#114

Broadcast Communications and Distributed Algorithms

RINA DECHTER AND LEONARD KLEINROCK, FELLOW, IEEE

Abstract — **The paper addresses ways in which one can use "broadcast communication" in distributed algorithms and the relevant issues of design and complexity. We present an algorithm for merging k sorted lists of n/k elements using k processors and prove its worst case complexity to be $2n$, regardless of the number of processors, while neglecting the cost arising from possible conflicts on the broadcast channel. We also show that this algorithm is optimal under single-channel broadcast communication. In a variation of the algorithm, we show that by using an extra local memory of $O(k)$ the number of broadcasts is reduced to n. When the algorithm is used for sorting n elements with k processors, where each processor sorts its own list first and then merging, it has a complexity of $O(n/k \log(n/k) + n)$, and is thus asymptotically optimal for large n. We also discuss the cost incurred by the channel access scheme and prove that resolving conflicts whenever k processors are involved introduces a cost factor of at least log k.**

Index Terms — **Access scheme, broadcast, complexity analysis, distributed algorithm, merging, parallel algorithms, sorting.**

I. Introduction

CONSIDER the following algorithm for finding the maximum of a set of k distinct, numerically valued elements where each element is stored within a separate processor. When the algorithm begins, each processor attempts to transmit its own value using a common broadcast channel to which all processors listen. However, only one processor is enabled (permitted) to transmit by means of some access scheme (conflict resolution scheme). Each processor compares its value to the largest value transmitted so far. All processors that have a larger value try again to broadcast their own values, etc. The algorithm terminates when all processors have either transmitted their values or have "given up," which is detected by silence on the channel. The last element to be broadcast is the maximum.

This admittedly simple algorithm (referred to as the "Max-Algorithm" and also presented in [16]) demonstrates how a distributed algorithm can utilize broadcast communication. The term broadcast implies the existence of a single channel on which only one node (processor) can transmit at one time, while all the others receive the message simultaneously.

"Algorithms by broadcasting" have not received much attention in the literature on parallel and distributed algorithms. An earlier report by the authors [8] was among the first discussions of such algorithms and those results constitute a portion of the current paper. Other contributions, appearing at around the same time, are [15] and [16]. In this introduction, we survey the motivation for using broadcasting as a model for distributed computation, point out its unique features, summarize relevant work and point out our contribution.

In the area of parallel algorithms, the closest thing to broadcasting is the assumption of the existence of a global or shared memory from which all the processors can simultaneously read the same value [5], [20]. However, shared memory models usually do not place any limit on the number of memory cells which are used by the processors. In the context of broadcasting, this would mean that there is more than one broadcast channel and that each processor can use any channel according to the requirements of the algorithm. In most broadcast-based networks (e.g., local area networks [18]), there is only one channel shared by all processors. Therefore, most of the results for shared memory models are not applicable to broadcasting. An example in which the results are appplicable is the search algorithm presented by Snir [20] using the CREW (concurrent read, exclusive write) model which utilizes only one memory location.

A major difficulty in using broadcast communication is the issue of access to the channel. Many access schemes have been proposed and analyzed [22]. The focus of most papers is on how to increase channel capacity and on the tradeoff between throughput and delay. Clearly, the access scheme may have a significant impact on the complexity of the algorithm. We will approach this problem through two models. In both models, processors broadcast one value at a time on a channel shared by all of them. Only one message will be posted at a slot on the channel, and all processors can read the posted message. In our first model, named IPABM (ideal parallel broadcast model), we assume that some "ideal" access scheme exists, i.e., if several processors demand the use of the channel at the same time, there is a global mechanism which enables one of them to transmit in a constant time. Later it is refined into a "more realistic" model, RPABM (realistic parallel broadcast model), that incorporates a conflict resolution protocol (CRP), and we discuss two specific access schemes and their influence on the time complexity of the algorithms.

The vehicle we use to study algorithms by broadcasting is via "comparison-based" algorithms (sorting, searching, etc.). Parallel versions of these algorithms have been extensively studied under various models of communications [2], [3], [5], [7], [19]–[21], [23], [24], and therefore, they are well-suited for studying the power and limitations of broadcast communication. For a review of sorting algorithms, see also [9].

Manuscript received July 13, 1983; revised October 4, 1985. This work was supported by the Defense Research Projects Agency under Contract MDA 903 82-C-0064.

The authors are with the Department of Computer Science, University of California, Los Angeles, CA 90024.

IEEE Log Number 8407299.

Relevant work in the area of broadcast algorithms includes the work by Levitan [15], who uses a PBM (broadcast protocol multiprocessor) model which is identical to our IPABM, and who obtains results similar to those given in the current paper; in particular, he presents a sorting algorithm which is identical to our second version of the merge algorithm when all processors have just one element. In addition, he gives an algorithm for finding a minimum spanning tree in a graph. Algorithms for finding the extrema in a broadcast model are presented in [16] and [4]. The latter uses a mixed model that, in addition to the conventional links, also allows a global bus for broadcast communication.

The main contributions of this paper are: an efficient algorithm for merging, proving its optimality, and dealing in a formal way with issues that emerge from the use of broadcasting as a model for distributed computing. In particular, in the analysis we take into account both the communication cost (time to broadcast a message) and the computation cost (time for performing a comparison). We also discuss the overhead introduced by different access schemes.

In the next section we present our model, discuss complexity issues and analyze the performance of the Max-algorithm discussed above. In Section III we present an algorithm for merging k sorted lists of n/k elements each. We show that the worst case performance of the algorithm is independent of the number of processors (which is also the number of lists), and is bounded from above by $2n - 1$ broadcasts and the same number of comparison stages. A comparison stage is one time slot in which several processors in parallel perform one comparison. In a variation of this algorithm, we show (Section III-C) that an additional $O(k)$ storage in each processor can reduce the number of messages broadcast to n. Using the merge algorithm for sorting n elements with k processors (Section III-D) yields a worst case time complexity of $O(n/k \log(n/k) + n)$. Thus, for large n, the Merge-sort algorithm achieves an asymptotic speedup ratio of k with respect to the best sequential (i.e., single processor) algorithm whose complexity is $O(n \cdot \log n)$. In Section IV we show the optimality of the Merge-algorithm by proving that any Merge-by-broadcast algorithm requires n broadcasts. Section V addresses access scheme issues.

