

**A Survey of Collective Communication in  
Wormhole-Routed Massively Parallel Computers**

*Philip K. McKinley, Yih-jia Tsai, and David F. Robinson*

Technical Report  
MSU-CPS-94-35  
June 1994



Submitted for publication, June 1994.

# A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers \*

*Philip K. McKinley, Yih-jia Tsai, and David F. Robinson*

Department of Computer Science  
Michigan State University  
East Lansing, Michigan 48824  
{mckinley, tsaiyi, robinsod}@cps.msu.edu

June 1994

## Abstract

Massively parallel computers (MPC) are characterized by the distribution of memory among an ensemble of nodes. Since memory is physically distributed, MPC nodes communicate by sending data through a network. In order to program an MPC, the user may directly invoke low-level message passing primitives, may use a higher-level communications library, or may write the program in a data parallel language and rely on the compiler to translate language constructs into communication operations. Whichever method is used, the performance of communication operations directly affects the total computation time of the parallel application. Communication operations may be either *point-to-point*, which involves a single source and a single destination, or *collective*, in which more than two processes participate.

This paper discusses the design of collective communication operations for current systems that use the wormhole routing switching strategy, in which messages are divided into small pieces and pipelined through the network. Compared to the *store-and-forward* switching method that was used in early multicomputers, wormhole routing often reduces the effect of path length on communication latency. Over the past several years, a number of researchers have exploited this property in the design of new collective communication algorithms, which differ fundamentally from their store-and-forward predecessors. This paper discusses the significant issues involved in wormhole-routed collective communication and presents the major classes of solutions that have been proposed to address the problem.

**Keywords:** Collective communication, parallel processing, wormhole routing, distributed memory, massively parallel computers.

---

\*This work was supported in part by the NSF grants MIP-9204066, CDA-9121641, CDA9222901, by DOE grant DE-FG02-93ER25167, and by an Ameritech Faculty Fellowship.

# 1 Introduction

A recent trend in supercomputing has been towards the use of parallel processing to solve computationally-intensive problems. Several so-called *scalable* parallel architectures, which offer corresponding increases in performance as the number of processors is increased, have been designed in the last few years. *Massively parallel computers* (MPCs) are characterized by the distribution of memory among an ensemble of computing nodes. Many such systems interconnect nodes through a *direct network*, in which each node has a connection to a set of other nodes, called neighbors.

Since memory is physically distributed, MPC nodes communicate by sending messages through the network. Historically, the programmer of a distributed-memory system has invoked various system primitives to send messages among processes executing on different nodes, resulting in a *message-passing* program. While such low-level control over communication allows the user to exploit characteristics of the architecture, this type of programming is often tedious and error-prone. Furthermore, parallel software development has long been plagued by the large variety of parallel architectures available; such diversity often has implied that a new version of a particular algorithm had to be developed for each new architecture. One method that has been used to address these problems is to construct communication libraries [1], which hide the details of the underlying architecture and vendor-specific interfaces from the user but provide a common interface across multiple platforms, permitting user code to be more easily ported among machines. In order to further simplify the programmer's task and improve code portability, an alternative approach to parallel programming is to use a *data parallel language*, such as High Performance Fortran (HPF) [2], which provides the user with control over data alignment and realignment, but which hides the communication calls from the user. For a distributed-memory system, the compiler for such a language must translate high-level data parallel language constructs into appropriate low-level communication primitives [3, 4].

Whether communication operations are programmed by the user, contained in a library, or generated by a compiler, their latency directly affects the total computation time of the parallel application. Communication operations may be either *point-to-point*, which involve a single source and a single destination, or *collective*, in which more than two processes participate. Collective communication operations are particularly important in scientific computing, where large data arrays are typically partitioned and distributed over the local memories of the nodes that are executing the program. In such applications, nodes use collective operations to distribute, gather, and exchange data, to perform global compute operations on distributed data, and to synchronize with one another at specific points in program flow. The growing interest in collective operations is evidenced by their inclusion in *Message Passing Interface* (MPI) [5], an emerging standard for communication routines used by message-passing programs, and by their increasing role in supporting various programming constructs in HPF [4]. A set of standard collective operations is reviewed in Section 2.

Efficient implementation of collective communication operations depends on the underlying ar-

chitecture of the MPC. While there has been little consensus on some aspects of communication architectures, such as network topology, there has been a good deal of agreement on the way in which messages are *switched* through the network. Specifically, many new generation MPCs employ *wormhole routing* [6], where each message is divided into small pieces that are pipelined through the network by way of *routers* at each node. Compared to the store-and-forward switching method that was used in early multicomputers, wormhole routing often reduces the effect of path length on communication latency [7]. In the absence of *channel contention*, which occurs when two messages simultaneously require the same channel, the measured latency of a wormhole-routed message has been shown to be nearly independent of the distance between the source and destination nodes [7]. However, in situations where multiple messages exist in the network concurrently, channel contention among those messages may be exacerbated by the use of wormhole routing, in which blocked messages hold some communication channels while waiting for other messages. The invocation of a collective operation, whose implementation may involve many messages, poses precisely such a situation.

