Chapter 4. Basic Communication Operations

In most parallel algorithms, processes need to exchange data with other processes. This exchange of data can significantly impact the efficiency of parallel programs by introducing interaction delays during their execution. For instance, recall from **Section 2.5** that it takes roughly $t_s + mt_w$ time for a simple exchange of an m-word message between two processes running on different nodes of an interconnection network with cut-through routing. Here t_s is the latency or the startup time for the data transfer and t_w is the per-word transfer time, which is inversely proportional to the available bandwidth between the nodes. Many interactions in practical parallel programs occur in well-defined patterns involving more than two processes. Often either all processes participate together in a single global interaction operation, or subsets of processes participate in interactions local to each subset. These common basic patterns of interprocess interaction or communication are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of these basic communication operations on various parallel architectures is a key to the efficient execution of the parallel algorithms that use them.

In this chapter, we present algorithms to implement some commonly used communication patterns on simple interconnection networks, such as the linear array, two-dimensional mesh, and the hypercube. The choice of these interconnection networks is motivated primarily by pedagogical reasons. For instance, although it is unlikely that large scale parallel computers will be based on the linear array or ring topology, it is important to understand various communication operations in the context of linear arrays because the rows and columns of meshes are linear arrays. Parallel algorithms that perform rowwise or columnwise communication on meshes use linear array algorithms. The algorithms for a number of

communication operations on a mesh are simple extensions of the corresponding linear array algorithms to two dimensions. Furthermore, parallel algorithms using regular data structures such as arrays often map naturally onto one- or two-dimensional arrays of processes. This too makes it important to study interprocess interaction on a linear array or mesh interconnection network. The hypercube architecture, on the other hand, is interesting because many algorithms with recursive interaction patterns map naturally onto a hypercube topology. Most of these algorithms may perform equally well on interconnection networks other than the hypercube, but it is simpler to visualize their communication patterns on a hypercube.

Why hypercube

The algorithms presented in this chapter in the context of simple network topologies are practical and are highly suitable for modern parallel computers, even though most such computers are unlikely to have an interconnection network that exactly matches one of the networks considered in this chapter. The reason is that on a modern parallel computer, the time to transfer data of a certain size between two nodes is often independent of the relative location of the nodes in the interconnection network. This homogeneity is afforded by a variety of firmware and hardware features such as randomized routing algorithms and cut-through routing, etc. Furthermore, the end user usually does not have explicit control over mapping processes onto physical processors. Therefore, we assume that the transfer of m words of data between *any* pair of nodes in an interconnection network incurs a cost of $t_s + mt_w$. On most architectures, this assumption is reasonably accurate as long as a free link is available between the source and destination nodes for the data to traverse. However, if many pairs of nodes are communicating simultaneously, then the messages may take longer. This can happen if the number of messages passing through a cross-section of the network exceeds the cross-section bandwidth (**Section 2.4.4**) of the network. In such situations, we need to adjust the value of t_w to reflect the slowdown due to congestion. As discussed in **Section 2.5.1**, we refer to the adjusted value of t_w as effective t_w. We will make a note in the text when we come across

May be when we don't have contention

Another assumption

When links can be congested

communication operations that may cause congestion on certain networks.

As discussed in **Section 2.5.2**, the cost of data-sharing among processors in the shared-address-space paradigm can be modeled using the same expression $t_s + mt_w$, usually with different values of t_s and t_w relative to each other as well as relative to the computation speed of the processors of the parallel computer. Therefore, parallel algorithms requiring one or more of the interaction patterns discussed in this chapter can be assumed to incur costs whose expression is close to one derived in the context of message-passing.

In the following sections we describe various communication operations and derive expressions for their time complexity. We assume that the interconnection network supports cut-through routing (**Section 2.5.1**) and that the communication time between any pair of nodes is practically independent of of the number of intermediate nodes along the paths between them. We also assume that the communication links are bidirectional; that is, two directly-connected nodes can send messages of size m to each other simultaneously in time $t_s + t_w m$. We assume a single-port communication model, in which a node can send a message on only one of its links at a time. Similarly, it can receive a message on only one link at a time. However, a node can receive a message while sending another message at the same time on the same or a different link.

