COMP 633

Kumar et al.

Chapter 3 extracts

CHAPTER 3

Basic Communication Operations

In most parallel algorithms, processors need to exchange data. This exchange of data significantly affects the efficiency of parallel programs by introducing communication delays during their execution. There are a few common basic patterns of interprocessor communication that 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 efficient algorithms for some basic communication operations on the ring, two-dimensional mesh, and hypercube architectures. For pedagogical reasons we assume that the mesh is a square two-dimensional array of processors with end-to-end wraparound connections in both dimensions. The time taken for any of these operations increases at most by a factor of four in the absence of wraparound connections (Problem 3.14). Although it is unlikely that large scale parallel computers will be based on the ring topology, it is important to understand various communication operations in the context of rings because the rows and columns of wraparound meshes are rings. Parallel algorithms that perform rowwise or columnwise communication on wraparound meshes use ring algorithms. Furthermore, the algorithms for a number of communication operations on a mesh are simple extensions of the corresponding ring algorithms in two dimensions.

We describe the procedures to implement the basic communication operations for both store-and-forward (SF) and cut-through (CT) routing schemes (Section 2.7). We assume that the communication links are bidirectional; that is, two directly-connected processors can send messages of size m to each other simultaneously in time $t_s + t_w m$, where t_s is the message startup time and t_w is the per-word transfer time. We also assume that a processor 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 processor can receive a message while sending another message at the same time on the same or a different link.

In the following sections we describe various communication operations and derive expressions for their time complexity on a variety of parallel architectures. 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.

3.1 Simple Message Transfer between Two Processors

Sending a message from one processor to another is the most basic communication operation. Recall from Section 2.7.1 that, with SF routing, sending a single message containing m words takes $t_s + t_w ml$ time, where l is the number of links traversed by the message. On an ensemble of p processors, l is at most $\lfloor p/2 \rfloor$ for a ring, $2\lfloor \sqrt{p}/2 \rfloor$ for a wraparound square mesh, and $\log p$ for a hypercube (assuming that messages are sent on the shortest path between the source and the destination processors). Thus, with SF routing, the time for a single message transfer on ring, mesh, and hypercube has an upper bound of $t_s + t_w m \lfloor p/2 \rfloor$, $t_s + 2t_w m \lfloor \sqrt{p}/2 \rfloor$, and $t_s + t_w m \log p$, respectively. If CT routing is available, then a message can be sent directly from the source to a destination l links away in time $t_s + t_w m + t_h l$ (Section 2.7.2), where t_h is the per-hop time.

If the size m of the message is very small, the time for a single message transfer is similar in both SF and CT routing schemes. In both cases, it is the sum of a constant and a term proportional to the shortest distance between the processors. On the other hand, if the message is large (that is, $m \gg l$), then the distance between the processors becomes unimportant on a parallel computer using CT routing. For such messages, the time for a single message transfer between any two processors with CT routing is approximately the same as the message transfer time between directly-connected processors on an SF routing network.

3.2 One-to-All Broadcast

Parallel algorithms often require a single processor to send identical data to all other processors or to a subset of them. This operation is known as ***one-to-all broadcast*** or ***single-node broadcast***. Initially, only the source processor 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 residing at each processor. Figure 3.1 shows one-to-all broadcast of a message M among p processors. The related problem of ***k-to-all broadcast*** is defined in Problem 3.21.

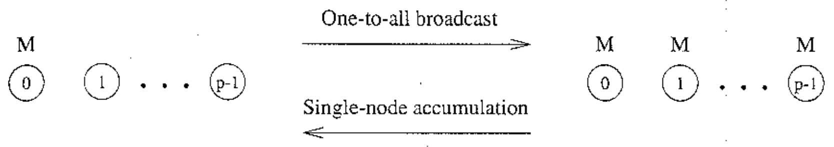


Figure 3.1 One-to-all broadcast and single-node accumulation.

A parallel algorithm may require that a single processor accumulates information from every other processor. This operation is known as ***single-node accumulation***, and is the dual of one-to-all broadcast (Figure 3.1). In single-node accumulation, every processor initially has a message containing m words. The data from all processors are combined through an associative operator, and accumulated at a single destination processor. The total size of the accumulated data remains m after the operation. Thus, single-node accumulation can be used to find the sum, product, maximum, or minimum of a set of numbers, or perform any bitwise operation on elements of the set.

One-to-all broadcast and single-node accumulation 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 architectures using both SF and CT routing schemes.

3.2.1 Store-and-Forward Routing

A naive way to perform one-to-all broadcast is to sequentially send $p - 1$ messages from the source to the other $p - 1$ processors. This is quite wasteful because with SF routing, a message traveling over more than one link is stored on all intermediate processors. We can avoid redundant transmission of the same message if every processor makes a copy of the message upon receiving it, and then forwards it to the next processor. We now present efficient ways to implement one-to-all broadcast with SF routing on the ring, mesh, and hypercube architectures.

Ring

The steps in a one-to-all broadcast on an eight-processor ring are shown in Figure 3.2. The processors 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. As shown in Figure 3.2, the source processor sends the message to its two neighbors in successive steps. Each processor receives a message on one of its links and passes that message to its neighbor on the second link. This process continues until all processors have a copy of the message. The entire procedure requires $\lceil p/2 \rceil$ steps on a p-processor ring. Each of the $\lceil p/2 \rceil$ nearest-neighbor communications takes $t_s + t_w m$ time, so the communication time of one-to-all broadcast

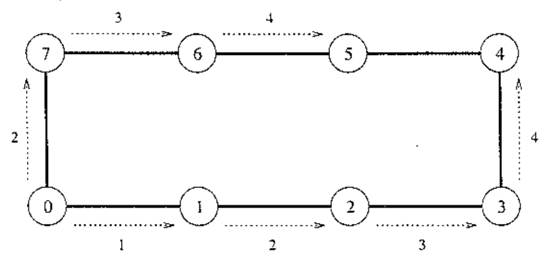


Figure 3.2 One-to-all broadcast on an eight-processor ring with SF routing. Processor 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

on a ring with SF routing is

$$T_{one\_to\_all} = (t_s + t_w m)\left\lceil \frac{p}{2} \right\rceil. \tag{3.1}$$

Example 3.1 Matrix-Vector Multiplication

Consider the problem of multiplying an $n \times n$ matrix with an $n \times 1$ vector on an $n \times n$ mesh of processors. As shown in Figure 3.3, each element of the matrix resides on a different processor, and the vector is distributed among the processors in the topmost row of the mesh. Since all the rows of the matrix must be multiplied with the vector, each processor needs the element of the vector residing in the topmost processor of its column. Hence, before computing the matrix-vector product, each column of processors performs a one-to-all broadcast of the vector elements with the topmost processor of the column as the source. This is done by treating each column of the $n \times n$ mesh as an n-processor ring, and simultaneously applying the ring broadcast procedure described previously to all columns. ■

Mesh

We can regard each row and column of a square mesh of p processors as a ring of $\sqrt{p}$ processors. So a number of communication algorithms on the mesh are simple extensions of their ring counterparts. Every ring communication operation discussed in this chapter 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 rings. In the second phase, the columns are treated similarly.

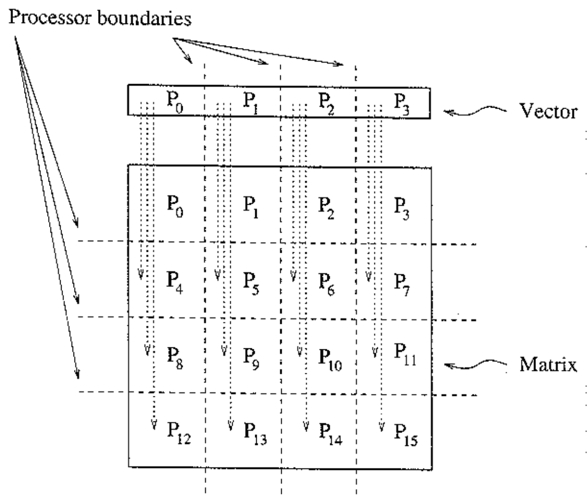


Figure 3.3 One-to-all broadcast in the multiplication of a 4×4 matrix with a 4×1 vector.

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 $(\sqrt{p} - 1)$ processors of the same row. Once all the processors 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 processor 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 3.4 for $n = 4$, with processor 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.

If the size of the message is m, the row broadcast takes $(t_s + t_w m)\lceil\sqrt{p}/2\rceil$ time. This is the same as the time required for one-to-all broadcast on a ring of $\sqrt{p}$ processors. In the second phase, the one-to-all broadcasts in all the columns are carried out in parallel. Hence, the second phase takes the same amount of time as the first, and the time for the entire broadcast is

$$T_{one\_to\_all} = 2(t_s + t_w m)\left\lceil \frac{\sqrt{p}}{2} \right\rceil. \tag{3.2}$$

We can use a similar procedure for one-to-all broadcast on a three-dimensional mesh as well. In this case, rows of $p^{1/3}$ processors in each of the three dimensions of the mesh are treated as rings. By applying the procedure for the ring in three phases, once along each dimension, the broadcast time is

$$T_{one\_to\_all} = 3(t_s + t_w m)\left\lceil \frac{p^{1/3}}{2} \right\rceil. \tag{3.3}$$

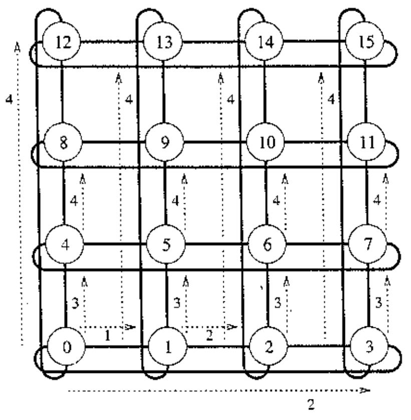


Figure 3.4 One-to-all broadcast on a 16-processor mesh with SF routing.

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 processors can be regarded as a d-dimensional mesh with two processors in each dimension. Hence, the mesh algorithm can be extended for the hypercube, except that the process is now carried out in d steps—one in each dimension.

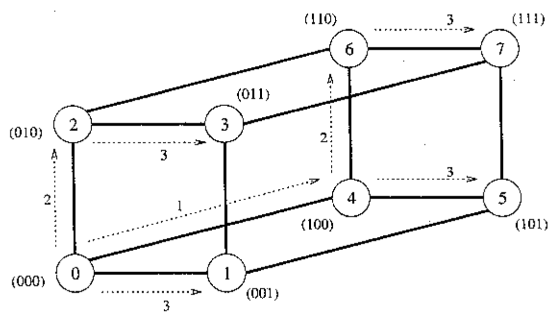


Figure 3.5 One-to-all broadcast on a three-dimensional hypercube. The binary representations of processor labels are shown in parentheses.

```
procedure ONE_TO_ALL_BC($d$, $my\_id$, $X$)
begin
   $mask := 2^d - 1$;                /* Set all $d$ bits of $mask$ to 1 */
   for $i := d - 1$ downto 0 do      /* Outer loop */
   begin
      $mask := mask$ XOR $2^i$;     /* Set bit $i$ of $mask$ to 0 */
      if ($my\_id$ AND $mask$) = 0 then
      /* If the lower $i$ bits of $my\_id$ are 0 */
         if ($my\_id$ AND $2^i$) = 0 then
         begin
            $msg\_destination := my\_id$ XOR $2^i$;
            send $X$ to $msg\_destination$;
         endif
         else
         begin
            $msg\_source := my\_id$ XOR $2^i$;
            receive $X$ from $msg\_source$;
         endelse;
   endfor;
end ONE_TO_ALL_BC
```

Program 3.1 One-to-all broadcast of a message X from processor 0 of a d-dimensional hypercube. AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

Each step is a one-to-all broadcast on a two-processor ring, which is the same as a simple message transfer between two directly-connected processors.

Figure 3.5 shows a one-to-all broadcast on an eight-processor hypercube with processor 0 as the source. As the figure shows, there is a total of three communication steps. Note that the order in which the dimensions are chosen for communication does not affect the outcome of the procedure. Figure 3.5 shows only one such order. In this scheme, communication starts along the highest dimension (that is, the dimension specified by the most significant bit of the binary representation of a processor label) and proceeds along successively lower dimensions in subsequent steps. Each of the $\log p$ steps takes $t_s + t_w m$ time for a single message transfer in each dimension. Therefore, the total time taken by the procedure on a p-processor hypercube is

$$T_{one\_to\_all} = (t_s + t_w m) \log p. \tag{3.4}$$

Example 3.2 A One-to-All Broadcast Procedure for Hypercube

In this example, we describe a procedure ONE_TO_ALL_BC(d, my_id, X) to implement the one-to-all broadcast algorithm on a d-dimensional hypercube. Program 3.1 gives the pseudocode for this procedure when processor 0 is the source of the broadcast. The procedure is executed at all processors concurrently. At any processor, the value of *my_id* is the label of that processor.

Let X be the message to be broadcast, which initially resides at the source processor 0. The procedure performs d communication steps, one in each dimension of the hypercube. In Program 3.1, 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 processors 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 3.5, i is equal to 2 in the first time step. Therefore, only processors 0 and 4 communicate, since their two least significant bits are zero. In the next time step, when $i = 1$, all processors (that is, 0, 2, 4, and 6) with zero in their least significant bits participate in communication.

The variable *mask* helps determine which processors 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 6). Line 7 determines which processors communicate in the current iteration of the outer loop. For instance, for the hypercube of Figure 3.5, *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 7 selects only those processors which have zero in their i least significant bits.