II. The IPABM Model and Complexity Measures

The model which is used through most of the paper is presented next. Let us define an IPABM as a collection of processors which compute in parallel synchronously and which communicate via a single broadcast channel. The channel is slotted into time slots of size T (where T is the time for a message transmission). At each step, each processor can read the message in the current slot on the channel, do some computation and submit a message to be broadcast in the next time slot. Any number of processors can read the current message on the channel but only one message among those submitted for transmission will be chosen by the global access mechanism to be broadcast in the next time slot. An empty slot indicates that no processor wants to talk.

The complexity of an algorithm will be measured by its computing time and its communication time. Dealing with comparison-based algorithms, we consider a comparison operation as the basic computation step and the broadcast of a message as the basic communication step. Thus, the "number of comparisons" (#comparisons) performed in parallel and the "number of broadcasts" (#broadcasts) characterize the computation time and broadcast time, respectively. Let t be the time for a comparison operation (since the algorithms we consider are synchronized the analysis does not take into account variations in computing time among processors). The above two measures are combined as follows:

$$T(A) = t \cdot (\#\text{comparisons}) + T \cdot (\#\text{broadcasts}) \quad (1)$$

where $T(A)$ stands for the worst case time complexity of algorithm A. By $\overline{T(A)}$ we denote the average time complexity of algorithm A over all problem instances.

In the Max-algorithm each broadcast is followed by one comparison operation performed in parallel by some of the processors. Therefore, it is sufficient to account only for #broadcasts as the measure of complexity.

Let $T(\text{Max}(k))$ be the number of broadcasts performed by the Max algorithm with k elements and k processors. On some input instances $\text{Max}(k)$ will require each element to be broadcast and thus,

$$T(\text{Max}(k)) = k . \quad (2)$$

The average number of broadcasts, $\overline{T(\text{Max}(k))}$, obeys the following recurrence:

$$\overline{T(\text{Max}(k))} = 1 + \frac{1}{k} \sum_{i=1}^{k} \overline{T(\text{Max}(k - i))} . \quad (3)$$

The last is true since if the first element to be broadcast is the ith smallest element, then exactly $i - 1$ among the rest of the $k - 1$ processors will remain silent and will not participate in the rest of the algorithm. We assume a homogeneous distribution of the elements among the processors and thus the above event has probability $1/k$ and the recurrence follows. (3) can be written as

$$\overline{T(\text{Max}(k))} = 1 + \frac{1}{k} \sum_{i=0}^{k-1} \overline{T(\text{Max}(i))} \quad (4)$$

with $\overline{T(\text{Max}(0))} = 0$, $\overline{T(\text{Max}(1))} = 0$. Solving this recurrence yields

$$\overline{T(\text{Max}(k))} = \sum_{i=1}^{k-1} \frac{1}{k} \leq \log k . \quad (5)$$

The same results were obtained in [16] using a slightly different analysis.

III. Parallel Merge by Broadcast

A. Description of the Algorithm

We present a distributed algorithm that merges k sorted lists of n/k distinct elements into a decreasing series, using an IPABM with $k + 1$ processors. Each of the first k processors contains one of the lists and each has an identity (id#) and a local memory. The size of local memory is fixed and not dependent on k or n. For simplicity we designate the $(k + 1)$

processor to be the "output processor" (the one in which the output will be stored). All processors cooperatively participate in the task of merging the sorted lists they possess. The maximum element in each processor's list is called its "current value."

The algorithm can be decomposed into cycles. In each cycle the maximum of the current values is determined. This element is broadcast to the output processor as the next element in the merged list, and is removed from the processor to which it belonged (the processor updates its current value). Each cycle is implemented by the Max-algorithm presented earlier. The processor that broadcasts first is the **initiator** of the cycle; the last is the **terminator** of the cycle. During the cycle, processors try to broadcast their current values as long as they have not yet heard a larger value being broadcast. In order to eliminate redundant broadcasts, there will be some dependency between the initiations of cycles.

When a processor succeeds in broadcasting its current value, it denotes the value which is broadcast immediately afterwards as its successor. The successor value is updated each time the current value is rebroadcast. When the current value is the terminator of the cycle, it has no successors. The current value is the predecessor of its successor. Each current value will have at most one successor at any given time (it may have none, if it was not broadcast yet). It will also have at most one predecessor. In terms of this terminology, the rule for cycle initiation is as follows: a processor initiates the next cycle (by rebroadcasting its current value) if the present cycle was terminated by a successor to its own current value. If there is no predecessor, the next cycle can be initiated by any processor.

In order to implement the above algorithm in a distributed fashion, processors must be able to detect the end of a cycle and its terminator, and to determine whether or not they should initiate the next cycle. The end of a cycle is determined by silence (i.e., an empty time slot). The initiator of a cycle is determined by the successor–predecessor relationship, as described earlier. Two empty slots in succession indicate that a cycle is terminated but that a specific initiator does not exist, in which case all processors try to initiate the next cycle.

The algorithm for each processor is described in Fig. 1. While listening to the channel, a processor can recognize one of the following three cases. A value is broadcast in the current slot (case 1), or the current slot is empty but the previous one holds a value (case 2), or two consecutive empty slots have occurred (case 3). In cases 2 and 3 a cycle has terminated and an initiator must be determined. We assume that each processor has a procedure for determining the terminator of a cycle which is used in case 2. The procedure **process-update** is used each time the processor has successfully broadcast its current value, *CV*. It determines whether the value is also the terminator of the cycle, or whether the successor value, *SUCC*, should be updated. If it is the terminator, the value at the top of its list is removed and the value of *CV* is updated to be the new maximum on its list. The first broadcast of each current value may terminate the cycle in which it participated. If it did not, this value will be rebroadcast only for initiating future cycles until it will terminate one. Then, the current value is updated and the processor tries to broadcast the new current value for the first time. A formal proof of this behavior is given later.

```
main

begin
  /*initialization */
  CV <- the maximum value in list
  SUCC <- nil
  TERM <- nil
  BV <- nil

  repeat
    BV <- read next msg
1.  if BV is not empty then
      if CV > BV then broadcast CV
        if successful then call process_update
2.  if BV=empty then   /*a cycle is terminated*/
      begin
        TERM <- terminator of the cycle
        if TERM = SUCC then
            broadcast & call process_update
      end
3.  if heard two empty slots then try to broadcast CV
        if successful then call process_update
until list is empty


process_update

begin
  BV <- read next msg
  if BV = empty then
      begin
        remove CV from list
        if list is empty CV <- NIL else
        CV <- next element in list
      end
  else
    SUCC <- BV
end
```

Fig. 1. The Merge-algorithm.