Many of the operations described here have duals and other related operations that we can perform by using procedures very similar to those for the original operations. The dual of a communication operation is the opposite of the original operation and can be performed by reversing the direction and sequence of messages in the original operation. We will mention such operations wherever applicable.

One-to-All Broadcast and All-to-One Reduction

Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as one-to-all broadcast. Initially, only the source process has the data of size m that needs to be broadcast. At the termination of the procedure, there are p copies of the initial data – one belonging to each process. The dual of one-to-all broadcast is all-to-one reduction. In an all-to-one reduction operation, each of the p participating processes starts with a buffer M containing m words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size m. Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers – the i th word of the accumulated M is the sum, product, maximum, or minimum of the i th words of each of the original buffers. **Figure 4.1** shows one-to-all broadcast and all-to-one reduction among p processes.

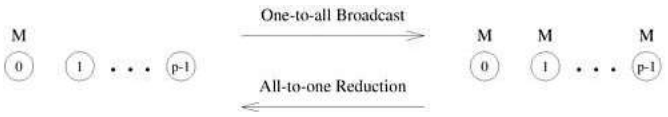


Figure 4.1. One-to-all broadcast and all-to-one reduction.

One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety of interconnection topologies.

Ring or Linear Array

A naive way to perform one-to-all broadcast is to sequentially send $p - 1$ messages from the source to the other $p - 1$ processes. However, this is inefficient because the source process becomes a bottleneck. Moreover, the communication network is underutilized because only the connection between a single pair of nodes is used at a time. A better broadcast algorithm can be devised using a technique commonly known as recursive doubling. The source process first sends the message to another process. Now both these processes can simultaneously send the message to two other processes that are still waiting for the message. By continuing this

procedure until all the processes have received the data, the message can be broadcast in $\log p$ steps.

The steps in a one-to-all broadcast on an eight-node linear array or ring are shown in **Figure 4.2**. The nodes are labeled from 0 to 7. Each message transmission step is shown by a numbered, dotted arrow from the source of the message to its destination. Arrows indicating messages sent during the same time step have the same number.

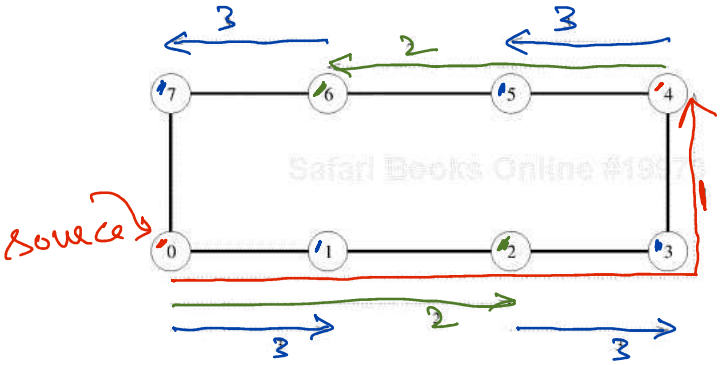
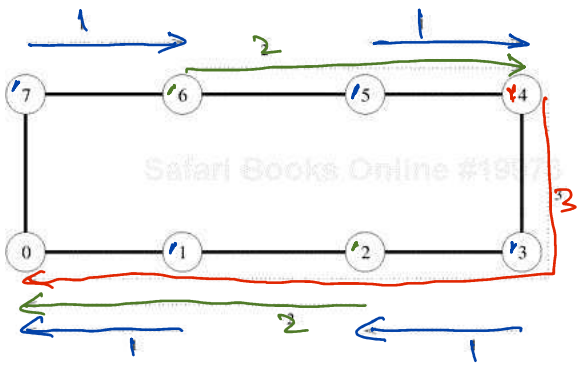


Figure 4.2. One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

Note that on a linear array, the destination node to which the message is sent in each step must be carefully chosen. In **Figure 4.2**, the message is first sent to the farthest node (4) from the source (0). In the second step, the distance between the sending and receiving nodes is halved, and so on. The message recipients are selected in this manner at each step to avoid congestion on the network. For example, if node 0 sent the message to node 1 in the first step and then nodes 0 and 1 attempted to send messages to nodes 2 and 3, respectively, in the second step, the link between nodes 1 and 2 would be congested as it would be a part of the shortest route for both the messages in the second step.