Among the processors selected for communication along dimension i, the processors with a zero in bit position i send the data, and the processors with a one in bit position i receive it. The test to determine the sending and receiving processors is performed on line 8. For example, in Figure 3.5, processor 0 (000) is the sender and processor 4 (100) is the receiver in the iteration corresponding to $i = 2$. Similarly, for $i = 1$, processors 0 (000) and 4 (100) are senders while processors 2 (010) and 6 (110) are receivers.

The procedure terminates after communication has taken place along all dimensions. ■

Example 3.3 A General One-to-All Broadcast Procedure for Hypercube

Program 3.1 works only if processor 0 is the source of the broadcast. For an arbitrary source, we must relabel the processors of the hypercube by XORing the label of each processor with the label of the source processor 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 Program 3.2. By performing the XOR operation at line 3, Program 3.2 relabels the source processor to 0, and relabels the other processors

```
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.      begin
7.         $mask := mask$ XOR $2^i$;   /* Set bit $i$ of $mask$ to 0 */
8.         if ($my\_virtual\_id$ AND $mask$) = 0 then
9.            if ($my\_virtual\_id$ AND $2^i$) = 0 then
10.           begin
11.              $virtual\_dest := my\_virtual\_id$ XOR $2^i$;
12.              send $X$ to ($virtual\_dest$ XOR $source$); /* Convert $virtual\_dest$
                                  to the label of the physical destination */
13.           endif
14.           else
15.           begin
16.              $virtual\_source := my\_virtual\_id$ XOR $2^i$;
17.              receive $X$ from ($virtual\_source$ XOR $source$);
     /* Convert $virtual\_source$ to the label of the physical source */
18.           endelse;
19.     endfor;
20.  end GENERAL_ONE_TO_ALL_BC
```

Program 3.2 One-to-all broadcast of a message X initiated by *source* in a d-dimensional hypercube. The AND and XOR operations are bitwise logical operations.

relative to the source. After this relabeling, the algorithm of Program 3.1 can be applied to perform the broadcast. ■

Example 3.4 A Hypercube Procedure for Single-Node Accumulation

Program 3.3 gives a procedure to perform a single-node accumulation on a d-dimensional hypercube such that the final result is accumulated on processor 0. Single node-accumulation is the dual of one-to-all broadcast. We obtain the communication pattern required to implement single-node accumulation by reversing the order and the direction of messages in one-to-all broadcast. Procedure SINGLE_NODE_ACC(d, my_id, m, X, sum) shown in Program 3.3 is very similar to procedure ONE_TO_ALL_BC(d, my_id, X) shown in Program 3.1. One difference is that the communication in single-node accumulation proceeds from the lowest to the highest dimension. This change is reflected in the way that variables *mask* and i are manipulated in Program 3.3. The criterion for determining the source and the destination among a pair of communicating processors is also reversed (line 8).

```
 1.  procedure SINGLE_NODE_ACC($d$, $my\_id$, $m$, $X$, $sum$)
 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
 6.     begin /* Select processors whose lower $i$ bits are 0 */
 7.        if ($my\_id$ AND $mask$) $= 0$ then
 8.           if ($my\_id$ AND $2^i$) $\neq 0$ then
 9.           begin
10.              $msg\_destination := my\_id$ XOR $2^i$;
11.              send $sum$ to $msg\_destination$;
12.           endif
13.           else
14.           begin
15.              $msg\_source := my\_id$ XOR $2^i$;
16.              receive $X$ from $msg\_source$;
17.              for $j := 0$ to $m-1$ do
18.                 $sum[j] := sum[j] + X[j]$;
19.           endelse;
20.        $mask := mask$ XOR $2^i$;   /* Set bit $i$ of $mask$ to 1 */
21.     endfor;
22.  end SINGLE_NODE_ACC
```

Program 3.3 Single-node accumulation on a d-dimensional hypercube. Each processor contributes a message X containing m words, and processor 0 is the destination of the sum. The AND and XOR operations are bitwise logical operations.

Apart from these differences, procedure SINGLE_NODE_ACC has extra instructions (lines 17 and 18) to add the contents of the messages received by a processor in each iteration.

Note that any associative operation can be used in place of addition. ■

Among the architectures considered so far, one-to-all broadcast takes the shortest time on a hypercube because a hypercube has a higher connectivity than a ring or a mesh. In Section 3.7.1 we show how we can further reduce the communication time of this procedure by splitting the message into smaller parts and routing each part separately. However, if a message is not routed in parts along separate paths and communication is allowed on only one link of each processor at a time, then one-to-all broadcast cannot be performed in less than $(t_s + t_w m) \log p$ time on any architecture. This can be inferred from two observations regarding the hypercube procedure illustrated in Figure 3.5. First, at any time, each processor that possesses the data is sending that data to a processor that

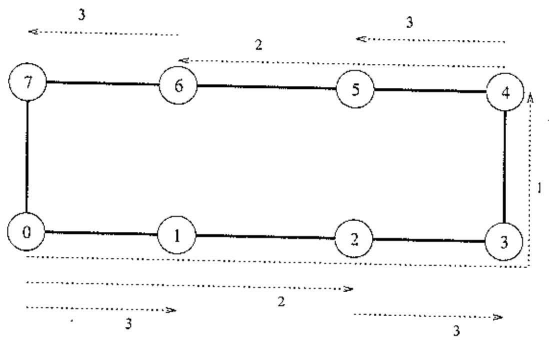


Figure 3.6 One-to-all broadcast with CT routing on an eight-processor ring.

needs it. This is not the case with the ring and the mesh algorithms. For instance, in Figure 3.2, processor 0 does not communicate in steps 3 or 4, although it has the message being broadcast, and there are processors waiting for it. Second, all messages are passed only between directly-connected processors on the hypercube; hence, each communication step is of the minimum possible duration for a message of size m. Thus, on a hypercube, every opportunity to send a message is used and each message transfer is of the smallest possible duration for the given message size. A better-connected architecture cannot send any more messages at a time, or reduce the transfer time for any message (for the same t_s, t_w, and m). Hence, the time $(t_s + t_w m) \log p$ is the best for one-to-all broadcast under the given conditions.

3.2.2 Cut-Through Routing

Of the three architectures considered in Section 3.2.1, one-to-all broadcast with SF routing is fastest on the hypercube. The communication time of one-to-all broadcast on a hypercube does not improve with CT routing due to exclusively nearest-neighbor communication. However, the operation benefits substantially from CT routing on ring and mesh architectures.

Ring

CT routing can be used advantageously for one-to-all broadcast on a ring by mapping the hypercube algorithm onto the ring. In every step, each processor of the ring communicates with the same processor as in the hypercube algorithm. This process is illustrated in Figure 3.6 for an eight-processor ring. Comparing Figure 3.5 with Figure 3.6 shows that in both cases, communication takes place between the same pair of processors in each step. With CT routing, such a mapping is useful because the complexity of passing a message of size m between processors separated by l links is only $\Theta(m + l)$. As Figure 3.6 illustrates,

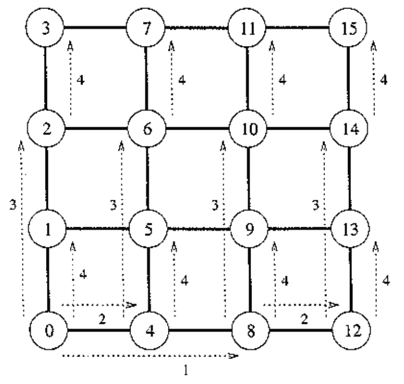


Figure 3.7 One-to-all broadcast on a 16-processor square mesh with CT routing.

in a p-processor ring the source processor first sends the data to a processor at a distance $p/2$. In the second step, both processors that have the data transmit it to processors at a distance of $p/4$ in the same direction. Assuming that p is a power of 2, in the i^{th} step, each processor that has the data sends it to a processor at a distance of $p/2^i$. All messages flow in the same direction. The algorithm concludes after $\log p$ steps.

The communication time in the i^{th} step is $t_s + t_w m + t_h p/2^i$. Hence, the total time for the broadcast with CT routing on a ring of p processors is

$$\begin{aligned} T_{one-to-all} &= \sum_{i=1}^{\log p} (t_s + t_w m + t_h p/2^i) \\ &= t_s \log p + t_w m \log p + t_h (p-1). \end{aligned} \quad (3.5)$$

For sufficiently large values of m, the t_h term is insignificant compared to the others. Thus, CT routing effectively reduces the communication time by a factor of $p/\log p$ over SF routing. For example, if $m = \Omega(p/\log p)$, then a one-to-all broadcast takes $\Theta(p^2/\log p)$ time on ring with SF routing, but only $\Theta(p)$ time with CT routing.

Mesh

On a two-dimensional square mesh with CT routing, one-to-all broadcast is performed in two phases. In each phase the ring procedure is applied in a different dimension of the mesh. The procedure is illustrated in Figure 3.7 for a 16-processor mesh. First, a one-to-all broadcast is initiated by the source processor among the $\sqrt{p}$ processors in its row (call it the source row). Second, a one-to-all broadcast is initiated in each column by its processor in the source row. Each of the two phases takes $(t_s + t_w m) \log \sqrt{p} + t_h(\sqrt{p} - 1)$ time, and

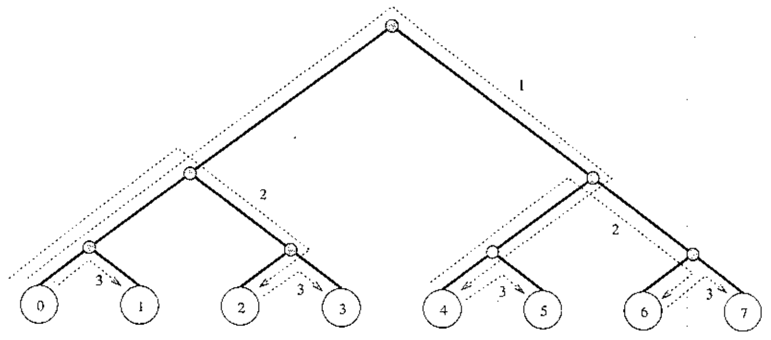


Figure 3.8 One-to-all broadcast on an eight-processor tree.

the time for the entire broadcast is

$$T_{one\_to\_all} = (t_s + t_w m) \log p + 2t_h(\sqrt{p} - 1). \quad (3.6)$$

Every communication step in the 16-processor mesh shown in Figure 3.7 takes place between exactly the same processors as in a 16-processor hypercube. Note that steps 2, 3, and 4 of Figure 3.7 are identical to steps 1, 2, and 3 of Figure 3.5. Like the one-to-all broadcast procedure for a ring with cut-through routing, the mesh procedure is also a direct adaptation of the hypercube procedure given in Programs 3.1 and 3.2.

Balanced Binary Tree

The hypercube algorithm for one-to-all broadcast maps naturally onto a balanced binary tree in which each leaf is a processor and intermediate nodes serve only as switching units. This is illustrated in Figure 3.8 for eight processors. In this figure, the communicating processors have the same labels as in the hypercube algorithm illustrated in Figure 3.5. Figure 3.8 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 3.8 is that there is a different number of switching nodes along different paths on the tree. Assuming that a per-hop time of t_h is associated with each link between two switching nodes or between a processor and a switching node, the time for one-to-all broadcast (Problem 3.1) on the tree is

$$T_{one\_to\_all} = (t_s + t_w m + t_h(\log p + 1)) \log p. \quad (3.7)$$

3.3 All-to-All Broadcast, Reduction, and Prefix Sums

All-to-all broadcast, also known as ***multinode broadcast***, is a generalization of one-to-all broadcast in which all p processors simultaneously initiate a broadcast. A processor

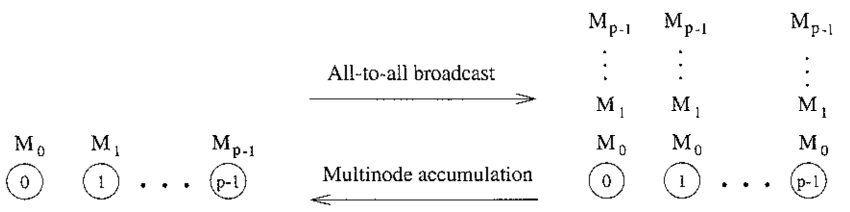


Figure 3.9 All-to-all broadcast and multinode accumulation.

sends the same m-word message to every other processor, but different processors 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 ***multinode accumulation***, in which every processor is the destination of a single-node accumulation (Problem 3.8). Figure 3.9 illustrates all-to-all broadcast and multinode accumulation.

The communication pattern of all-to-all broadcast can be used to perform some other operations as well, such as, *reduction* and *prefix sums*. Examples 3.7 and 3.8 discuss reduction and prefix sums on a hypercube.

One way to perform an all-to-all broadcast is to perform p one-to-all broadcasts, one starting at each processor. 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 ring, mesh, and hypercube topologies using both SF and CT routings.

3.3.1 Store-and-Forward Routing

Ring

The one-to-all broadcast procedure shown in Figure 3.2 for a ring with SF routing shows that at most two communication links are active during any given time step. For all-to-all broadcast, all channels can be kept busy simultaneously because, unlike one-to-all broadcast, each processor always has some information that it can pass along to its neighbor. Each processor first sends to one of its neighbors the data it needs to broadcast. In subsequent steps, it forwards the data received from one of its neighbors to its other neighbor.