It is convenient to trace the execution of the algorithm using a global work stack in which the values being broadcast are recorded. The work stack could also be kept in each of the processors and thus be used to control the algorithm (see Section III-C. Here it is utilized only to explain the rule for cycle initiation.

Consider the following example. Suppose we have $n = 8$, $k = 4$ and the initial situation is as follows:

P_1	P_2	P_3	P_4
63	79	84	66
54	64	75	65

The execution of the algorithm is traced in Fig. 2. For each cycle we give the sequence of (processor, element) pairs in that cycle, the element determined to be the next in the merged list (i.e., the output list), and the contents of the working stack. The algorithm is initiated by processor P_1 and we assume that access right is given to the processor with the smallest identification number among those that want to talk. The termination of a cycle is detected by an empty time slot. Broadcast elements are pushed into the work stack as they are heard. The Max element is popped from the work stack and joins the output list. The next cycle is then initiated by the top element.

In the first cycle, 84 is determined to be the Max element. It is removed from the list of processor P_3, the highest element of which then becomes 75. The second cycle is initiated by processor P_2 since it broadcast immediately before P_3 in cycle 1, and so on. The mechanism of a work stack suggests that rebroadcasting the element that initiates a cycle by a processor could be avoided altogether, since the processors already heard that value and they can memorize it. Indeed,

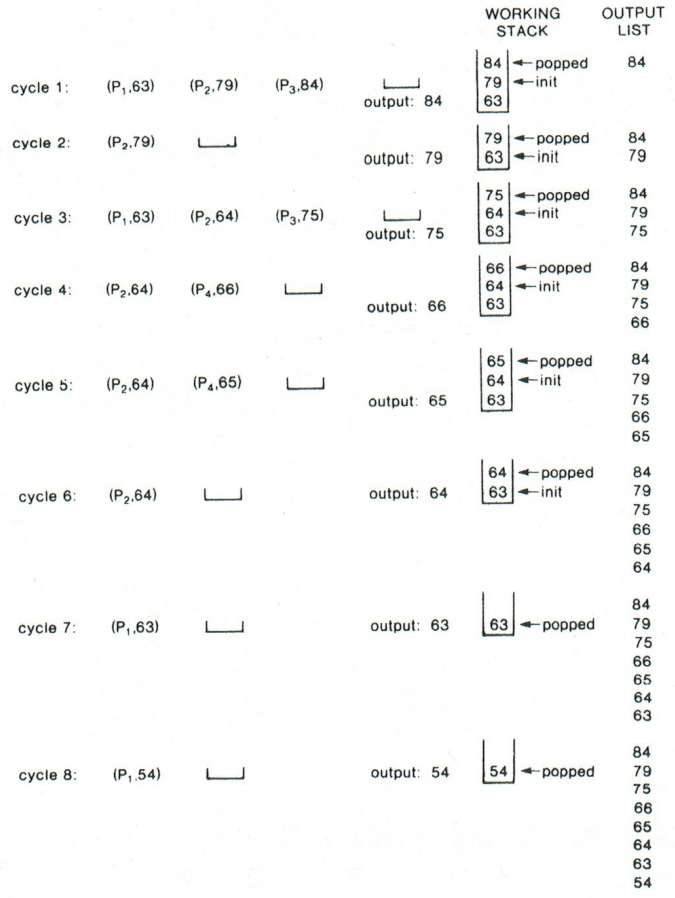


Fig. 2. The execution of Merge-algorithm on an example problem.

this is the basis of the improved Merge-algorithm to be described later.

B. Correctness and Complexity Analysis

The correctness of the algorithm follows immediately from the following three facts.

1) The first element in each list is the largest in that list at all times.

2) In each cycle the maximum of all the first elements is determined.

3) The determined maximum is removed from its list and added to the output list.

In the complexity analysis we calculate only the number of broadcasts performed, since each broadcast is followed by a comparison stage. We show that the worst case complexity is $2n - 1$. In order to prove this, we consider the following two lemmas.

Lemma 1: Let CV_i denote the current value of processor P_i. Whenever CV_i is rebroadcast after the first time, it initiates a cycle.

Proof: Assume to the contrary that the claim is not correct. There is, therefore, a situation in which a processor, that had already broadcast its current value, later hears a smaller value on the channel. The processor, in response, will rebroadcast its current value. Also, any current value that was broadcast must have a successor value, and that successor is larger than itself. Consider the first cycle in which this situation occurs and let CV_i be the largest among the already-heard current values at the time of this cycle that hears a smaller value on the channel (case 1 in the algorithm). Let CV_j be the successor of CV_i at that time (any current value that was broadcast must have a successor). This successor is larger then CV_i and, therefore, it is also larger than the value on the channel. However, since we picked CV_i to be the largest one with this property, we get a contradiction. □

It follows that from the time an element is first broadcast until it is merged, only values greater than or equal to itself can be broadcast. Let $\#V_i$ be the number of times element V_i is broadcast. The lemma implies that if $\#V_i > 1$ then V_i initiated at least $\#V_i - 1$ cycles. From Lemma 1 we can conclude also that there is at most one predecessor to each current value. The reason is that a value is determined as a successor only during its first transmission (it cannot be the initiator of that cycle). Therefore, it will be a successor to only one current value.

Lemma 2: Each time V_i initiates a cycle (except for the last cycle which includes only V_i), the cycle is terminated by an element that has never been broadcast before.