Reduction on a linear array can be performed by simply reversing the direction and the sequence of communication, as shown in **Figure 4.3**. In the first step, each odd numbered node sends its buffer to the even numbered node just before itself, where the contents of the two buffers are combined into one. After the first step, there are four buffers left to be reduced on nodes 0, 2, 4, and 6, respectively. In the second step, the contents of the buffers on nodes 0 and 2 are accumulated on node 0 and those on

nodes 6 and 4 are accumulated on node 4. Finally, node 4 sends its buffer to node 0, which computes the final result of the reduction.



1: 8 to 4 m after reduction
2: 4 to 2 m // //
3: 2 to 1 m // //

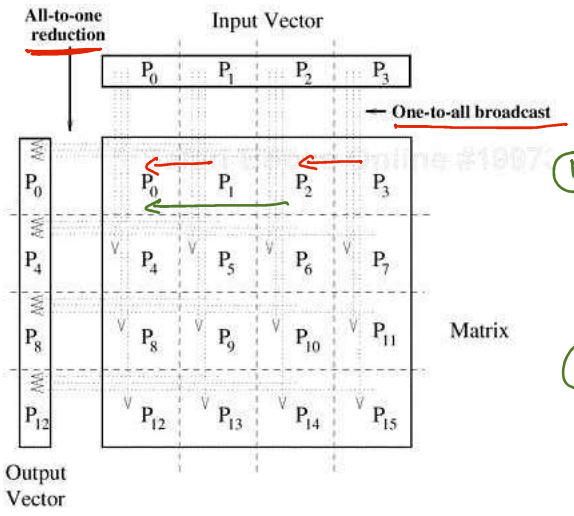
Compare with Fig 4.2

Figure 4.3. Reduction on an eight-node ring with node 0 as the destination of the reduction.

EXAMPLE 4.1 MATRIX-VECTOR MULTIPLICATION

Consider the problem of multiplying an $n \times n$ matrix A with an $n \times 1$ vector x on an $n \times n$ mesh of nodes to yield an $n \times 1$ result vector y. Algorithm 8.1 shows a serial algorithm for this problem. **Figure 4.4** shows one possible mapping of the matrix and the vectors in which each element of the matrix belongs to a different process, and the vector is distributed among the processes in the topmost row of the mesh and the result vector is generated on the leftmost column of processes.

Recall owner's compute rule!



In parallel:
① In $\log_2 4 = 2$ steps all processes have required data
② In 2 steps reduction happens

FIGURE 4.4. ONE-TO-ALL BROADCAST AND ALL-TO-ONE REDUCTION IN THE MULTIPLICATION OF A 4 × 4 MATRIX WITH A 4 × 1 VECTOR.

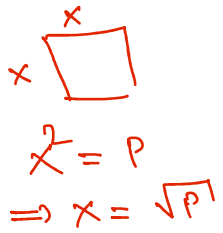
Since all the rows of the matrix must be multiplied with the vector, each process needs the element of the vector residing in the topmost process of its column. Hence, before computing the ma-

trix-vector product, each column of nodes performs a one-to-all broadcast of the vector elements with the topmost process of the column as the source. This is done by treating each column of the $n \times n$ mesh as an n-node linear array, and simultaneously applying the linear array broadcast procedure described previously to all columns.

After the broadcast, each process multiplies its matrix element with the result of the broadcast. Now, each row of processes needs to add its result to generate the corresponding element of the product vector. This is accomplished by performing all-to-one reduction on each row of the process mesh with the first process of each row as the destination of the reduction operation.