Figure 3.10 illustrates this procedure for an eight-processor ring. As with the previous figures, the integer label of an arrow indicates the time step during which the message is sent. In all-to-all broadcast, p different messages circulate in the p-processor ensemble. In Figure 3.10, each message is identified by its initial source, whose label appears in parentheses along with the time step. For instance, the arc labeled 2 (7) between processors 0

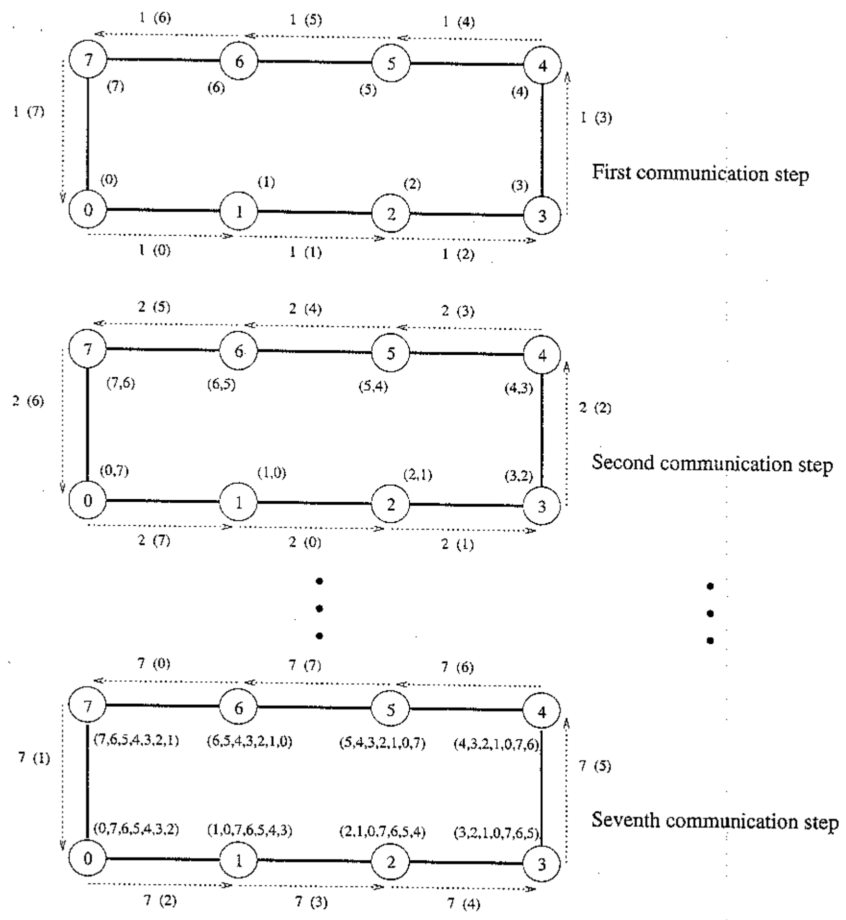


Figure 3.10 All-to-all broadcast on an eight-processor ring with SF routing. In addition to the time step, the label of each arrow has an additional number in parentheses. This number labels a message and indicates the processor from which the message originated in the first step. The number(s) in parentheses next to each processor are the labels of processors from which data has been received prior to the communication step. Only the first, second, and last communication steps are shown.

and 1 represents the data communicated in time step 2 that processor 0 received from processor 7 in the preceding step. As Figure 3.10 shows, if communication is performed circularly in a single direction, then each processor receives all $(p-1)$ pieces of information from all other processors in $(p-1)$ steps. The time taken by the entire operation is

$$T_{all\_to\_all} = (t_s + t_w m)(p-1). \tag{3.8}$$

The straightforward all-to-all broadcast algorithm presented above for a simple architecture like a ring with SF routing has great practical importance. A close look at the algorithm reveals that it is a sequence of p one-to-all broadcasts, each with a different source. These broadcasts are pipelined so that all of them are complete in a total of p nearest-neighbor communication steps. Many parallel algorithms involve a series of one-to-all broadcasts with different sources, often interspersed with some computation. If each one-to-all broadcast is performed using the hypercube algorithm given in Section 3.2.1, then the total time spent in communication is $n(t_s + t_w m)\log p$, where n is the number of broadcasts. On the other hand, by pipelining the broadcasts as shown in Figure 3.10, all the of them can be performed spending no more than $(t_s + t_w m)(p-1)$ time in communication, provided that the sources of all broadcasts are different and $n \le p$. In later chapters, we show how such pipelined broadcast improves the performance of some parallel algorithms such as Gaussian elimination (Section 5.5.1), back substitution (Section 5.5.3), Fox's algorithm for matrix multiplication (Problem 5.23), and Floyd's algorithm for finding the shortest paths in a graph (Section 7.4.3).

Mesh

Just like one-to-all broadcast, the all-to-all broadcast algorithm for the 2-D mesh is based on the ring algorithm, treating rows and columns of the mesh as rings. Once again, communication takes place in two phases. In the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the ring. In this phase, all processors collect $\sqrt{p}$ messages corresponding to the $\sqrt{p}$ processors of their respective rows. Each processor consolidates this information into a single message of size $m\sqrt{p}$, and proceeds to the second communication phase of the algorithm. The second communication phase is a columnwise all-to-all broadcast of the consolidated messages. By the end of this phase, each processor obtains all p pieces of m-word data that originally resided on different processors. The distribution of data among the processors of a 3×3 mesh at the beginning of the first and the second phases of the algorithm is shown in Figures 3.11(a) and (b), respectively.

The first phase of $\sqrt{p}$ simultaneous all-to-all broadcasts (each among $\sqrt{p}$ processors) concludes in time $(t_s + t_w m)(\sqrt{p}-1)$. The number of processors participating in each all-to-all broadcast in the second phase is also $\sqrt{p}$, but the size of each message is now $m\sqrt{p}$. Therefore, this phase takes $(t_s + t_w m\sqrt{p})(\sqrt{p}-1)$ time to complete. The time for the entire all-to-all broadcast on a p-processor two-dimensional square mesh is the sum of the times spent in the individual phases, which is

$$T_{all\_to\_all} = 2t_s(\sqrt{p}-1) + t_w m(p-1). \tag{3.9}$$

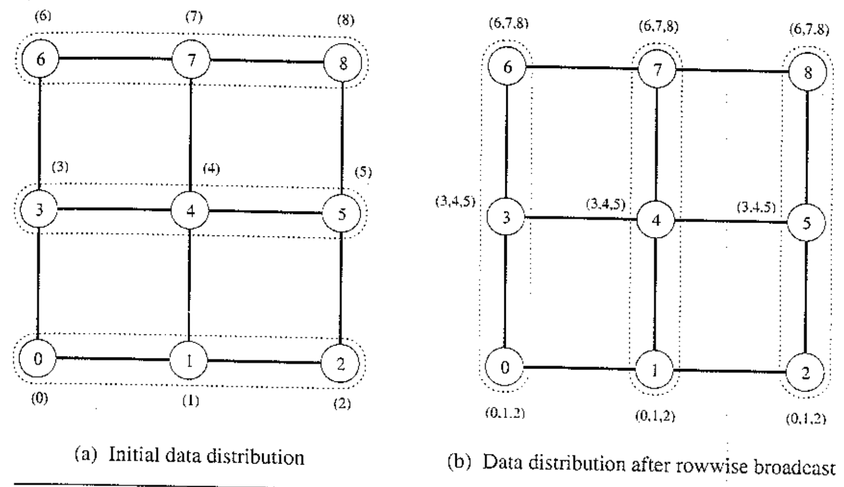


Figure 3.11 All-to-all broadcast on a 3×3 mesh. The groups of processors communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all processors get (0,1,2,3,4,5,6,7) (that is, a message from each processor).

Example 3.5 Procedures for All-to-All Broadcast on Ring and Mesh
Programs 3.4 and 3.5 give procedures for all-to-all broadcast on a p-processor ring and a p-processor mesh, respectively. The initial message to be broadcast is known locally as *my_msg* at each processor. At the end of the procedure, each processor stores the collection of all p messages in *result*. As the programs show, all-to-all broadcast on a mesh applies the ring procedure twice, once along the rows and once along the columns. ■

Hypercube

The hypercube algorithm for all-to-all broadcast is an extension of the mesh algorithm to $\log p$ dimensions. The procedure requires $\log p$ steps. Communication takes place along a different dimension of the p-processor hypercube in each step. In every step, pairs of processors exchange their data and double the size of the message to be transmitted in the next step by concatenating the received message with their current data. Figure 3.12 shows these steps for an eight-processor hypercube with bidirectional communication channels. The size of the messages exchanged in the i^{th} of the $\log p$ steps is $2^{i-1}m$. The time it takes a pair of processors to send and receive messages from each other is $t_s + 2^{i-1}t_w m$. Hence,

```
procedure ALL_TO_ALL_BC_RING($my\_id$, $my\_msg$, $p$, $result$)
begin
    $left := (my\_id - 1) \bmod p$;
    $right := (my\_id + 1) \bmod p$;
    $result := my\_msg$;
    $msg := result$;
    for $i := 1$ to $p - 1$ do
    begin
        send $msg$ to $right$;
        receive $msg$ from $left$;
        $result := result \cup msg$;
    endfor;
end ALL_TO_ALL_BC_RING
```

Program 3.4 All-to-all broadcast on a p-processor ring.

the time it takes to complete the entire procedure is

$$\begin{aligned} T_{all\_to\_all} &= \sum_{i=1}^{\log p}(t_s + 2^{i-1}t_w m) \\ &= t_s \log p + t_w m(p-1). \end{aligned} \tag{3.10}$$

Example 3.6 An All-to-All Broadcast Procedure for Hypercube
Program 3.6 gives a procedure for implementing all-to-all broadcast on a d-dimensional hypercube. Communication starts from the lowest dimension of the hypercube and then proceeds along successively higher dimensions (line 4). In each iteration, processors communicate in pairs so that the labels of the processors communicating with each other in the i^{th} iteration differ in the i^{th} least significant bit of their binary representations (line 6). After an iteration's communication steps, each processor concatenates the data it receives during that iteration with its resident data (line 9). This concatenated message is transmitted in the following iteration. ■

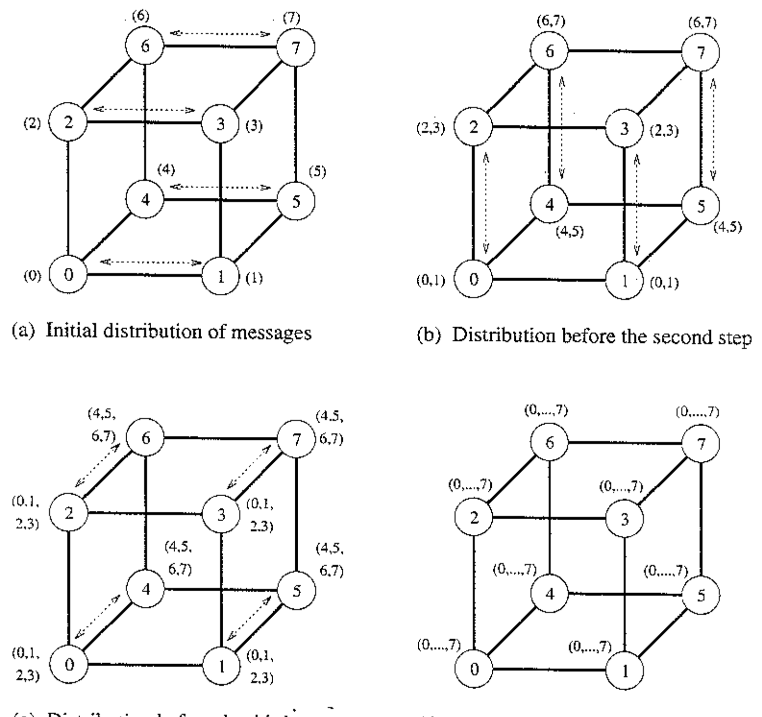
Example 3.7 Reduction on a Hypercube
The communication pattern used in all-to-all broadcast is employed in other hypercube algorithms as well. For example, consider the operation in which every processor of a hypercube starts with one value and needs to know the sum of the values stored at all the processors. This operation is known as ***reduction***. In general, any associative operation (such as logical OR, logical AND, maximum, or minimum) can be used instead of addition. Reduction is often used to implement barrier synchronization (Section 13.4.2) on a message-passing computer. The semantics of the

```
procedure ALL_TO_ALL_BC_MESH($my\_id$, $my\_msg$, $p$, $result$)
begin
/* Communication along rows */
    $left := (my\_id - 1) \bmod p$;
    $right := (my\_id + 1) \bmod p$;
    $result := my\_msg$;
    $msg := result$;
    for $i := 1$ to $\sqrt{p} - 1$ do
    begin
        send $msg$ to $right$;
        receive $msg$ from $left$;
        $result := result \cup msg$;
    endfor;
/* Communication along columns */
    $up := (my\_id - \sqrt{p}) \bmod p$;
    $down := (my\_id + \sqrt{p}) \bmod p$;
    $msg := result$;
    for $i := 1$ to $\sqrt{p} - 1$ do
    begin
        send $msg$ to $down$;
        receive $msg$ from $up$;
        $result := result \cup msg$;
    endfor;
end ALL_TO_ALL_BC_MESH
```

Program 3.5 All-to-all broadcast on a square mesh of p processors.

reduction operation are such that, while executing a parallel program, no processor can finish reduction before each processor has contributed a value.