Proof: Assume $\#V_i > 2$. We arbitrarily choose the cycle which is initiated by the mth broadcast of V_i ($1 < m < \#V_i$). Let the cycle be terminated by an element denoted u_i^m. If u_i^m participated in an earlier cycle, it must initiate all other cycles in which it participates (according to Lemma 1), which leads to a contradiction. □

This lemma implies that with each broadcast of V_i, excluding the first and the last, we can associate a distinct element which is broadcast just once. This element is removed immediately after it is broadcast. Since no two distinct elements initiate the same cycle, we can partition the set $\{V_1, \cdots, V_n\}$ of all elements into disjoint subsets $M = \{S_1, S_2, \cdots, S_t\}$ such that each subset S_i either consists of a single element, which is broadcast once or twice, or S_i consists of m elements, one of which V_j is broadcast $m + 1$ times and the other $m - 1$ elements are those which terminate each of the $m - 1$ cycles initiated by V_j. Thus:

1) $\forall i, j \; S_i \cap S_j = \phi$

2) if $|S_i| = m$, then the total number of broadcasts, $B(S_i)$, by elements in S_i satisfies $B(S_i) \leq 2m$.

This leads to the following theorem.

Theorem 1: Let $T(n, k)$ be the number of broadcasts performed by the Merge-algorithm. $T(n, k)$ satisfies

$$n \leq T(n, k) \leq 2n - 1 . \tag{6}$$

Proof: It is obvious that $n \leq T(n, k)$ since each element has to be broadcast at least once. Also

$$T(n, k) \leq \sum_{S_i \in M} B(S_i) \leq \sum_{S_i \in M} 2|S_i| = 2n \tag{7}$$

since the element which terminates the first cycle is broadcast just once. We are left with $n - 1$ elements for which we have shown in (7) that the upper bound for their total broadcast time is $2(n - 1)$, which yields

$$T(n, K) \leq 2(n - 1) + 1 = 2n - 1 . \qquad \square$$

C. An Improved Merge-Algorithm

As mentioned earlier, it is possible to decrease the number of broadcasts required by making the initiation and termination of a cycle more sophisticated. However, these savings require a larger local memory for each processor.

In the modified version, each processor stores all the elements that were broadcast in a stack called wstack (as we did in the example). The initiation of cycles by elements that were broadcast before will be avoided altogether, since this information exists in the **wstack** of every processor. Each cycle now begins by broadcasting the second element relative to this cycle in the original algorithm, and cycles with one element will now reduce to empty cycles.

At the beginning of a cycle each processor compares the value at the head of its list to the top element in the wstack and decides to broadcast only if its value is larger. The rest of the cycle proceeds in the same manner as in the previous algorithm, where each processor pushes values onto its wstack as it hears them. At the end of a cycle (determined by an empty slot), each processor pops the top element from its wstack. Any consecutive empty slot following the first corresponds to an empty cycle (a cycle of 1 element in the previous algorithm). For each such empty, slot a processor pops its wstack and this value joins the merged list. When a wstack is empty but its list is not, the processor knows that the initiation of a cycle by a value which was never broadcast before is called for.

The number of broadcasts required by this algorithm is exactly n. Each element is broadcast just once. The number of comparison stages, however, remains the same as before. Note that the new algorithm requires that each processor maintain a stack of size $O(k)$, thus presenting a tradeoff between the number of broadcasts and the size of local memory.

In the rest of this paper, whenever we talk about the "Merge-by-broadcast" algorithm, we mean the first version, unless otherwise specified.

D. Sorting algorithms

The Merge-by-broadcast algorithms can be used to sort n elements with k processors with IPABM by initially having each processor sort its own list, using some efficient sequential algorithm (such as quicksort or sequential Merge sort [1]). The merging phase is performed by our Merge-by-broadcast algorithm. Let us call this sorting algorithm Broad-Sort(n, k).

Theorem 2: The complexity of Broad-Sort(n, k) is given by:

$$T(\text{Broad-Sort}(n,k)) = \left(\frac{n}{k}\log\frac{n}{k} + 2n - 1\right)t + (2n-1)T = O\left(\frac{n}{k}\log\frac{n}{k} + n\right). \quad (8)$$

Proof: Here we use the combined measure of performance that accounts for both comparison time and communication. The theorem is proved as follows:

$(n/k \log n/k)$ is the number of comparisons required to sort each list.

$(2n - 1)$ is the number of comparisons for merging.

$(2n - 1)$ is the number of broadcasts. □

The maximum number of comparisons required for sorting a sequence of n elements on a sequential processor is asymptotically $n \log n$. Therefore, when k is smaller than $\log n$ the asymptotic speedup ratio of the optimal sequential algorithm over the above algorithm is k, which is optimal. However, when k is greater than $\log n$, the total execution time required is asymptotically linear in n. Formally, the speedup ratio between sequential sorting and this Merge-Sort algorithm is greater than

$$\frac{k}{1 + \dfrac{k}{\log n}}.$$

When the improved Merge-algorithm is used to sort n elements with n processors ($k = n$), then we get the algorithm described by Levitan [15]. This algorithm uses exactly n broadcasts and $2n$ comparison stages. In this case (when $k = n$), the number of comparison stages can be reduced to n by having the elements which were not transmitted yet keep a pointer to their relative order in the wstack. Each processor can do that with no extra cost since they listen to the channel anyway. In that case, when a new cycle begins with the top element in the wstack, the elements know their relation to it and they do not need to make the comparison.

This argument cannot be extended to the Merge-algorithm since the elements that are updated to be the new current values were not compared to the values that were broadcast already.

Another way to sort n elements with n processors is as follows. Each processor broadcasts its value in a prespecified order. Each processor listens to the channel and remembers the value of its immediate successor in the list. After n broadcasts, all processors are linked in the order of their values. In the next phase, the processors will broadcast their values again according to the order dictated by the linked list. The first will be the Max value (the one with no successors). This algorithm uses $2n$ broadcasts and n comparison stages. Its advantage over Levitan's algorithm is that there is no contention on the channel and, therefore, in the RPABM model (to be discussed later), no extra cost will have to be paid for accessing the channel. This algorithm cannot be extended to a Merge-algorithm (at least not in a straightforward way) without increasing the number of comparison stages significantly.