For example, P_9 will receive x [1] from P_1 as a result of the broadcast, will multiply it with A[2, 1] and will participate in an all-to-one reduction with P_8, P_{10}, and P_{11} to accumulate y[2] on P_8. ▪

Mesh



We can regard each row and column of a square mesh of p nodes as a linear array of $\sqrt{p}$ nodes. So a number of communication algorithms on the mesh are simple extensions of their linear array counterparts. A linear array communication operation can be performed in two phases on a mesh. In the first phase, the operation is performed along one or all rows by treating the rows as linear arrays. In the second phase, the columns are treated similarly.

Consider the problem of one-to-all broadcast on a two-dimensional square mesh with $\sqrt{p}$ rows and $\sqrt{p}$ columns. First, a one-to-all broadcast is performed from the source to the remaining $(\sqrt{p} - 1)$ nodes of the same row. Once all the nodes in a row of the mesh have acquired the data, they initiate a one-to-all broadcast in their respective columns. At the end of the second phase, every node in the mesh has a copy of the initial message. The communication steps for one-to-all broadcast on a mesh are illustrated in **Figure 4.5** for p = 16, with node 0 at the bottom-left corner as

the source. Steps 1 and 2 correspond to the first phase, and steps 3 and 4 correspond to the second phase.

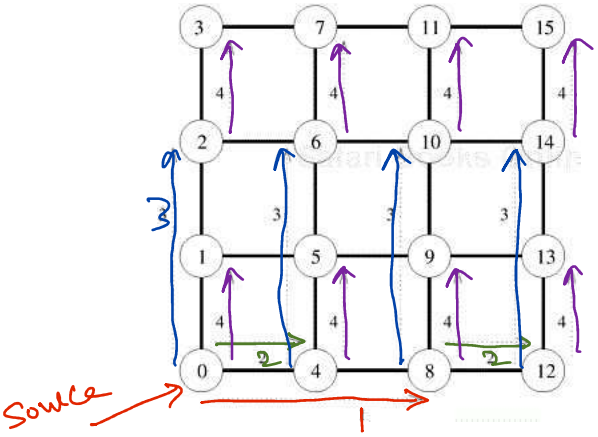


Figure 4.5. One-to-all broadcast on a 16-node mesh.

We can use a similar procedure for one-to-all broadcast on a three-dimensional mesh as well. In this case, rows of $p^{1/3}$ nodes in each of the three dimensions of the mesh would be treated as linear arrays. As in the case of a linear array, reduction can be performed on two- and three-dimensional meshes by simply reversing the direction and the order of messages.

Hypercube

The previous subsection showed that one-to-all broadcast is performed in two phases on a two-dimensional mesh, with the communication taking place along a different dimension in each phase. Similarly, the process is carried out in three phases on a three-dimensional mesh. A hypercube with 2^d nodes can be regarded as a d-dimensional mesh with two nodes in each dimension. Hence, the mesh algorithm can be extended to the hypercube, except that the process is now carried out in d steps – one in each dimension.

Figure 4.6 shows a one-to-all broadcast on an eight-node (three-dimensional) hypercube with node 0 as the source. In this figure, communication starts along the highest dimension (that is, the dimension specified by the most significant bit of the binary representation of a node label) and proceeds along successively lower dimensions in subsequent steps. Note that the source and the destination nodes in three communication

steps of the algorithm shown in Figure 4.6 are identical to the ones in the broadcast algorithm on a linear array shown in Figure 4.2. However, on a hypercube, the order in which the dimensions are chosen for communication does not affect the outcome of the procedure. Figure 4.6 shows only one such order. Unlike a linear array, the hypercube broadcast would not suffer from congestion if node 0 started out by sending the message to node 1 in the first step, followed by nodes 0 and 1 sending messages to nodes 2 and 3, respectively, and finally nodes 0, 1, 2, and 3 sending messages to nodes 4, 5, 6, and 7, respectively.