A naive algorithm for reduction would perform an all-to-all broadcast, gather all the numbers at each processor, and then add them locally on all processors. Since the message size m is one word, the communication time of this procedure is $t_s \log p + t_w(p-1)$. A faster method to perform reduction is to perform a single-node accumulation followed by a one-to-all broadcast. However, there is an even faster way to perform reduction by using the communication pattern of all-to-all broadcast. Figure 3.12 illustrates this algorithm for $p = 8$. Assume that each integer in parentheses in the figure, instead of denoting a message, denotes a number to be added that originally resided at the processor with that integer label. To perform reduction, we follow the communication steps of the all-to-all broadcast



(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages

Figure 3.12 All-to-all broadcast on an eight-processor hypercube.

procedure, but at the end of each step, add two numbers instead of concatenating two messages. At the termination of the reduction procedure, each processor holds the sum $(0 + 1 + 2 + \cdots + 7)$ (rather than eight messages numbered from 0 to 7, as in the case of all-to-all broadcast). Unlike all-to-all broadcast, each message transferred in the reduction operation has only one word. The size of the messages does not double in each step because the numbers are added instead of being concatenated. Therefore, the total communication time for all $\log p$ steps is

$$T_{reduction} = (t_s + t_w)\log p. \tag{3.11}$$

Program 3.6 can be used to perform a sum of p numbers if *my_msg*, *msg* and *result* are numbers (rather than messages), and the union operation ('∪') on line 9 is replaced by addition. ■

```
1.  procedure ALL_TO_ALL_BC_HCUBE(my_id, my_msg, d, result)
2.  begin
3.      result := my_msg;
4.      for i := 0 to d − 1 do
5.      begin
6.          partner := my_id XOR 2^i;
7.          send result to partner;
8.          receive msg from partner;
9.          result := result ∪ msg;
10.     endfor;
11. end ALL_TO_ALL_BC_HCUBE
```

Program 3.6 All-to-all broadcast on a d-dimensional hypercube.

Example 3.8 Prefix Sums on a Hypercube

Finding ***prefix sums*** is another important problem that can be solved by using a communication pattern similar to that used in reduction. Given p numbers $n_0, n_1, \ldots, n_{p-1}$ (one on each processor), the problem is to compute the sums $s_k = \Sigma_{i=0}^{k} n_i$ for all k between 0 and $p - 1$. For example, if the original sequence of numbers is $\langle 0, 1, 2, 3, 4\rangle$, then the sequence of prefix sums is $\langle 0, 1, 3, 6, 10\rangle$. Initially, n_k resides on the processor labeled k, and at the end of the procedure, the same processor holds s_k.

Figure 3.13 illustrates the prefix sums procedure for an eight-processor hypercube. This figure is a modification of Figure 3.12. The modification is required to accommodate the fact that in prefix sums the processor with label k uses information from only the k-processor subset of those processors whose labels are less than or equal to k. To accumulate the correct prefix sum, every processor maintains an additional result buffer. This buffer is denoted by square brackets in Figure 3.13. At the end of a communication step, the content of an incoming message is added to the result buffer only if the message comes from a processor with a smaller label than that of the recipient processor. The contents of the outgoing message (denoted by parentheses in the figure) are updated with every incoming message, just as in Example 3.7. For instance, after the first communication step, processors 0, 2, and 4 do not add the data received from processors 1, 3, and 5 to their result buffers. However, the contents of the outgoing messages for the next step are updated.

Since not all of the messages received by a processor contribute to its final result, some of the messages it receives may be redundant. We have omitted these steps of the standard all-to-all broadcast communication pattern from Figure 3.13, although the presence or absence of these messages does not affect the results of the algorithm. Program 3.7 gives a procedure to solve the prefix sums problem on a d-dimensional hypercube. ■

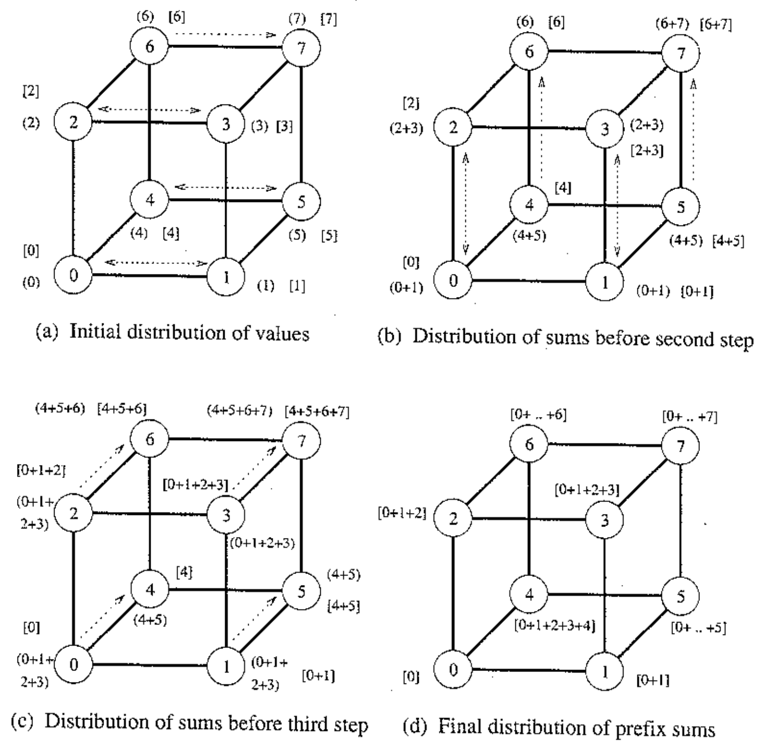


Figure 3.13 Computing prefix sums on an eight-processor hypercube. At each processor, square brackets show the local prefix sum accumulated in a buffer and parentheses enclose the contents of the outgoing message buffer for the next step.

3.3.2 Cut-Through Routing

For one-to-all broadcast, we obtain better algorithms for the ring and the mesh with CT routing simply by mapping the hypercube algorithm onto them. This strategy does not yield all-to-all broadcast algorithms for ring and mesh that are strictly better with CT routing than with SF routing. The reason is that, unlike one-to-all broadcast, the hypercube procedure for all-to-all broadcast cannot be mapped onto a ring or a mesh because it causes congestion on the communication channels. For instance, Figure 3.14 shows the result of performing the third step (Figure 3.12(c)) of the hypercube all-to-all broadcast procedure on a ring. One of the links of the ring is traversed by all four messages. Hence, passing these messages with CT routing will not be any faster than performing this communication in four steps using SF routing. However, with CT routing the ring and the mesh procedures for all-to-all

```
procedure PREFIX_SUMS_HCUBE($my\_id$, $my\_number$, $d$, $result$)
begin
    $result := my\_number$;
    $msg := result$;
    for $i := 0$ to $d - 1$ do
    begin
        $partner := my\_id$ XOR $2^i$;
        send $msg$ to $partner$;
        receive $number$ from $partner$;
        $msg := msg + number$;
        if ($partner < my\_id$) then $result := result + number$;
    endfor;
end PREFIX_SUMS_HCUBE
```

Program 3.7 Prefix sums on a d-dimensional hypercube.

broadcast shown in Figures 3.10 and 3.11 do not require wraparound connections, provided that communication channels are bidirectional. For large messages, the term associated with t_h is comparatively small and can be ignored. In this case, the communication time for all-to-all broadcast with CT routing and bidirectional links is the same on a linear array (mesh without wraparound connections) as with SF routing on a ring (mesh with wraparound connections) (Problem 3.20).

Section 3.3.1 shows that the term associated with t_w in the expressions for the communication time of all-to-all broadcast is $t_w m(p - 1)$ for all the architectures. This term also serves as a lower bound for the communication time of all-to-all broadcast for

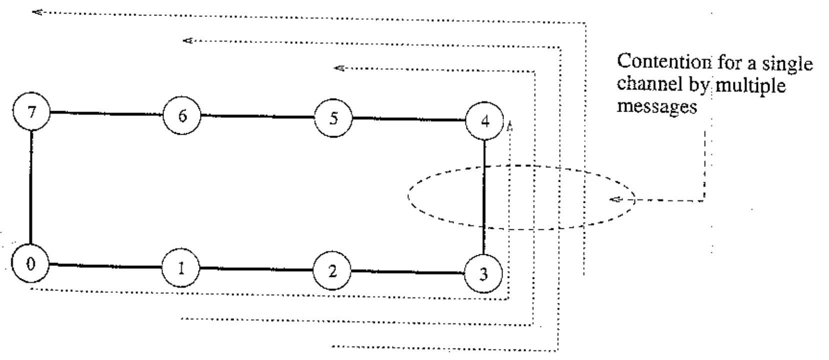


Figure 3.14 Contention for a channel when the communication step of Figure 3.12(c) for the hypercube is mapped onto a ring.

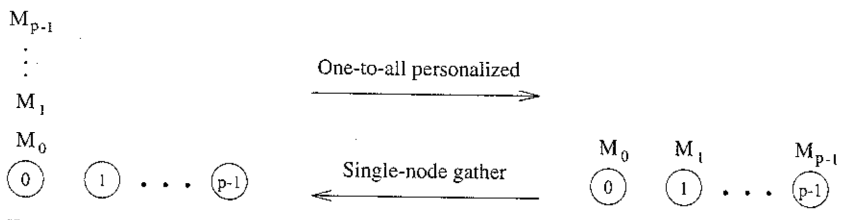


Figure 3.15 One-to-all personalized communication and its dual—single-node gather.

parallel computers on which a processor can communicate on only one of its ports at a time. This is because each processor receives at least $m(p-1)$ words of data, regardless of the architecture or routing scheme.

3.4 One-to-All Personalized Communication

In ***one-to-all personalized communication***, a single processor sends a unique message of size m to every other processor. This operation is also known as ***single-node scatter***. One-to-all personalized communication is different from one-to-all broadcast in that the source processor starts with p unique messages—one destined for each processor. Unlike one-to-all broadcast, one-to-all personalized communication does not involve any duplication of data. The related problem of ***k-to-all personalized communication*** is explored in Problem 3.26. The dual of one-to-all personalized communication is ***single-node gather***, in which a single processor collects a unique message from each other processor. The procedure for single-node gather can be derived for any interconnection topology simply by reversing the direction and sequence of messages in the corresponding one-to-all personalized communication algorithm. Again, a gather operation is different from an accumulation operation in that it does not involve any combination or reduction of data. Figure 3.15 illustrates the one-to-all personalized communication and single-node gather operations.

The complexity of one-to-all personalized communication on various architectures is similar to that of all-to-all broadcast. In all-to-all broadcast, each processor receives $m(p-1)$ words, whereas, in one-to-all personalized communication, the source processor transmits m words for each of the other $p-1$ processors in the system. Therefore, as in the case of all-to-all broadcast, $t_w m(p-1)$ is a lower bound on the communication time of one-to-all personalized communication. This lower bound is independent of the architecture or routing scheme. Because of its similarity to all-to-all broadcast, we describe this operation in detail for only the hypercube architecture. Algorithms for ring and mesh topologies are left as an exercise (Problem 3.6).

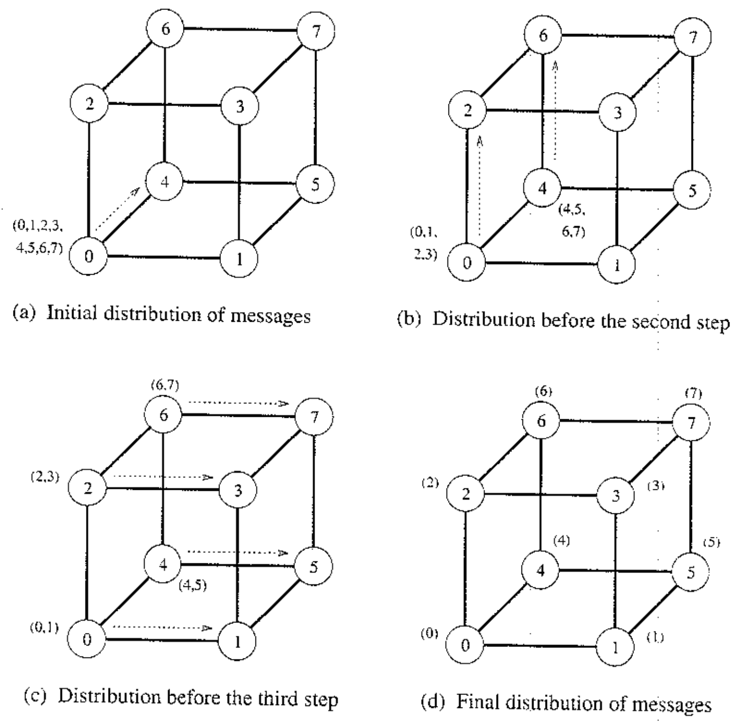

(0,1,2,3,
4,5,6,7)
(a) Initial distribution of messages
(0,1,
2,3)
(4,5,
6,7)
(b) Distribution before the second step
(6,7)
(2,3)
(4,5)
(0,1)
(c) Distribution before the third step
(6)
(7)
(2)
(3)
(4)
(5)
(0)
(1)
(d) Final distribution of messages

Figure 3.16 One-to-all personalized communication on an eight-processor hypercube.

Hypercube

Figure 3.16 shows the communication steps for one-to-all personalized communication on an eight-processor hypercube. Initially, the source processor (processor 0) contains all the messages. In the figure, messages are identified by the labels of their destination processors. In the first communication step, the source transfers half of the messages to one of its neighbors. In subsequent steps, each processor that has some data transfers half of it to a neighbor that has yet to receive any data. There is a total of $\log p$ communication steps corresponding to the $\log p$ dimensions of the hypercube. Note that the communication pattern of one-to-all broadcast (Figure 3.5) and one-to-all personalized communication (Figure 3.16) are identical. Only the size and the contents of messages are different.