IV. Optimality of the Merge-Algorithm

In this section we establish a lower bound on the performance of all Merge-by-broadcast algorithms when the only criterion is the number of broadcasts performed. Consequently, the above Merge-algorithms are shown to be optimal since they meet this bound.

We consider all possible Merge-algorithms using the IPABM to merge n distinct elements drawn from a set S on which an order is defined. The n elements are grouped into k sorted lists each with n/k elements. There are k processors, each containing one of the sorted lists. The output is obtained

in an independent processor (the output processor). We claim that any algorithm that merges the lists requires at least n broadcasts. More specifically, it requires that each of the elements will be broadcast at least once.

This claim might seem trivial since for the output processor to create the merged list it must hear all the values! However, if we reformulate the requirement such that the output processor need not know the actual values but simply their order, the claim is less obvious. Formally, let V_{ij} be the jth element in processor P_i (or the jth element in the ith sorted list) and v_{ij} be the value of the element in this location for a given input of n elements. The output processor should be able to give, for each input, a series of locations $V_{i_1j_1}, V_{i_2j_2}, \cdots, V_{i_rj_r}, \cdots, V_{i_nj_n}$ such that the sequence of values $v_{i_1j_1}, v_{i_2j_2}, \cdots, v_{i_rj_r}, \cdots, V_{i_nj_n}$ is the final merged list. From what we will show it follows that the values themselves are also available at the output processor. First we prove our claim for the special case of n lists having 1 element each.

Lemma 3: To merge n lists of 1 element each with IPABM and using n processors, each of the elements must be broadcast.

Proof (sketch): Any two elements which are adjacent in the sorted list must be compared directly. Let a_i, a_{i+1} be two consecutive elements. The order between these two elements and the rest is exactly the same, thus, to determine their internal order they must be compared directly. Since a_i and a_{i+1} are located in different processors and the outcome of the comparison must be available at the output processor which does not know them initially, both values must be broadcast. Each one of the processors, including the output processor, can then compare and determine the order between the two elements. It might be argued that it is sufficient to have one processor broadcast its value and the other only indicate whether it is larger or smaller; however, we count all messages in the same way and since the broadcast of the value gives more information we assume the values themselves are transmitted. We can conclude that since there are $n - 1$ adjacent pairs, $n - 1$ comparisons are required, each corresponding to two broadcasts. Since $n - 2$ of the elements participate in two adjacent pairs the number of broadcasts required is at least

$$2(n - 1) - (n - 2) = n. \tag{9}$$

□

In the general case, each processor has n/k sorted elements. As argued earlier, any adjacent pair of elements must be compared. If two adjacent elements are in the same processor, it is not necessary to broadcast both elements in order to make a comparison (a processor can make the comparison and then just broadcast the result in some coding or broadcast only the larger value). However, it is possible to create an input for which no two adjacent elements are located in the same processor. Fig. 3 illustrates such a case. Let $a_1 > a_2, \cdots, > a_n$ be the sorted list. The list of elements in processor P_i $(1 \le i \le k)$ is $a_i, a_{i+k}, \cdots, a_{i+k\cdot(n/k-1)}$. Thus, any comparison between adjacent values requires that both of them be broadcast. This yields the following theorem.

Theorem 3: Any Merge by broadcast algorithm, using IPABM, with n elements and k lists (k processors), requires

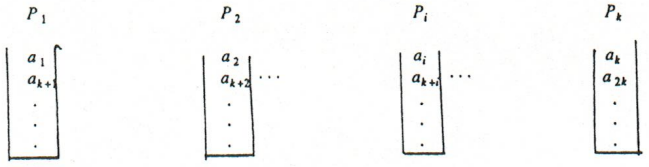


Fig. 3. A worst case example.

n broadcasts in the worst case if the output is accumulated in a separate output processor. □

It is easy to show that when the output processor is a processor which contains one of the lists, the lower bound decreases to $n - n/k$ broadcasts (the output processor need not broadcast its own values).

From Theorems 1 and 2, we conclude that the Merge-by-broadcast algorithm presented in Section III is asymptotically optimal w.r.t. number of broadcasts, while its modified version is absolutely optimal.

V. Access Scheme Considerations

So far in our discussion we have assumed the existence of an "ideal" access scheme that resolves all conflicts in constant time. Even if such an access scheme is not available, this assumption is appropriate when the algorithm itself is designed so that conflicts never arise (i.e., in each time slot at most one processor is enabled) as in the last sorting algorithm in Section III-D. If the algorithm is not designed in this way, conflicts between processors will generally arise and the access scheme used may have a profound effect on the complexity of the algorithm.

Numerous access schemes have been proposed and analyzed [6],[10]–[13], [22] in the context of broadcast communications. The measures that are used for evaluating their performance are those of channel capacity, throughput and delay. Of most interest to us is capacity; in particular, we are interested in the ratio between the time the channel is used for conflict resolution and the time it is used for "useful" communication.

We now modify our ideal model to include these access considerations. Let RPABM be an IPABM with the following changes: the channel is slotted into two types of time slots, one of length T for message transmission and the other one of length τ. The RPABM has a conflict resolution protocol, CRP, which substitutes for the global access mechanism in IPABM. The CRP is invoked whenever a conflict arises and it uses a τ-slotted channel. A T slot carrying a transmission marks the termination of CRP. Usually, $\tau << T$.

The communication time of an algorithm with RPABM now includes the number of T slots required for real transmissions and the number of τ slots used by the CRP. Let $\#T, \#\tau$, be the maximum number of T slots and the maximum number of τ slots, respectively, used by algorithm A for broadcasts and for resolving conflicts given a specific CRP. The worst case communication time for algorithm A with CRP, denoted by Comm(A, CRP), is defined as

$$\text{Comm}(A, \text{CRP}) = \#T \cdot T + \#\tau \cdot \tau. \tag{10}$$

Since $\tau << T$, the contribution of the second term (the cost of accessing) might be negligible for some specific parame-

ters of the problem. However, for the asymptotic complexity, τ cannot be ignored. The ratio $\#\tau/\#T$ characterizes the impact of the access scheme used on the asymptotic complexity of the algorithm. Specifically, if $\#\tau/\#T = O(f(k))$ where $f(k)$ is a nondecreasing function of k (the number of processors), then it is easy to see that

$$\text{Comm}(A, \text{CRP}) = O(\#T \cdot (f(k) + 1)). \quad (11)$$

In the next two subsections we present the "Merge-by-broadcast" algorithm using two realistic access schemes: MSAP (mini-slot alternating priority) [11] and the Tree-algorithm [6]. Following that, we give some lower bounds for CRP performance.