That is because a lot of links are available

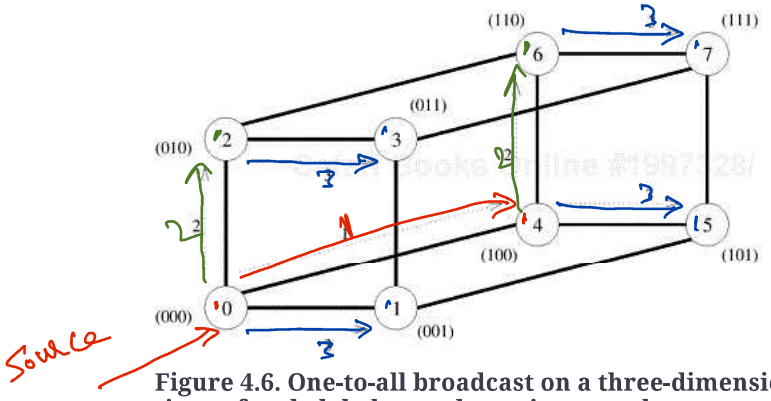


Figure 4.6. One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses.

Balanced Binary Tree

The hypercube algorithm for one-to-all broadcast maps naturally onto a balanced binary tree in which each leaf is a processing node and intermediate nodes serve only as switching units. This is illustrated in Figure 4.7 for eight nodes. In this figure, the communicating nodes have the same labels as in the hypercube algorithm illustrated in Figure 4.6. Figure 4.7 shows that there is no congestion on any of the communication links at any time. The difference between the communication on a hypercube and the tree shown in Figure 4.7 is that there is a different number of switching nodes along different paths on the tree.

Observations

① No link congestion in steps

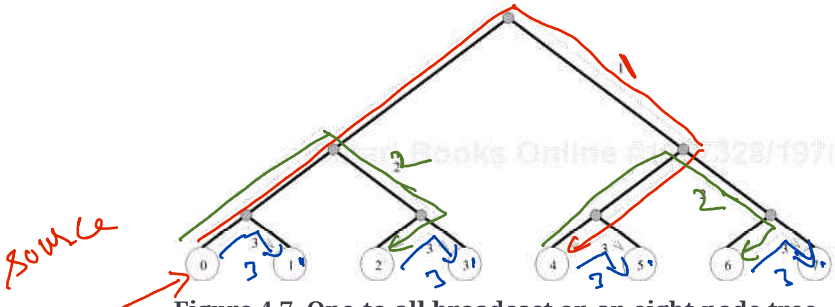


Figure 4.7. One-to-all broadcast on an eight-node tree.

Detailed Algorithms

A careful look at Figures 4.2, 4.5, 4.6, and 4.7 would reveal that the basic communication pattern for one-to-all broadcast is identical on all the four interconnection networks considered in this section. We now describe procedures to implement the broadcast and reduction operations. For the sake of simplicity, the algorithms are described here in the context of a hypercube and assume that the number of communicating processes is a power of 2. However, they apply to any network topology, and can be easily extended to work for any number of processes (Problem 4.1).

Algorithm 4.1 shows a one-to-all broadcast procedure on a 2^d-node network when node 0 is the source of the broadcast. The procedure is executed at all the nodes. At any node, the value of *my_id* is the label of that node. Let X be the message to be broadcast, which initially resides at the source node 0. The procedure performs d communication steps, one along each dimension of a hypothetical hypercube. In **Algorithm 4.1**, communication proceeds from the highest to the lowest dimension (although the order in which dimensions are chosen does not matter). The loop counter i indicates the current dimension of the hypercube in which communication is taking place. Only the nodes with zero in the i least significant bits of their labels participate in communication along dimension i. For instance, on the three-dimensional hypercube shown in **Figure 4.6**, i is equal to 2 in the first time step. Therefore, only nodes 0 and 4 communicate, since their two least significant bits are zero. In the next time step, when i = 1, all nodes (that is, 0, 2, 4, and 6) with zero in their least significant bits participate in communication. The procedure terminates after communication has taken place along all dimensions.