All links of a p-processor hypercube along a certain dimension join two $p/2$-processor subcubes (Section 2.4.1). As Figure 3.16 illustrates, in each communication step of one-to-all personalized communication, data flow from one subcube to another. The data that a processor has before starting communication in a certain dimension are

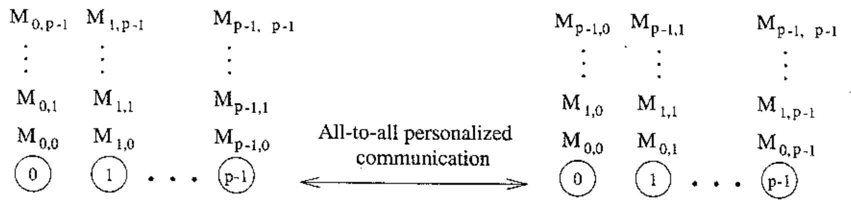


Figure 3.17 All-to-all personalized communication.

such that half of them need to be sent to a processor in the other subcube. In every step, a communicating processor keeps half of its data, meant for the processors in its subcube, and sends the other half to its neighbor in the other subcube. The time in which all data are distributed to their respective destinations is

$$T_{one\_to\_all\_pers} = t_s \log p + t_w m(p-1). \tag{3.12}$$

This time is the same as the time required for all-to-all broadcast on a similar hypercube. One-to-all personalized communication can be performed in time $(t_s + t_w m)(p-1)$ on a ring and in time $2t_s(\sqrt{p}-1) + t_w m(p-1)$ on a 2-D square mesh for both SF and CT routing (Problem 3.6).

3.5 All-to-All Personalized Communication

In ***all-to-all personalized communication***, also known as ***total exchange***, each processor sends a distinct message of size m to every other processor. Each processor sends different messages to different processors, unlike all-to-all broadcast, in which each processor sends the same message to all other processors. Figure 3.17 illustrates the all-to-all personalized communication operation. This operation is used in parallel fast Fourier transform, matrix transpose, and some parallel database join operations.

We now discuss the implementation of all-to-all personalized communication on parallel computers with ring, mesh, and hypercube interconnection networks. The communication patterns of all-to-all personalized communication are identical to those of all-to-all broadcast on all three architectures. Only the size and the contents of messages are different.

3.5.1 Store-and-Forward Routing

Ring

Figure 3.18 shows the steps in all-to-all personalized communication on a six-processor ring. To perform this operation, every processor sends $p-1$ pieces of data, each of size m. In the figure, these pieces of data are identified by pairs of integers of the form $\{i, j\}$, where

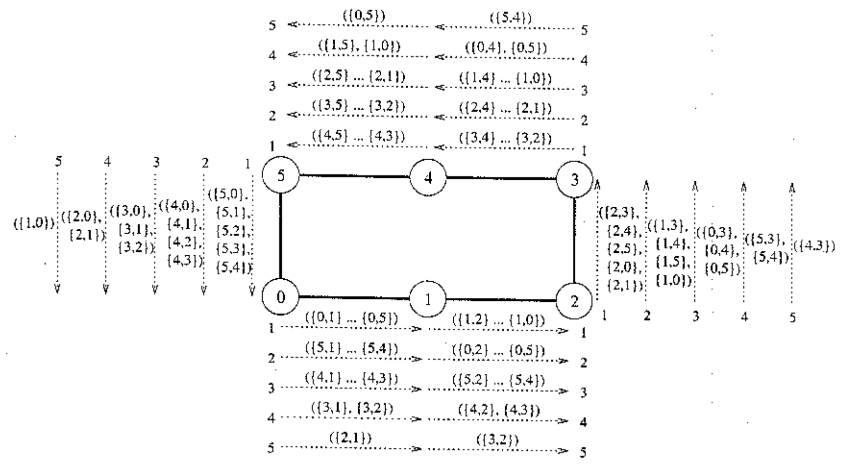
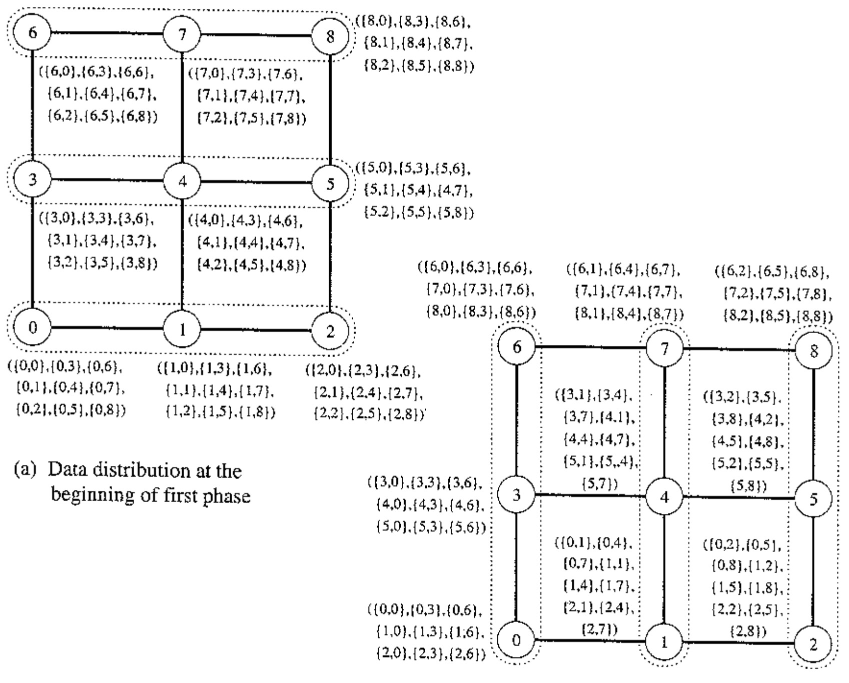


Figure 3.18 All-to-all personalized communication on a six-processor ring. The label of each message is of the form $\{x, y\}$, where x is the label of the processor that originally stored the message, and y is the label of the processor that is the final destination of the message. The label $(\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\})$ indicates a message that is formed by concatenating n individual messages.

i is the source of the message and j is its final destination. First, each processor sends all pieces of data as one consolidated message of size $m(p-1)$ to one of its neighbors (all processors communicate in the same direction). Of the $m(p-1)$ words of data received by a processor in this step, one m-word packet belongs to it. Therefore, each processor extracts the information meant for it from the data received, and forwards the remainder ($p-2$ pieces of size m each) to the next processor. This process continues for $p-1$ steps. The size of the messages being transferred between processors decreases by m words in each successive step. In every step, each processor adds to its collection one m-word packet originating from a different processor. Hence, in $p-1$ steps, every processor receives the information from all other processors in the ensemble. Since the size of the messages transferred in the ith step is $m(p-i)$ on a ring of processors, the total time taken by this operation is

$$\begin{aligned} T_{all\_to\_all\_pers} &= \sum_{i=1}^{p-1}(t_s + t_w m(p-i)) \\ &= t_s(p-1) + \sum_{i=1}^{p-1} i t_w m \\ &= (t_s + \frac{1}{2} t_w m p)(p-1). \end{aligned} \tag{3.13}$$



(a) Data distribution at the beginning of first phase

(b) Data distribution at the beginning of second phase

Figure 3.19 The distribution of messages at the beginning of each phase of all-to-all personalized communication on a 3×3 mesh. At the end of the second phase, processor i has messages $(\{0,i\}, \ldots, \{8,i\})$, where $0 \le i \le 8$. The groups of processors communicating together in each phase are enclosed in dotted boundaries.

In the procedure we just described, all messages are sent in the same direction. If half of the messages are sent in one direction and the remaining half are sent in the other direction, then the term associated with t_w can be reduced by a factor of two (Problem 3.3). For the sake of simplicity, we ignore this constant-factor improvement in the remainder of the section.

Mesh

In all-to-all personalized communication on a $\sqrt{p} \times \sqrt{p}$ mesh, each processor first groups its p messages according to the columns of their destination processors. Figure 3.19 shows a 3×3 mesh, in which every processor initially has nine m-word messages—one meant for each processor. Each processor assembles its data into three groups of three messages each (in general, $\sqrt{p}$ groups of $\sqrt{p}$ messages each). The first group contains the messages destined for processors labeled 0, 3, and 6; the second group contains the messages for processors labeled 1, 4, and 7; and the last group has messages for processors labeled 2, 5, and 8.

After the messages are grouped, all-to-all personalized communication is performed independently in each row with clustered messages of size $m\sqrt{p}$. One cluster contains the information for all $\sqrt{p}$ processors of a particular column. Figure 3.19(b) shows the distribution of data among the processors at the end of this phase of communication. Assuming a square mesh, we can compute the time spent in this phase by substituting $\sqrt{p}$ for the number of processors, and $m\sqrt{p}$ for the message size in Equation 3.13. The result of this substitution is $(t_s + t_w mp/2)(\sqrt{p} - 1)$.

Before the second communication phase, the messages in each processor are sorted again, this time according to the rows of their destination processors; then communication similar to the first phase takes place in all the columns of the mesh. By the end of this phase, each processor receives a message from every other processor. The time spent in this phase is the same as that in the first phase. Therefore, the total time for all-to-all personalized communication of messages of size m on a p-processor two-dimensional square mesh is

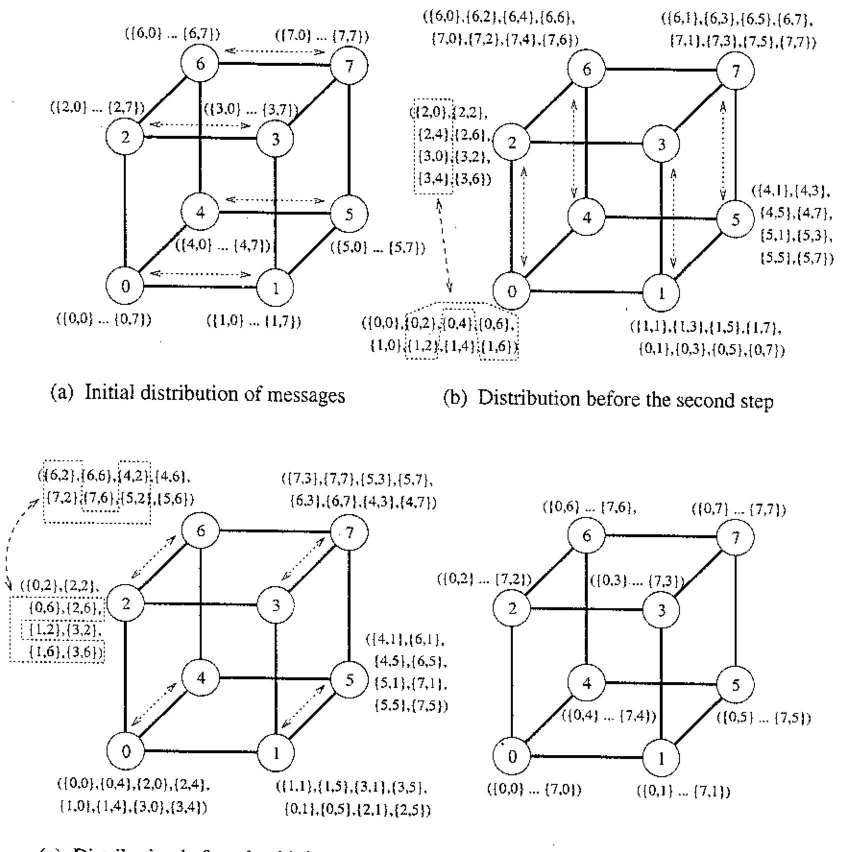
$$T_{all\_to\_all\_pers} = (2t_s + t_w mp)(\sqrt{p} - 1). \qquad (3.14)$$

The expression for the communication time of all-to-all personalized communication in Equation 3.14 does not take into account the time required for the local rearrangement of data (that is, sorting the messages by rows or columns). Assuming that initially the data is ready for the first communication phase, the second communication phase requires the rearrangement of mp words of data. If t_r is the time to perform a read and a write operation on a single word of data in a processor's local memory, then the total time spent in data rearrangement by a processor during the entire procedure is $t_r mp$ (Problem 3.27). This time is much smaller than the time spent by each processor in communication.

Hypercube

The all-to-all personalized communication algorithm for a p-processor hypercube with SF routing is simply an extension of the two-dimensional mesh algorithm to $\log p$ dimensions. Figure 3.20 shows the communication steps required to perform this operation on a three-dimensional hypercube. As shown in the figure, communication takes place in $\log p$ steps. Pairs of processors exchange data in a different dimension in each step. Recall that in a p-processor hypercube, a set of $p/2$ links in the same dimension connects two subcubes of $p/2$ processors each (Section 2.4.1). At any stage in all-to-all personalized communication, every processor holds p packets of size m each. While communicating in a particular dimension, every processor sends $p/2$ of these packets (consolidated as one message). The destinations of these packets are the processors of the other subcube connected by the links in current dimension. Thus, $mp/2$ words of data are exchanged along the bidirectional channels in each of the $\log p$ iterations. The resulting total communication time is

$$T_{all\_to\_all\_pers} = (t_s + \frac{1}{2}t_w mp) \log p. \qquad (3.15)$$



(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages

Figure 3.20 All-to-all personalized communication on a three-dimensional hypercube with SF routing.

In the preceding procedure, a processor must rearrange its messages locally before each of the log p communication steps. This is necessary to make sure that all $p/2$ messages destined for the same processor in a communication step occupy contiguous memory locations so that they can be transmitted as a single consolidated message. Before each of the log p communication steps, a processor rearranges mp words of data (Problem 3.28). Hence, a total of $t_r mp \log p$ time is spent by each processor in local rearrangement of data during the entire procedure. Here t_r is the time needed to perform a read and a write operation on a single word of data in a processor's local memory. For most practical computers, t_r is much smaller that t_w; hence, the time to perform an all-to-all personalized communication is dominated by the communication time.