A. Merge with MSAP

In the MSAP scheme, processors obtain permission to broadcast according to some predetermined priority order among them which is dictated, for example, by their id's. Processors may broadcast one after the other in a round robin fashion, and when a processor has nothing to say, its turn is passed to the next in order after an empty τ slot. When it does broadcast it uses a T slot, and then gives the turn to the next processor. This is similar to a token ring.

This method is very appealing for the Merge algorithm since we have cycles built into it in which each processor might want to transmit once. Thus, integrating the access scheme into the Merge algorithm is straightforward: processors are ordered in increasing order of their id's. Any cycle of the algorithm is initiated by its initiator if the algorithm determines one. Otherwise, P_1 is the initiator. If P_i is an initiator of a cycle then the next one that can talk is P_{i+1}, and then P_{i+2}, and so on, until the end of the cycle. A processor determines when its turn comes by detecting the end of a cycle, by knowing the initiator's id and by counting the number of τ and T slots that occurred.

In Fig. 4 we show how the algorithm works with MSAP using the same input as in Fig. 2. Note that we do not need an extra time slot to determine the end of a cycle with this CRP. The empty slots are of length τ and are denoted by "␣," the full slots are of length T. We next determine the communication complexity of the Merge-algorithm with the MSAP CRP.

Theorem 4: For any instance I of Merge(n, k) with RPABM when CRP is MSAP, the number of τ slots used $\#\tau$ satisfies

$$\frac{n \cdot (k-1)}{2} \le \#\tau \le n \cdot (k-1). \quad (12)$$

Proof: Since the number of cycles is n and in each cycle we can have at most $k-1$ empty τ slots (when there is just one transmission, for example) there are at most $n \cdot (k-1)$ τ slots, i.e.:

$$\#\tau \le n \cdot (k-1).$$

We now determine the lower bound. An element is added to the Merged list when it terminates a cycle. To determine if a cycle is terminated by an element of processor P_i, $k-i$ empty τ slots must pass by. Since there are n/k elements in

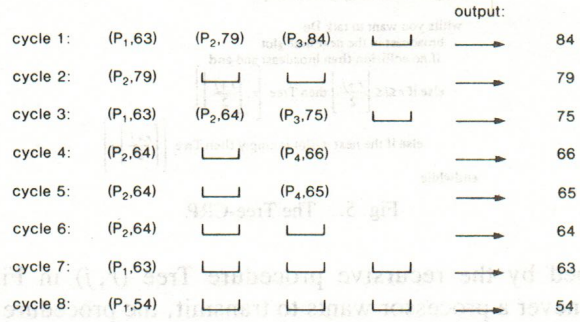


Fig. 4. Example of Merge with MSAP.

processor i and each terminates a cycle once, we have

$$\#\tau \ge \frac{n}{k} \sum_{i=1}^{k} (k-i) = \frac{n}{k} \sum_{i=0}^{k-1} i = \frac{n \cdot (k-1)}{2}. \quad (13)$$

□

We can conclude that using MSAP we have $\#\tau = \Theta(n \cdot k)$. Altogether

$$\text{Comm}(\text{Merge}(n, k), \text{MSAP}) = (2n-1) \cdot T + (n \cdot k) \cdot \tau = \Theta(n \cdot k). \quad (14)$$

Note that

$$\frac{\#\tau}{\#T} = O(k). \quad (15)$$

Thus, the asymptotic complexity increased by a factor of k; this is substantial when k is not a constant but rather a function of n.

B. Merge with the Tree-Algorithm

To apply the MSAP access scheme to the Merge-algorithm, we took advantage of the structure of the algorithm and its decomposition into cycles. The access scheme resolves conflicts for each cycle of transmissions and not for each transmission independently.

Our approach in the following scheme is to consider each conflict independently without acquiring information from previous conflicts or previous steps of the algorithm. Whenever a conflict occurs, the CRP is invoked and gives the right to transmit to one of the involved processors. The question we want to address is: Given k processors that want to transmit, how many time slots (for conflict resolution) are needed until one of the processors succeeds in broadcasting (this is essentially the well-known election problem in distributed computing). Greenberg [10] addresses similiar questions; however, none of them is identical to our question and therefore, none of his results are the same. For instance, he considers the problem of having k processors that want to talk in the same slot, and provides a probabilistic protocol which, on the average, enables all the k messages to be posted in time $O(k)$. In our case however, the situation changes after each successful broadcast, namely, if a processor wanted to talk and another processor was chosen to broadcast, it may not want to talk after it heard the broadcast message.

We apply Capetanakis' tree algorithm [6] to resolve conflicts using, again, the processor id's. The Tree-CRP is de-

```
Tree(r,j) /*executed by processor P_i*/
  while you want to talk Do
    broadcast in the next time slot
    if no collision then broadcast and end
    else if r ≤ i ≤ ⌊(r+j)/2⌋ then Tree [r, ⌊(r+j)/2⌋]
         else if the next τ-slot is empty then Tree [⌈(r+j)/2⌉, j]
  endwhile
```

Fig. 5. The Tree-CRP.

scribed by the recursive procedure Tree (r, j) in Fig. 5. Whenever a processor wants to transmit, the procedure Tree $(1, k)$ is invoked where k is the number of processors.

All processors try to broadcast in the first slot. If there is a collision, only those with indentification numbers less than $k/2$ try to transmit again. Another collision enables only those processors with id#'s $\leq k/4$ to transmit, and so on, until a successful transmission or an empty slot occurs. The latter event activates another subset of processors to keep trying in the same manner. The CRP uses a sequence of τ slots which are either collision slots or empty slots, and terminates by a successful transmission, i.e., by a T slot.

Resolving one conflict using the Tree-algorithm may require $2 \cdot \log k$ slots in the worst case. The scheme can be easily modified to take only $\log k\ \tau$ slots by noticing that an empty slot implies a collision in the subsequent slot and can therefore be skipped.