The variable *mask* helps determine which nodes communicate in a particular iteration of the loop. The variable *mask* has d (= log p) bits, all of which are initially set to one (Line 3). At the beginning of each iteration, the most significant nonzero bit of *mask* is reset to zero (Line 5). Line 6 determines which nodes communicate in the current iteration of the outer loop. For instance, for the hypercube of **Figure 4.6**, *mask* is initially set to 111, and it would be 011 during the iteration corresponding to i = 2

(the i least significant bits of *mask* are ones). The AND operation on Line 6 selects only those nodes that have zeros in their i least significant bits. Among the nodes selected for communication along dimension i, the nodes with a zero at bit position i send the data, and the nodes with a one at bit position i receive it. The test to determine the sending and receiving nodes is performed on Line 7. For example, in **Figure 4.6**, node 0 (000) is the sender and node 4 (100) is the receiver in the iteration corresponding to $i = 2$. Similarly, for $i = 1$, nodes 0 (000) and 4 (100) are senders while nodes 2 (010) and 6 (110) are receivers.

Algorithm 4.1 works only if node 0 is the source of the broadcast. For an arbitrary source, we must relabel the nodes of the hypothetical hypercube by XORing the label of each node with the label of the source node before we apply this procedure. A modified one-to-all broadcast procedure that works for any value of *source* between 0 and $p - 1$ is shown in **Algorithm 4.2**. By performing the XOR operation at Line 3, **Algorithm 4.2** relabels the source node to 0, and relabels the other nodes relative to the source. After this relabeling, the algorithm of **Algorithm 4.1** can be applied to perform the broadcast.

Algorithm 4.3 gives a procedure to perform an all-to-one reduction on a hypothetical d-dimensional hypercube such that the final result is accumulated on node 0. Single node-accumulation is the dual of one-to-all broadcast. Therefore, we obtain the communication pattern required to implement reduction by reversing the order and the direction of messages in one-to-all broadcast. Procedure ALL_TO_ONE_REDUCE(*d*, *my_id*, *m*, *X*, *sum*) shown in **Algorithm 4.3** is very similar to procedure ONE_TO_ALL_BC(*d*, *my_id*, *X*) shown in **Algorithm 4.1**. One difference is that the communication in all-to-one reduction proceeds from the lowest to the highest dimension. This change is reflected in the way that variables *mask* and i are manipulated in **Algorithm 4.3**. The criterion for determining the source and the destination among a pair of communicating nodes is also reversed (Line 7). Apart from these differences, procedure ALL_TO_ONE_REDUCE has extra instructions (Lines 13 and 14) to add the

contents of the messages received by a node in each iteration (any associative operation can be used in place of addition).

Example 4.1. One-to-all broadcast of a message *X* from node 0 of a *d*-dimensional *p*-node hypercube ($d = \log p$). AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

```
1.    procedure ONE_TO_ALL_BC(d, my_id, X)
2.    begin
3.       mask := 2^d - 1;                  /* Set all d bits of mask to 1 */
4.       for i := d - 1 downto 0 do        /* Outer loop */
5.           mask := mask XOR 2^i;         /* Set bit i of mask to 0 */
6.           if (my_id AND mask) = 0 then /* If lower i bits of my_id are 0
7.               if (my_id AND 2^i) = 0 then
8.                   msg_destination := my_id XOR 2^i;
9.                   send X to msg_destination;
10.              else
11.                  msg_source := my_id XOR 2^i;
12.                  receive X from msg_source;
13.              endelse;
14.          endif;
15.      endfor;
16.   end ONE_TO_ALL_BC
```