3.5.2 Cut-Through Routing

Ring and Mesh

Recall the all-to-all personalized communication procedure described in Section 3.5.1 for a p-processor ring. We know that each processor sends $m(p-1)$ words of data because it has an m-word packet for every other processor. Assume that all messages are sent either clockwise or counterclockwise. The average distance that an m-word packet travels is $(\Sigma_{i=1}^{p-1} i)/(p-1)$, which is equal to $p/2$. Since there are p processors, each performing the same type of communication, the total traffic (the total number of data words transferred between directly-connected processors) on the network is $m(p-1) \times p/2 \times p$. The total number of communication channels in the network to share this load is p. Hence, the communication time for this operation is at least $(t_w \times m(p-1)p^2/2)/p$, which is equal to $t_w m(p-1)p/2$. Ignoring the message startup time t_s, this is exactly the time taken by the ring procedure. Hence, this procedure cannot be improved by using CT routing. Similarly, regardless of the routing mechanism, the mesh procedure of Section 3.5.1 is optimal within a small constant factor (Problem 3.12). However, with CT routing, the ring and mesh procedures for all-to-all personalized communication shown in Figures 3.18 and 3.19 do not require wraparound connections, provided that communication channels are bidirectional. Thus, with CT routing, the times for all-to-all personalized communication on a linear array and a mesh without wraparound are the same as those with SF routing on a ring and a mesh with wraparound, respectively.

Hypercube

Interestingly, using CT (instead of SF) routing does improve the performance of all-to-all personalized communication on a hypercube. The average distance between any two processors on a hypercube is $(\log p)/2$; hence, the total traffic is $m(p-1) \times (\log p)/2 \times p$. Since there is a total of $(p \log p)/2$ links in the hypercube network, the lower bound on the all-to-all personalized communication time is

$$\begin{aligned} T_{all\_to\_all\_pers}^{lower\_bound} &= \frac{t_w m(p-1)(p\log p)/2}{(p\log p)/2} \\ &= t_w m(p-1). \end{aligned}$$

This is smaller than the communication time of $(t_s + t_w mp/2)\log p$ for the hypercube procedure described in Section 3.5.1.

An all-to-all personalized communication effectively results in all pairs of processors exchanging some data. If cut-through routing is available on a hypercube, then the best way to perform this exchange is to have every pair of processors communicate directly with each other. Thus, each processor simply performs $p-1$ communication steps,

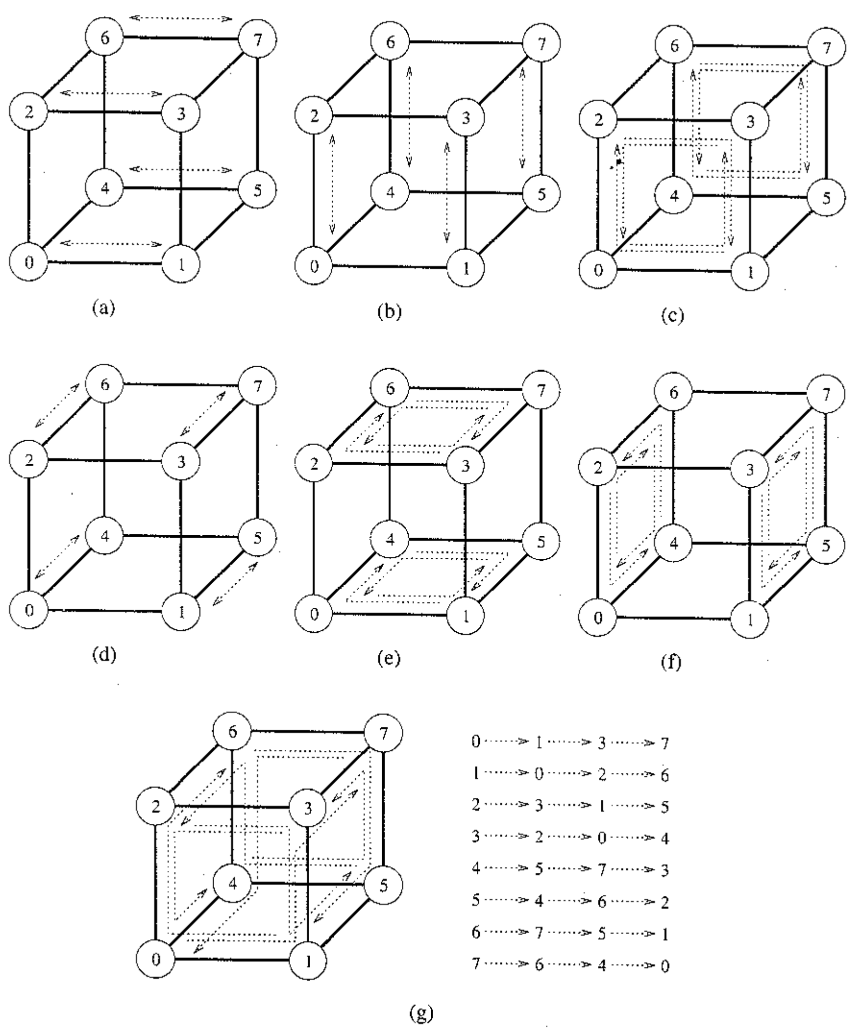


Figure 3.21 Seven steps in all-to-all personalized communication on an eight-processor hypercube with CT routing.

```
procedure ALL_TO_ALL_PERSONAL($d$, $my\_id$)
begin
    for $i := 1$ to $2^d - 1$ do
    begin
        $partner := my\_id$ XOR $i$;
        send $M_{my\_id, partner}$ to $partner$;
        receive $M_{partner, my\_id}$ from $partner$;
    endfor;
end ALL_TO_ALL_PERSONAL
```

Program 3.8 A procedure to perform all-to-all personalized communication on a d-dimensional hypercube with CT routing. The message $M_{i,j}$ initially resides on processor i and is destined for processor j.

exchanging m words of data with a different processor in every step. A processor must choose its communication partner in each step so that the hypercube links do not suffer congestion. Figure 3.21 shows one such congestion-free schedule for pairwise exchange of data in a three-dimensional hypercube. As the figure shows, in the j^{th} communication step, processor i exchanges data with processor (i XOR j). For example, in part (a) of the figure (step 1), the labels of communicating partners differ in the least significant bit. In part (g) (step 7), the labels of communicating partners differ in all the bits, as the binary representation of seven is 111. In this figure, all the paths in every communication step are congestion-free, and none of the bidirectional links carry more than one message in the same direction. This is true in general for a hypercube of any dimension. If the messages are routed appropriately, a congestion-free schedule exists for the $p - 1$ communication steps of all-to-all personalized communication on a p-processor hypercube.

Recall from Section 2.4.1 that a message traveling from processor i to processor j on a hypercube must pass through at least l links, where l is the Hamming distance between i and j (that is, the number of nonzero bits in the binary representation of (i XOR j)). A message traveling from processor i to processor j traverses links in l dimensions (corresponding to the nonzero bits in the binary representation of (i XOR j)). Although the message can follow one of the several paths of length l that exist between i and j (assuming $l > 1$), a distinct path is obtained by sorting the dimensions along which the message travels in ascending order. According to this strategy, the first link is chosen in the dimension corresponding to the least significant nonzero bit of (i XOR j), and so on. This routing scheme is known as ***ascending routing*** or ***E-cube routing***. A more detailed description of E-cube routing can be found in Section 2.6. By using E-cube routing, and by choosing communication pairs according to Program 3.8, a communication time of $t_s + t_w m + t_h l$ is guaranteed for a message transfer between processor i and processor j, where l is the Hamming distance between i and j. For a given i, on a p-processor hypercube, the sum of all l for $0 \leq j < p$ is $(p \log p)/2$. The total communication time for the entire operation is

$$T_{all\_to\_all\_pers} = (t_s + t_w m)(p - 1) + \frac{1}{2} t_h p \log p. \qquad (3.16)$$

A comparison of Equations 3.15 and 3.16 shows the term associated with t_s is higher for the CT routing procedure, while the term associated with t_w is higher for the SF routing procedure by a factor of almost $(\log p)/2$. Furthermore, CT routing obviates the need for local rearrangement of messages required in the SF routing procedure. For small messages, the startup time may dominate, and the procedure of Section 3.5.1 may still be useful.

3.6 Circular Shift

A ***permutation*** is a simultaneous, one-to-one data redistribution operation in which each processor sends a packet of m words to a unique processor. In this section, we discuss a particular type of permutation called circular shift. We define a ***circular q-shift*** as the operation in which processor i sends a data packet to processor $(i + q) \bmod p$ in a p-processor ensemble $(0 < q < p)$. The shift operation finds application in some matrix computations and in string and image pattern matching.

Since the implementation of a circular q-shift is fairly intuitive on a ring (it can be performed by $\min\{q, p - q\}$neighbor-to-neighbor communications in one direction), we discuss this operation in detail only on a mesh and a hypercube.

3.6.1 Store-and-Forward Routing

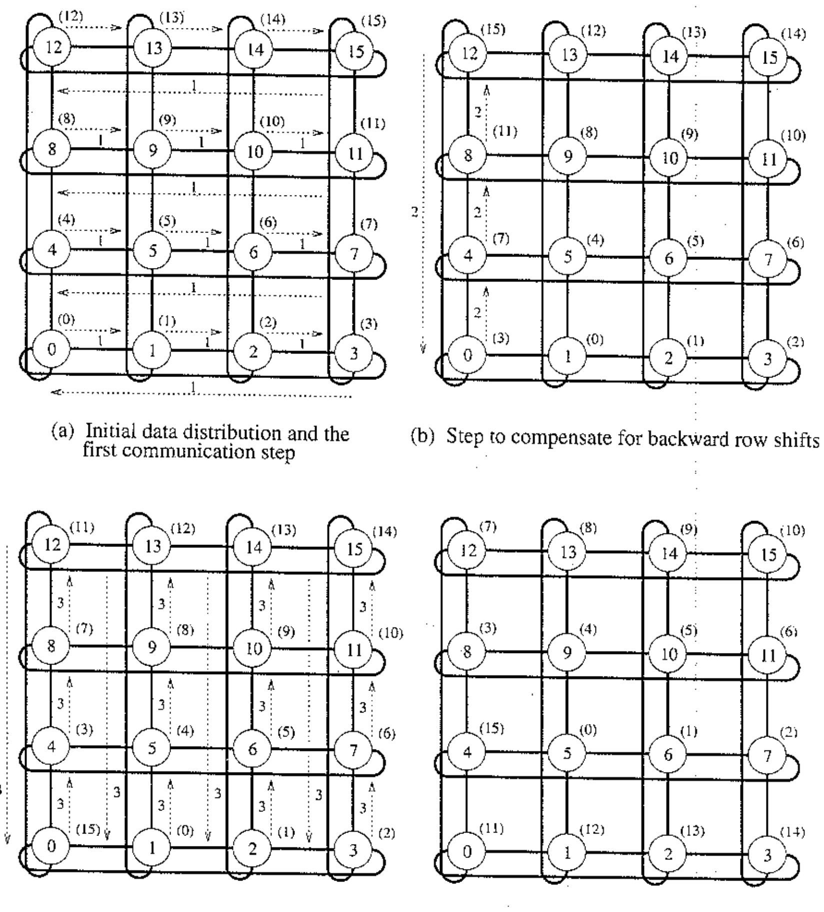
Mesh

If the processors of the mesh have row-major labels, a circular q-shift can be performed on a p-processor square wraparound mesh in two stages. First, the entire set of data is shifted simultaneously by $(q \bmod \sqrt{p})$ steps along the rows. Then it is shifted by $\lfloor q/\sqrt{p} \rfloor$ steps along the columns. During the circular row shifts, some of the data traverse the wraparound connection from the highest to the lowest labeled processors of the rows. All such data packets must shift an additional step forward along the columns to compensate for the $\sqrt{p}$ distance that they lost while traversing the backward edge in their respective rows.

Figure 3.22 shows a circular 5-shift on a 16-processor mesh. It requires one row shift, a compensatory column shift, and finally one column shift. In practice, we can chose the direction of the shifts in both the rows and the columns to minimize the number of steps in a circular shift. For instance, a 3-shift on a 4 × 4 mesh can be performed by a single backward row shift. Using this strategy, the number of unit shifts in a direction cannot exceed $\lfloor \sqrt{p}/2 \rfloor$.

Taking into account the compensating column shift for some packets, the total time for any circular q-shift on a p-processor mesh using packets of size m has an upper bound of

$$T_{circular\_shift} = (t_s + t_w m)(2\lfloor \frac{\sqrt{p}}{2} \rfloor + 1).$$

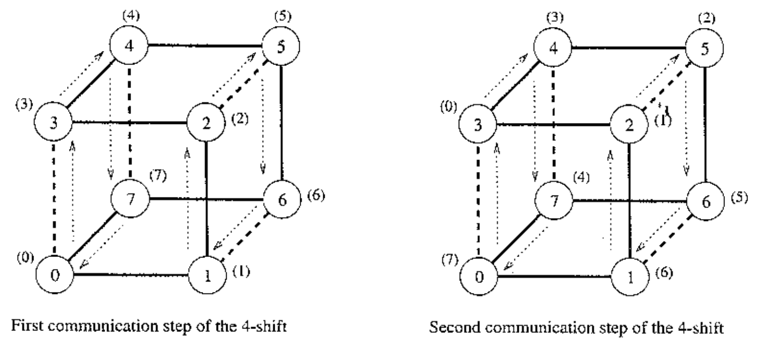


(a) Initial data distribution and the first communication step
(b) Step to compensate for backward row shifts
(c) Column shifts in the third communication step
(d) Final distribution of the data

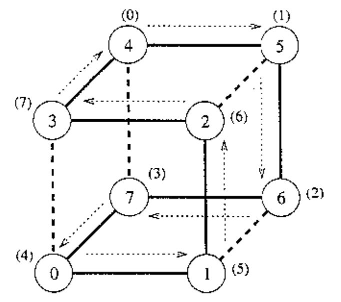
Figure 3.22 The communication steps in a circular 5-shift on a 4 × 4 mesh.