Theorem 5: The number of τ slots required by the Merge(n, k) algorithm in its two versions, with RPABM when the Tree-CRP is used, satisfies $\#\tau \leq n \cdot \log k$.

Proof: The number of broadcasts in the improved algorithm is n. In the first version of the algorithm there are $2n$ broadcasts, but only the first broadcast of each value may be involved in conflicts on the channel. In subsequent broadcasts the value initiates a cycle in which case it is the only one to access the channel. Therefore, for both versions of the Merge-algorithm there are n broadcasts that may be involved in a conflict. Since $\log k\ \tau$ slots are used for solving the conflict, the claim follows. □

We conclude that using the Tree-CRP and the first version of the Merge-algorithm, the communication time is as follows:

$$\text{Comm}(\text{Merge}(n,k), \text{Tree}-\text{CRP}) =$$

$$(2n - 1) \cdot T + n \cdot \log k \cdot \tau = O(n \log k). \tag{16}$$

In this case,

$$\frac{\#\tau}{\#T} = O(\log k). \tag{17}$$

We now show that any algorithm for conflict resolution can do no better than the Tree-CRP. First, we introduce some formalism. Let a conflict resolution protocol, CRP, for a set of k processors $\{1, \cdots, k\}$ be a function from the power set of $\{1, \cdots, k\}$ to the set $\{1, \cdots, k\}$. The CRP determines, for any subset of conflicting processors, one processor that can talk.

The execution of CRP for any subset of processors is over the τ slotted channel where each slot is either an empty slot or a conflict slot. The last slot is of length T. The sequence of empty slots and conflict slots up to but not including the T slot could be considered the encoded information by which CRP selects a specific processor. Note that a processor does not know which subset is currently being worked on by the CRP, but only whether or not it belongs to this subset. Let $C(S, x)$ be the binary code (i.e., the sequence of empty and conflict slots) which result when all processors in a subset of processors S want to transmit, and x is the first which succeeds. Two properties are required from any CRP.

1) If $\text{CRP}(S) = x$ where S is a subset of $\{1, 2, \cdots, k\}$ then $x \in S$.

2) If S_1, S_2 are any two subsets such that $x \in S_1 \cap S_2$ and if

$$\text{CRP}(S_1) = x$$

and

$$\text{CRP}(S_2) = y \neq x$$

then

$$C(S_1, x) \neq C(S_2, y).$$

Let $l(\text{CRP})$ be the maximum code length of a CRP.

Theorem 6: For every CRP defined over a set of k elements, $l(\text{CRP}) \geq \log k$.

Proof: For any given CRP we create a special family of subsets of processors

$$\text{CORE}(\text{CRP}) = \{S_1, S_2, \cdots, S_k\}$$

in the following recursive way:

$$S_1 = \{1, \cdots, k\}$$

$$S_{i+1} = S_i - \{x_i\}$$

where

$$x_i = \text{CRP}(S_i).$$

Obviously,

$$S_k \subset S_{k-1} \subset \cdots S_3 \subset S_2 \subset S_1$$

also,

$$\text{CRP}(S_i) \neq \text{CRP}(S_j).$$

We now show that the code sequences for $\text{CRP}(S_i)$ are all different. That is:

$$\forall i, j\ C(S_i, x_i) \neq C(S_j, x_j).$$

Suppose this is not true and for some i, j

$$C(S_i, x_i) = C(S_j, x_j)$$

where $j > i$. Then

$$S_i \supset S_j \Rightarrow x_j \in S_i.$$

Thus, $x_j \in S_i \cap S_j$ with $\text{CRP}(S_j) = x_j$ and $\text{CRP}(S_i) = x_i \neq x_j$, but $C(S_i, x_i) = C(S_j, x_j)$ which contradicts property 2 of the CRP. This proves that every CRP must create at least k different binary codes which is known to require $\log k$ binary slots. □

We see that any deterministic CRP does add a complexity factor of $\log k$ for every broadcast that enables more than one

processor. It can be shown that even when we limit the size of conflicts to only two out of the k processors, the worst case complexity of any CRP is still $\log k$. The argument goes as follows. In the case of a conflict, a CRP assigns a subset of the processors to talk in the first time slot and the other to be silent. If there is exactly one processor talking, the conflict is resolved and the CRP stops. Otherwise, in the case of a collision, a subset of the colliding processors can be chosen for the second time slot. If there was silence in the first slot, a subset of the remaining processors will be selected for the second time slot, etc. In the worst case, the two colliding processors can always be in the subset which is larger in each step of division, therefore, only after $\log k$ steps (there are k processors) the CRP will stop. This argument provides a different proof to Theorem 6 as well. This suggests approaching the design of broadcast algorithms in a way that minimizes the number of broadcasts that can result in conflict or not to allow conflicts at all.

VI. Conclusion

In this paper, we addressed issues of design and complexity involved in incorporating "broadcast communication" into distributed algorithms. We presented algorithms for merging k lists of n/k elements each by k processors and proved the complexity to be $O(n)$, regardless of the number of lists (processors). We also showed that this performance is optimal under the scheme of one-channel broadcast.

We initially avoided the effect of conflicts which exist in this mode of communication by introducing the algorithm in an ideal environment in which we pay no penalty for accessing the channel. We then showed that the problem of accessing the channel adds a factor of at least $\log k$ to the algorithm's performance. This suggests a need to investigate whether a different approach; i.e., minimizing the number of conflicts while designing the algorithm, might result in a better total performance. We also note that the use of a single channel limits the performance considerably (for example, merging cannot be accomplished in less than n sequential time slots) which motivates the use of more complex configurations of broadcast with more than one channel. For recent work on broadcast networks with multiple channels see [14] and [17].

Acknowledgment

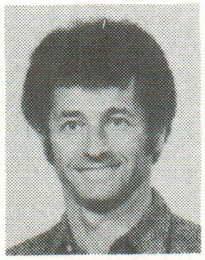
We would like to thank E. Gafni for many stimulating discussions and helpful remarks, and J. Marberg for many comments and a thorough review of the manuscript.

