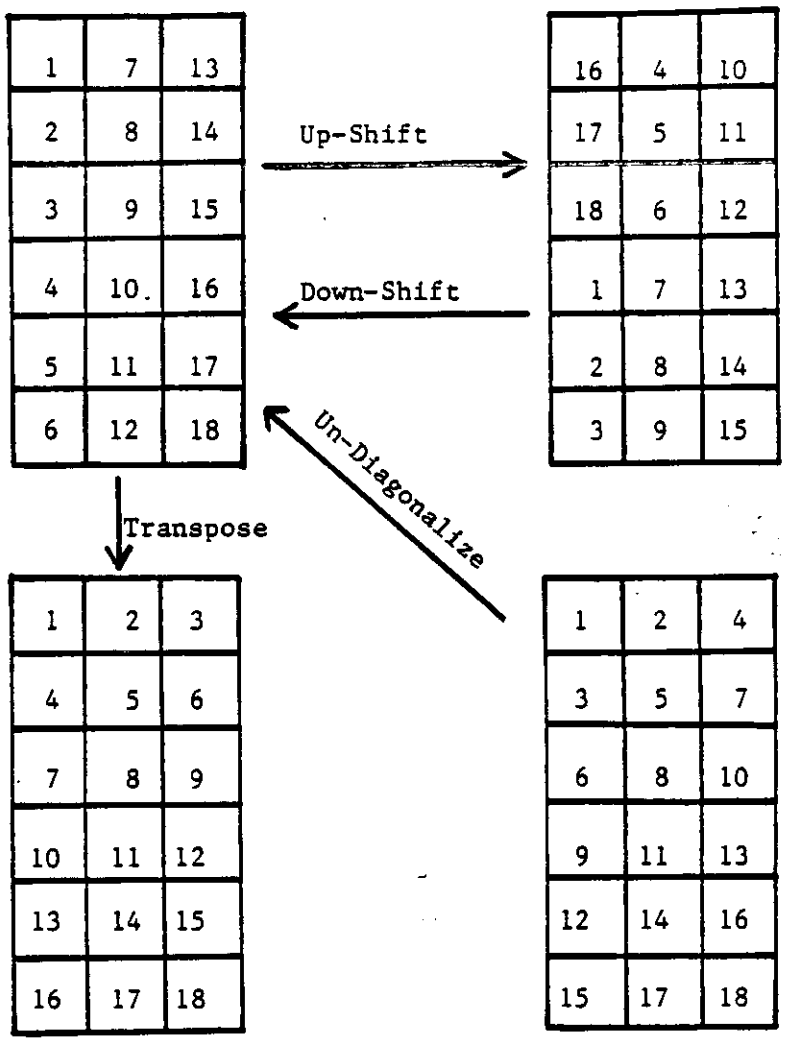

Figure 1. Matrix Transformations

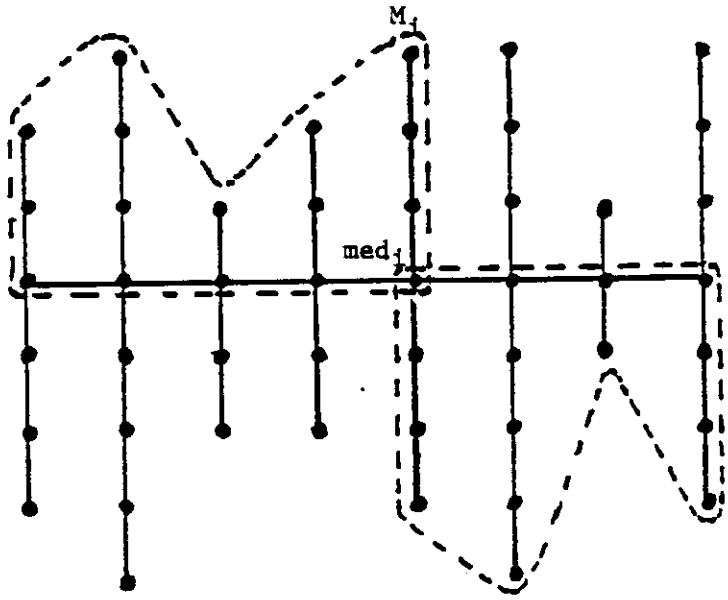

Figure 2. The Filtering Phase