Example 4.2. One-to-all broadcast of a message *X* initiated by *source* on a *d*-dimensional hypothetical hypercube. The AND and XOR operations are bitwise logical operations.

```
1.    procedure GENERAL_ONE_TO_ALL_BC(d, my_id, source, X)
2.    begin
3.       my_virtual_id := my_id XOR source;
4.       mask := 2^d - 1;
5.       for i := d - 1 downto 0 do   /* Outer loop */
6.           mask := mask XOR 2^i;    /* Set bit i of mask to 0 */
7.           if (my_virtual_id AND mask) = 0 then
8.               if (my_virtual_id AND 2^i) = 0 then
9.                   virtual_dest := my_virtual_id XOR 2^i;
```

```
10.                 send X to (virtual_dest XOR source);
        /* Convert virtual_dest to the label of the physical destination */
11.             else
12.                 virtual_source := my_virtual_id XOR 2^i;
13.                 receive X from (virtual_source XOR source);
        /* Convert virtual_source to the label of the physical source */
14.             endelse;
15.     endfor;
16. end GENERAL_ONE_TO_ALL_BC
```

Example 4.3. Single-node accumulation on a *d*-dimensional hypercube. Each node contributes a message *X* containing *m* words, and node 0 is the destination of the sum. The AND and XOR operations are bitwise logical operations.

```
1.  procedure ALL_TO_ONE_REDUCE(d, my_id, m, X, sum)
2.  begin
3.     for j := 0 to m - 1 do sum[j] := X[j];
4.     mask := 0;
5.     for i := 0 to d - 1 do
           /* Select nodes whose lower i bits are 0 */
6.         if (my_id AND mask) = 0 then
7.             if (my_id AND 2^i) ≠ 0 then
8.                 msg_destination := my_id XOR 2^i;
9.                 send sum to msg_destination;
10.            else
11.                msg_source := my_id XOR 2^i;
12.                receive X from msg_source;
13.                for j := 0 to m - 1 do
14.                    sum[j] :=sum[j] + X[j];
15.            endelse;
16.         mask := mask XOR 2^i; /* Set bit i of mask to 1 */
17.    endfor;
18. end ALL_TO_ONE_REDUCE
```

Cost Analysis

Caution :-
Be careful about this Cost. Don't blindly apply!

Analyzing the cost of one-to-all broadcast and all-to-one reduction is fairly straightforward. Assume that p processes participate in the opera-

tion and the data to be broadcast or reduced contains m words. The broadcast or reduction procedure involves $\log p$ point-to-point simple message transfers, each at a time cost of $t_s + t_w m$. Therefore, the total time taken by the procedure is

Equation 4.1.

$$T = (t_s + t_w m) \log p.$$

All-to-All Broadcast and Reduction

All-to-all broadcast is a generalization of one-to-all broadcast in which all p nodes simultaneously initiate a broadcast. A process sends the same m-word message to every other process, but different processes may broadcast different messages. All-to-all broadcast is used in matrix operations, including matrix multiplication and matrix-vector multiplication. The dual of all-to-all broadcast is all-to-all reduction, in which every node is the destination of an all-to-one reduction (Problem 4.8). **Figure 4.8** illustrates all-to-all broadcast and all-to-all reduction.

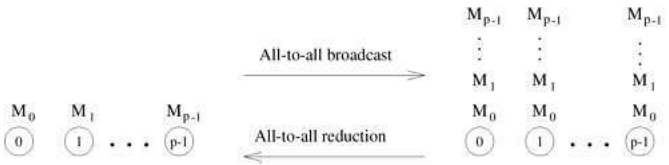


Figure 4.8. All-to-all broadcast and all-to-all reduction.

One way to perform an all-to-all broadcast is to perform p one-to-all broadcasts, one starting at each node. If performed naively, on some architectures this approach may take up to p times as long as a one-to-all broadcast. It is possible to use the communication links in the interconnection network more efficiently by performing all p one-to-all broadcasts simultaneously so that all messages traversing the same path at the same time are concatenated into a single message whose size is the sum of the sizes of individual messages.

The following sections describe all-to-all broadcast on linear array, mesh, and hypercube topologies.

Linear Array and Ring