References

[1] Aho, Hoipcroft, and Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison Wesley, 1974.
[2] K. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.
[3] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers," *IEEE Trans. Comput.*, vol. C-27, Jan. 1978.
[4] S. H. Bokhari, "Max: An algorithm for finding maximum," in *Proc. 1981 Conf. Parallel Processing*, 1981, pp. 302–303.
[5] A. Borodin and J. H. Hopcroft, "Routing, merging, and sorting in parallel models of computations," in *Proc. 14th ACM Symp. Theory Comput.*, 1982.
[6] J. Capetanakis, "Generalized TDMA: The multiaccessing tree protocol," *IEEE Trans. Commun.*, vol. COM-27, pp. 1476–1484, Oct. 1979.
[7] S. Cook and C. Dwark, "Bounds on the time for parallel RAM's to compute simple functions," *14th ACM Symp. Theory Comput.*, 1982.
[8] R. Dechter, and L. Kleinrock, "Parallel algorithms for multiprocessors using broadcast channel," Dep. Comput. Sci., Univ. California, Los Angeles, CA Tech. Rep. 850025, 1981.
[9] D. Ditton, D. J. Dewitt, D. K. Hsiao, and J. Menon, "A taxonomy of parallel sorting," *Comput. Surveys*, vol. 16, no. 3, pp. 287–318, 1984.
[10] A. G. Greenberg, "On the time complexity of broadcast communication schemes," in *Proc. 14th ACM Symp. Theory Comput.*, 1982.
[11] L. Kleinrock and M. O. Scholl, "Packet Switching in radio channels: New conflict-free multiple access schemes for a small number of data users," *IEEE Trans. Commun.*, vol. COM-28, pp. 1015–1029, 1980.
[12] L. Kleinrock, *Queuing Systems*, vol. 2. New York: Wiley, 1976.
[13] S. S. Lam, "A carier sense multiple access protocol for local networks," *Comput. Networks*, vol. 4, pp. 21–32, 1980.
[14] G. M. Landau, M. M. Yung, and Z. Galil, "Distributed algorithms in synchronous broadcasting networks," in *Proc. 12th Int. Conf. Automata Languages, Programming*, Nafplion, Greece, 1985, pp. 363–372.
[15] S. Levitan "Algorithms for broadcast protocol multiprocessor," in *Proc. 3rd Int. Conf. Distributed Comput. Syst.*, 1982, pp. 666–671.
[16] S. P. Levitan and C. C. Foster, "Finding an extermum in a network," in *Proc. 9th Ann. Int. Symp. Comput. Architect.*, Austin, TX, 1982.
[17] J. Marberg, and E. Gafni, "Sorting and selection in multichannel broadcast networks," in *Proc. 1985 Int. Conf. Parallel Processing*, 1985, pp. 846–850.
[18] R. Metcalfe and D. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. Ass. Comput. Mach.*, vol. 19, no. 17, pp. 395–404, 1976.
[19] F. P. Preparata, "New parallel sorting schemes,"*IEEE Trans. Comput.*, vol C-27, pp. 669–673, 1978.
[20] M. Snir, "On parallel search," in *Proc. ACM Symp. Distributed Algorithms*, 1982.
[21] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, no. 4, pp. 153–161, 1972.
[22] B. W. Stuck, and E. Arthurs, *A Computer Communications Network Performance Analysis Primer*. Englewood Cliffs, NJ: Prentice Hall, 1985.
[23] C. D. Thompson and H. T. Kung, "Sorting on a mesh connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, no. 4, pp. 263–271, 1977.
[24] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, pp. 348–355, 1975.

Rina Dechter was born in Natania, Israel, in August 1950. She received the B.S. degree in mathematics from the Hebrew University, Jerusalem, Israel in 1973, the M.S. degree in applied mathematics from the Weizmann Institute, Rehovot, Israel in 1975, and the Ph.D degree in computer science from the University of California, Los Angeles in 1985.

During the years 1975–1978 she was a Staff Member at the Everyman's University, Tel-Aviv, Israel, developing self-study programs for teaching college mathematics. From 1978 to 1980 she was with Perceptroncs Inc., Woodland Hills, CA, responsible for modeling, experimental design, and computer simulations for human-resource test and evaluation for advanced weapon systems. She is currently at the Cognitive System Laboratory in the Computer-System Department at the University of California doing research in the area of artifical intelligence.



Leonard Kleinrock (S'55–M'64–SM'71–F'73) received the B.S. degree in electrical engineering from the City College of New York in 1957 and the M.S.E.E. and Ph.D.E.E. degrees from the Massachusetts Institute of Technology, Cambridge, in 1959 and 1963, respectively.

While at M.I.T., he worked at the Research Laboratory for Electronics as well as with the computer research group at Lincoln Laboratory in Advanced Technology. He joined the faculty at UCLA in 1963. His research interests focus on computer networks,

packet radio systems, and local area networks. He has had over 120 papers published and is the author of three books, *Communication Nets: Stochastic Message Flow and Delay*, 1964; *Queueing Systems, Volume I: Theory*, 1975; *Queueing Systems, Volume II: Computer Applications*, 1976, and also *Solutions Manual for Queueing Systems, Volume I*, 1982.

Professor Kleinrock served as the head of the University of California, Los Angeles, Computer Science Department Research Laboratory and is a well-known lecturer in the computer industry. He is Principal Investigator for the Advanced Research Projects Agency Advanced Teleprocessing Systems contract at UCLA and co-Principal Investigator for the National Science Foundation Advanced Network Environment for Distributed Systems Research Project. He was recently elected to the National Academy of Engineering, is a Guggenheim Fellow, and serves on the Boards of Governors of various advisory councils in the computer field. He is a member of the Science Advisory Committee for IBM. He has received numerous best paper and teaching awards, including the ICC 1978 Prize Winning Paper Award, the 1976 Lanchester Prize for outstanding work in Operations Research, the Communications Society 1975 Leonard G. Abraham Prize Paper Award. In 1982, as well as having been selected to receive the City College of New York Townsend Harris Medal, he was co-winner of the L. M. Ericsson Prize, presented by His Majesty, King Carl Gustaf of Sweden, for his outstanding contributions in packet switching technology.