Hypercube

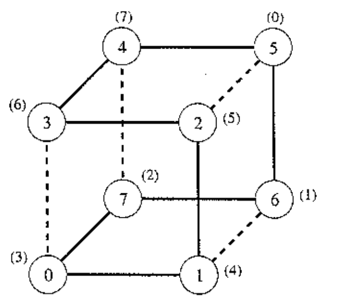
In developing a hypercube algorithm for the shift operation, we map a ring with 2^d processors onto a d-dimensional hypercube. We do this by assigning processor i of the ring to processor j of the hypercube such that j is the d-bit binary reflected Gray code (RGC) of i. Figure 3.23 illustrates this mapping for eight processors. A property of this mapping is that any two processors at a distance of 2^i on the ring are separated by exactly two links on


First communication step of the 4-shift
Second communication step of the 4-shift

(a) The first phase (a 4-shift)



(b) The second phase (a 1-shift)



(c) Final data distribution after the 5-shift

Figure 3.23 The mapping of an eight-processor ring onto a three-dimensional hypercube to perform a circular 5-shift as a combination of a 4-shift and a 1-shift.

the hypercube. An exception is $i = 0$ (that is, directly-connected processors on the ring) when only one hypercube link separates the two processors.

To perform a q-shift, we expand q as a sum of distinct powers of two. The number of terms in the sum is the same as the number of ones in the binary representation of q. For example, the number five can be expressed as $2^2 + 2^0$. These two terms correspond to bit positions 0 and 2 in the binary representation of five, which is 101. If q is the sum of s distinct powers of two, then the circular q-shift on a hypercube is performed in s phases.

In each phase of communication, all data packets move closer to their respective destinations by short cutting the ring (mapped onto the hypercube) in leaps of the powers of two. For example, as Figure 3.23 shows, a 5-shift is performed by a 4-shift followed by a 1-shift. The number of communication phases in a q-shift is exactly equal to the number of ones in the binary representation of q. Each phase consists of two communication steps,

except the 1-shift, which, if required (that is, if the least significant bit of q is one), consists of a single step. For example, in a 5-shift, the first phase of a 4-shift (Figure 3.23(a)) consists of two steps and the second phase of a 1-shift (Figure 3.23(b)) consists of one step. Thus, the total number of steps for any q in a p-processor hypercube is at most $2 \log p - 1$.

All communications in a given time step are congestion-free. This is ensured by the property of the ring mapping that all processors whose mutual distance on the ring is a power of two are arranged in disjoint subrings on the hypercube. Thus, all processors can freely communicate in a circular fashion in their respective subrings. This is shown in Figure 3.23(a), in which processors labeled 0, 3, 4, and 7 form one subring and processors labeled 1, 2, 5, and 6 form another subring.

The upper bound on the total communication time for any shift of m-word packets on a p-processor hypercube is

$$T_{circular\_shift} = (t_s + t_w m)(2 \log p - 1). \tag{3.17}$$

We can reduce this upper bound to $(t_s + t_w m) \log p$ by performing both forward and backward shifts (Problem 3.29). For example, on eight processors, a 6-shift can be performed by a single backward 2-shift instead of a forward 4-shift followed by a forward 2-shift.

3.6.2 Cut-Through Routing

Cut-through routing does not aid a shift operation on a ring or a mesh due to congestion on communication links. On a hypercube, however, CT routing can improve the time of a shift operation by almost a factor of $\log p$ for large messages. To perform a circular q-shift on hypercube with CT routing, the standard hypercube labeling of processors is used (instead of the RGC labeling used with SF routing). Each processor directly sends the data to be shifted to its destination processor. If the E-cube routing described in Section 3.5.2 is used, then each message has a congestion-free path (Problem 3.30). Figure 3.24 illustrates the non-conflicting paths of all the messages in circular q-shift operations for $1 \le q < 8$ on an eight-processor hypercube. In a circular q-shift on a p-processor hypercube, the longest path contains $\log p - \gamma(q)$ links, where $\gamma(q)$ is the highest integer j such that q is divisible by 2^j (Problem 3.31). Thus, the total communication time for messages of length m is

$$T_{circular\_shift} = t_s + t_w m + t_h(\log p - \gamma(q)). \tag{3.18}$$

For large messages, this time is approximately equal to $t_s + t_w m$.

3.7 Faster Methods for Some Communication Operations

So far in this chapter, we have derived procedures for various communication operations and their communication times under certain assumptions. We now briefly discuss the impact of relaxing these assumptions on some of the communication operations.

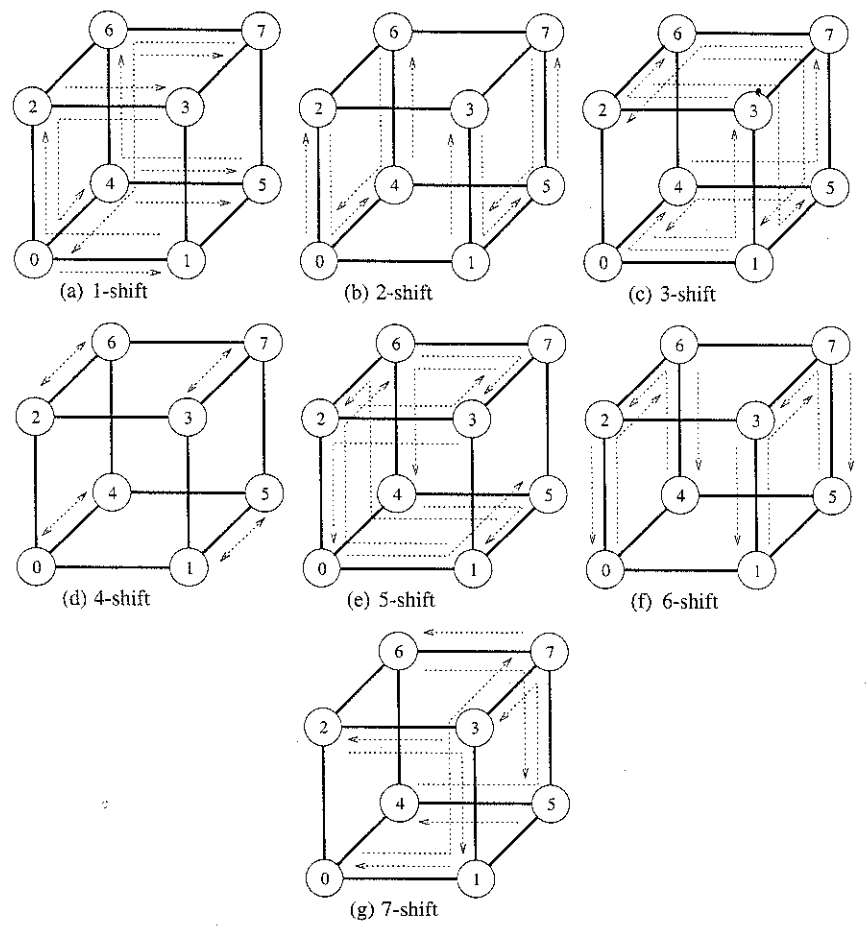

(a) 1-shift (b) 2-shift (c) 3-shift (d) 4-shift (e) 5-shift (f) 6-shift (g) 7-shift

Figure 3.24 Circular q-shifts on an 8-processor hypercube for $1 \le q < 8$.

★ 3.7.1 Routing Messages in Parts

In the procedures described in Sections 3.1–3.6, we assumed that an entire m-word packet of data travels between the source and the destination processors along the same path. If we split a message into smaller parts and then route these parts through different paths, we may be able to utilize the communication network better. For example, consider the transfer of a message of size m between two processors of a p-processor hypercube. Section 3.1 shows that this communication takes at most $t_s + t_w m \log p$ time with SF routing. One of the properties of a p-processor hypercube is that there are $\log p$ distinct paths between any pair

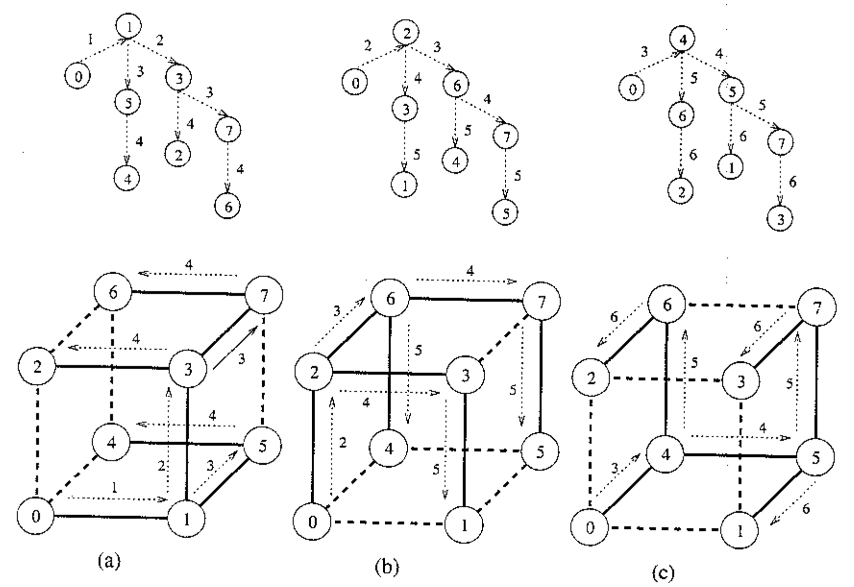

(a) (b) (c)

Figure 3.25 The six time-steps in one-to-all broadcast on an eight-processor hypercube with SF routing when the message is split into three parts that are routed separately on three different spanning binomial trees.

of processors. If the labels of two processors differ in l bits, then l of these paths contain l links each, and the remaining $(\log p - l)$ paths contain $l+2$ links each (Problem 2.8). If the message is split at the source into $\log p$ parts and each part is sent to the destination along a separate path (starting with the longer paths first), then the destination can receive the entire data in at most $2 \log p$ communication steps involving messages of size $m/\log p$. In $\log p$ steps, the source sends the smaller packets out on all the $\log p$ paths. The last packet takes at most $\log p$ steps to reach the destination. That way, the communication time is at most $2(t_s \log p + t_w m)$. This time reflects an improvement by a factor of $\Theta(\log p)$ in the t_w term over the method described in Section 3.1. Although, the t_s term increases by a similar factor, for sufficiently large messages, this method could still be faster than sending the entire message along the same path.

Now consider one-to-all broadcast on a hypercube. We first describe a property of the hypercube network that is useful for performing this operation. A ***spanning tree*** of a graph is defined as a tree whose set of nodes or vertices is identical to that of the graph. A one-node ***binomial tree*** is the node itself. A p-node binomial tree is constructed from two $p/2$-node trees by adding an edge from the root of one tree to the root of the second tree—making the second tree a subtree of the first. It is a property of a p-processor hypercube that a p-node binomial tree can be embedded into it with each node of the tree mapped onto a distinct processor. Thus, this binomial tree is also a spanning tree of the hypercube. Moreover, it is possible to construct $\log p$ different spanning binomial trees rooted at each of the $\log p$ neighbors of any given processor in a p-processor hypercube.

For performing one-to-all broadcast, we consider a hypothetical spanning binomial tree rooted at each of the neighbors of the source of the broadcast. Figure 3.25 shows such spanning trees for a three-dimensional hypercube with processor 0 as the source. These three trees are rooted at the three neighbors of processor 0—processors 1, 2, and 4. Moreover, these trees are oriented so that the source itself is the smallest subtree (ontaining a single node) of each binomial tree. In order to be broadcast, the m-word message is first split into $\log p$ parts at the source processor. The source sends one of these parts to the root of each spanning tree in three consecutive steps. Each processor (including the root) of every spanning tree stores any message that it receives, and sends it out to all of its subtrees in the order of decreasing sizes of the subtrees. Figure 3.25 shows that there is never a conflict between two messages traveling in the same direction on any channel in the same time step.

The source sends out the $\log p$ messages sequentially. It takes another $\log p$ steps for the message sent to the last spanning tree to percolate down to all the leaves of the tree. Since all processors lie on each spanning tree (by definition) and each tree carries one of the $\log p$ parts of the original message, all the processors receive the complete message by the end of the procedure. With individual messages of size $m/\log p$, the time taken to complete all the $2\log p$ steps of the broadcast is

$$\begin{aligned} T_{one\_to\_all} &= (t_s + t_w m/\log p) \times 2\log p \\ &= 2(t_s \log p + t_w m). \end{aligned} \tag{3.19}$$

Note that the t_w term is reduced by a factor of $(\log p)/2$ over the algorithm presented in Section 3.2, but the t_s term has doubled.

In this section, we discussed how the communication time of one-to-all broadcast can be reduced by splitting a message into smaller parts that are routed independently. Another algorithm to perform one-to-all broadcast on a hypercube in time $2(t_s \log p + t_w m)$ is given in Problem 3.24. However, algorithms like the one presented here and in Problem 3.24 are usually difficult to program and incur additional overhead in breaking, routing, queuing, and reassembling the messages. Moreover, the original message must contain at least $\log p$ words for successful splitting. In practice, messages need to be even longer in order to offset the doubled startup cost. Hence, the smarter broadcast algorithms are useful only if the message size if sufficiently large.

3.7.2 All-Port Communication

In a parallel architecture, a single processor may have multiple communication ports with links to other processors in the ensemble. For example, each processor in a two-dimensional wraparound mesh has four ports, and each processor in a d-dimensional hypercube has d ports. In this book, we generally assume what is known as the ***one-port communication*** model. In one-port communication, a processor can send data on only one of its ports at a time. Similarly, a processor can receive data on only one port at a time. However, a processor can send and a receive data simultaneously—either on the same port or on separate ports. In contrast to the one-port model, an ***all-port communication*** model permits simultaneous communication on all the channels connected to a processor.

On a p-processor hypercube with all-port communication, the coefficients of t_w in the expressions for the communication times of one-to-all and all-to-all broadcast and personalized communication are all smaller than their one-port counterparts by a factor of $\log p$. Since the number of channels per processor for a ring or a mesh is constant, all-port communication does not provide any asymptotic improvement in communication time on these architectures.

Despite the apparent speedup, the all-port communication model has certain limitations. For instance, not only is it difficult to program, but it requires that the messages are large enough to be split efficiently among different channels. In several parallel algorithms, an increase in the size of messages means a corresponding increase in the granularity of computation at the processors. When the processors are working with large data sets, the interprocessor communication time is dominated by the computation time if the computational complexity of the algorithm is higher than the communication complexity. For example, in the case of matrix multiplication, there are n^3 computations for n^2 words of data transferred among the processors. If the communication time is a small fraction of the total parallel run time, then improving the communication by using sophisticated techniques is not very advantageous in terms of the overall run time of the parallel algorithm.

Even with today's technology, the one-port communication model is quite relevant. In some state-of-the-art parallel computers such as the CM-5, the all-port model is not applicable at all. The CM-5 can execute most one-port hypercube algorithms without a substantial extra communication penalty. Unlike a real hypercube, each CM-5 processor has only one communication port because its interconnection network is a pseudo fat tree. In any case, even if multiple ports are available, all-port communication can be effective only if data can be fetched and stored in memory at a rate sufficient to sustain all the parallel communication. For example, to utilize all-port communication effectively on a p-processor hypercube, the memory bandwidth must be greater than the communication bandwidth of a single channel by a factor of at least $\log p$; that is, the memory bandwidth must increase with the number of processors to support simultaneous communication on all ports.

3.7.3 Special Hardware for Global Operations

In addition to the standard data network, some parallel computers have a fast control network that can perform certain global operations in a small constant time. One such operation commonly implemented using special hardware is reduction (Example 3.7). A reduction operation starts with a different value on each processor and ends with a single value on each processor. The final value is the result of applying an associative operator

Table 3.1 Summary of communication times of various operations discussed in Sections 3.2–3.5 on different architectures with one-port communication and CT routing. The message size for each operation is m and the number of processors is p. The time for one-to-all broadcast on the hypercube is not optimal, and, as shown in Section 3.7.1 and Problem 3.24, can be improved to $2(t_s \log p + t_w m)$. In the hypercube expression for circular q-shift, $\gamma(q)$ is the highest integer j such that q is divisible by 2^j.

Operation	Ring	2-D Mesh (wraparound, square)	Hypercube
One-to-all broadcast	$(t_s + t_w m)\log p$ $+t_h(p-1)$	$(t_s + t_w m)\log p$ $+2t_h(\sqrt{p}-1)$	$(t_s + t_w m)\log p$
All-to-all broadcast	$(t_s + t_w m)(p-1)$	$2t_s(\sqrt{p}-1) + t_w m(p-1)$	$t_s \log p + t_w m(p-1)$
One-to-all personalized	$(t_s + t_w m)(p-1)$	$2t_s(\sqrt{p}-1) + t_w m(p-1)$	$t_s \log p + t_w m(p-1)$
All-to-all personalized	$(t_s + t_w mp/2)(p-1)$	$(2t_s + t_w mp)(\sqrt{p}-1)$	$(t_s + t_w m)(p-1)$ $+(t_h/2)p\log p$
Circular q-shift	$(t_s + t_w m)\lfloor p/2\rfloor$	$(t_s + t_w m)(2\lfloor\sqrt{p}/2\rfloor + 1)$	$t_s + t_w m$ $+t_h(\log p - \gamma(q))$

(such as addition, maximum, minimum, or a logical bitwise operator) on all the starting values.

A fast, (almost) constant time reduction, while providing a natural way to implement accumulation, can also be used to perform broadcasts. If the source starts with a datum to be broadcast, and every other processor starts with a zero, a reduction with addition as the associative operator results in the distribution of the source's datum to all the processors. If t_r is the time to perform one reduction, then the control network provides a fast means to implement one-to-all broadcast of a message of size m in $t_r m$ time, as opposed to $(t_s + t_w m)\log p$ time using the hypercube algorithm described in Section 3.2.

3.8 Summary

Table 3.1 summarizes the communication times for the operations discussed in this chapter on ring, mesh, and hypercube architectures with cut-through routing. Most of the entries in the table are valid for store-and-forward routing as well. The exceptions are: (1) the communication times of one-to-all broadcast on a ring and a mesh with SF routing,

which are $(t_s + t_w m)\lceil p/2\rceil$ and $2(t_s + t_w m)\lceil\sqrt{p}/2\rceil$, respectively; and (2) the time taken by all-to-all personalized communication operation on a hypercube with SF routing, which is $(t_s + t_w mp/2)\log p$. The time for one-to-all broadcast on the hypercube is not optimal; as we saw in Section 3.7.1, it can be improved to $2(t_s \log p + t_w m)$.

All communication patterns discussed in this chapter are very regular and predictable. Therefore, we have been able to describe their algorithms in terms of discrete time steps, avoiding temporal and spatial (on the channels) overlap of messages. As a result, all the algorithms described here will work as expected on SIMD computers. However, since it is theoretically impossible to impose any synchrony on the processors of an MIMD computer, the communication times may deviate somewhat from their theoretical expressions, especially if communication and computation are interspersed.

3.9 Bibliographic Remarks

In this chapter, we studied a variety of data communication operations for the ring, mesh, and hypercube interconnection topologies. Saad and Schultz [SS89b] discuss implementation issues for these operations on these and other architectures, such as shared-memory and a switch or bus interconnect.

The hypercube algorithm for a certain communication operation is often the best algorithm for other less-connected architectures too, if they support cut-through routing. Due to the versatility of the hypercube architecture and the wide applicability of its algorithms, extensive work has been done on implementing various communication operations on hypercubes [BOS⁺91, BR90, BT89, FF86, JH89, Joh90, MdV87, RS90, SS89a, SW87]. The properties of a hypercube network that are used in deriving the algorithms for various communication operations on it are described by Saad and Schultz [SS88].

The all-to-all personalized communication problem in particular has been analyzed for the hypercube architecture by Boppana and Raghavendra [BR90], Johnsson and Ho [JH91], Seidel [Sei89], and Take [Tak87]. Ascending or E-cube routing that guarantees congestion-free communication in Program 3.8 for all-to-all personalized communication is described by Nugent [Nug88], and is used in Intel's iPSC/2 hypercube.

The reduction and the prefix sums algorithms of Examples 3.7 and 3.8 are described by Ranka and Sahni [RS90]. Our discussion of the circular shift operation is adapted from Bertsekas and Tsitsiklis [BT89].

The hypercube algorithm for one-to-all broadcast using spanning binomial trees is described by Bertsekas and Tsitsiklis [BT89] and Johnsson and Ho [JH89]. In the spanning tree algorithm described in Section 3.7.1, we split the m-word message to be broadcast into $\log p$ parts of size $m/\log p$ for ease of presenting the algorithm. Johnsson and Ho [JH89] show that the optimal size of the parts is $\lceil(\sqrt{t_s m/t_w \log p})\rceil$. In this case, the number of messages may be greater than $\log p$. These smaller messages are sent from the root of the spanning binomial tree to its $\log p$ subtrees in a circular fashion. With this strategy, one-to-all broadcast on a p-processor hypercube can be performed in time $t_s \log p + t_w m + 2t_w\lceil(\sqrt{t_s m/t_w \log p})\rceil \log p$.

Algorithms using the all-port communication model have been described for a variety of communication operations on the hypercube architecture by Bertsekas and Tsitsiklis [BT89], Johnsson and Ho [JH89], Ho and Johnsson [HJ87], Saad and Schultz [SS89a], and Stout and Wagar [SW87]. Johnsson and Ho [JH89] show that on a p-processor hypercube with all-port communication, the coefficients of t_w in the expressions for the communication times of one-to-all and all-to-all broadcast and personalized communication are all smaller than their one-port counterparts by a factor of $\log p$. Gupta and Kumar [GK91] show that all-port communication may not improve the scalability of an algorithm on a parallel architecture over one-port communication.

The network architecture of CM-5 that supports a fast reduction operation is described by Leiserson et al. [L$^+$92]. The same operation is discussed by Stolfo and Miranker [SM86] in the context of the DADO parallel computer. Besides reduction, parallel computers like the CM-5 and DADO also support other related operations such as prefix sums. A generalized form of prefix sums, often referred to as ***scan***, has been used by some researchers as a basic primitive in data-parallel programming. Blelloch [Ble90] define a ***scan vector model***, and describes how a wide variety of parallel programs can be expressed in terms of the scan primitive and its variations.

The elementary operations described in this chapter are not the only ones used in parallel applications. A variety of other useful operations for parallel computers have been described in literature, including selection [Akl89], pointer jumping [HS86, Jaj92], BPC permutations [Joh90, RS90], fetch-and-op [GGK$^+$83], packing [Lev87, Sch80], bit reversal [Loa92], and keyed-scan or multi-prefix [Ble90, Ran89].

Sometimes data communication does not follow any predefined pattern, but is arbitrary, depending on the application. In such cases, a simplistic approach of routing the messages along the shortest data paths between their respective sources and destinations leads to contention and imbalanced communication. Leighton, Maggs, and Rao [LMR88], Valiant [Val82], and Valiant and Brebner [VB81] discuss efficient routing methods for arbitrary permutations of messages.

Problems

3.1 **(One-to-all broadcast on a tree)** Show that one-to-all broadcast of an m-word message can be performed in time $(t_s + t_w m + t_h(\log p + 1))\log p$ on a balanced binary tree on which each of the p leaves is a processor and each intermediate node is a switching node. Assume that a message takes time $t_s + t_w m + t_h l$ to traverse a path with $l - 1$ switching nodes.

3.2 Consider a linear array (without a wraparound connection) of p processors labeled from 0 to $p-1$. The average distance (in terms of the number of links) from processor 0 to any of the other $p-1$ processors is $(\Sigma_{i=1}^{p-1} i)/(p-1)$, which is equal to $p/2$. Derive an expression for the average distance to any of the (four) corner processors from all the other processors in a $\sqrt{p} \times \sqrt{p}$ mesh without wraparound

connections. What is the average distance of a processor in a d-dimensional hypercube from the other processors?

3.3 Describe a procedure for all-to-all personalized communication of m-word messages on a ring of p processors with SF routing such that the procedure takes $t_s(p-1) + t_w m p^2/4$ time if p is even and $t_s(p-1) + t_w m(p^2-1)/4$ time if p is odd.

3.4 **(All-to-all broadcast on a tree)** Given a balanced binary tree as shown in Figure 3.8, describe a procedure to perform all-to-all broadcast that takes $(t_s + t_w mp/2)\log p$ time for m-word messages on p processors. Assume that only the leaves of the tree contain processors, and that an exchange of two m-word messages between any two processors connected by bidirectional channels takes $t_s + t_w mk$ time if the communication channel (or a part of it) is shared by k simultaneous messages.

3.5 Derive an optimal algorithm along the lines of Example 3.7 for adding p numbers on a p-processor mesh and distributing the sum to all the processors. What is its parallel run time? Show that your algorithm is optimal.

3.6 **(One-to-all personalized communication on a ring and a mesh)** Give the procedures and their communication times for one-to-all personalized communication of m-word messages on p processors for the ring and the mesh architectures.
Hint: For the mesh, the algorithm proceeds in two phases as usual and starts with the source distributing pieces of $m\sqrt{p}$ words among the $\sqrt{p}$ processors in its row such that each of these processors receives the data meant for all the $\sqrt{p}$ processors in its column.

3.7 Section 3.2.1 shows informally that the hypercube algorithm described in that section for one-to-all broadcast is optimal if an entire message is routed along the same path. Why can't the same argument be applied to the hypercube algorithm for all-to-all personalized communication described in Section 3.5.1?

3.8 **(Multinode accumulation)** The dual of all-to-all broadcast is multinode accumulation, in which each processor is the destination of a single-node accumulation. For example, consider the scenario where p processors have a vector of p elements each, and the i^{th} processor (for all i such that $0 \le i < p$) gets the sum of the i^{th} elements of all the vectors. Describe an algorithm to perform multinode accumulation on a hypercube with addition as the associative operator. If each message contains m words and t_{add} is the time to perform one addition, how much time does your algorithm take (in terms of m, p, t_{add}, t_s and t_w)?
Hint: In all-to-all broadcast, each processor starts with a single message and collects p such messages by the end of the operation. In multinode accumulation, each processor starts with a p distinct messages (one meant for each processor) but ends up with a single message.

3.9 Parts (c), (e), and (f) of Figure 3.21 show that for any processor in a three-dimensional hypercube, there are exactly three processors whose shortest distance from the processor is two links. Derive an exact expression for the number of