This paper addresses the design of collective communication operations for wormhole-routed networks. Ni and McKinley [7] previously surveyed wormhole routing techniques for direct networks; that paper focused on switching architectures, performance comparisons with other switching strategies, deadlock prevention methods, and adaptive routing algorithms. In the years since that paper was originally written, a relatively large body of work has been published in the area of collective communication algorithms for wormhole-routed systems. Most notably, many new collective algorithms have been designed that exploit the relative distance-insensitivity of wormhole routing, differing fundamentally from their store-and-forward counterparts. Other architectural properties, such as the network topology and the number of ports connecting each node to the network, are also important to their design and performance. This paper is intended to present the main issues involved in that research and to describe the major classes of solutions that have been proposed. It is not, however, intended to be an exhaustive survey of the literature.

Following a review of collective communication operations, we briefly describe wormhole-routed architectures, focusing on those architectural characteristics that are particularly important to collective communication. We then devote a significant amount of discussion to the implementation of collective communication operations in software. These approaches are designed for systems that support only point-to-point, or *unicast*, communication in hardware. In these environments, collective operations must be implemented by sending multiple unicast messages; such implementations are called *unicast-based* [8]. We first describe the implementation of tree-like arrangements of messages in various network topologies; these structures are used in several of the “elementary” collective operations, such as broadcast and global compute operations. We next discuss the so-called “all-to-all” collective operations, in which many nodes are both sources and recipients of data. Finally, we describe how some collective operations may be supported directly in hardware. We conclude the paper with the discussion of several open issues in this area of research.

## 2 Collective Communication Operations

Collective operations are usually defined in terms of a group of processes. The operation is executed by having all processes in the group call the communication routine with matching parameters. The group, which may constitute all or a subset of the processes in the parallel application, is assumed to have been previously defined and is identified in one of the parameters to the collective operation [5]. Collective operations can operate synchronously or asynchronously, depending on whether a calling process can return before other processes in the group have called the routine.

**Definitions of Operations.** Collective operations may be used for process control, data movement, or global operations; examples are listed in Table 1. Data movement operations include *broadcast*, in which one process sends the same message to all other group members; *scatter*, in which one process sends a different message to each of a set of destinations; and *gather*, in which one process receives a message from each of a group of processes. These basic operations can be extended and combined to form more complex operations. In *all-to-all broadcast*, every process sends a message to all members of the group. In *all-to-all scatter-gather*, also referred to as *complete exchange*, every member of a group sends different data to every other node in the group. *Barrier synchronization* defines a logical point in the control flow of an algorithm at which all the members of the group must arrive before any of the processes in the subset is allowed to proceed further.

Table 1. Examples of collective communication primitives

<i>Category</i>	<i>Primitive</i>	<i>Description</i>
data movement	<b>broadcast</b>	one member sends same message to all members
	<b>scatter</b>	one member sends different message to each member
	<b>gather</b>	every member sends a message to a single member
	<b>all-to-all broadcast</b>	every member performs a <b>broadcast</b>
	<b>all-to-all scatter-gather</b>	every member performing <b>scatter</b>
process control	<b>barrier synchronization</b>	all members must reach point before any can proceed
global operation	<b>reduction</b>	perform a global operation on distributed data
	<b>scan (parallel prefix)</b>	“partial” <b>reduction</b> based upon relative process number

Global compute operations include both *reduction* and *scan* (also known as *parallel prefix*). In reduction, an associative and commutative operation is applied across data items from each member of the group. Example operations include sum, max, min, bitwise operations, and so on. In an  $N/1$  reduction operation, the resultant data resides at a single process, called the *root*. In an  $N/N$  reduction operation, every process involved in the operation obtains a copy of the reduced data. In scan operations, given processes  $P_0, P_1, \dots, P_n$  and data items  $d_0, d_1, \dots, d_n$ , an associative and commutative operator ‘ $*$ ’ is applied such that the result at process  $p_i$  is  $d_0 * d_1 * \dots * d_i$ .

Figure 1 depicts examples of collective operations among a group of four processes; the actual messages sent between processes may differ from those shown, depending on the underlying algorithm

and computing platform. Figures 1(a) through 1(e), which depict the data movement operations, are self-explanatory. Figure 1(f) illustrates the execution of barrier synchronization in a distributed memory environment. In this particular implementation, process  $P_0$  plays the role of a *barrier process*. In the first phase of the operation, each process that reaches the barrier sends a message indicating this fact to process  $P_0$ . As soon as  $P_0$  has received messages from all the other processes, it broadcasts a message to the group, indicating to each member that all processes have reached the barrier and that they may proceed. Figure 1(g) shows a special case of reduction (with a generic operator, denoted by ‘\*’) in which the result  $R$  of the operation resides at a single process, in this case  $P_0$ . In other cases, the result may be distributed to some or all of the processes involved. A scan operation, using the same operator ‘\*’, is shown in Figure 1(h).

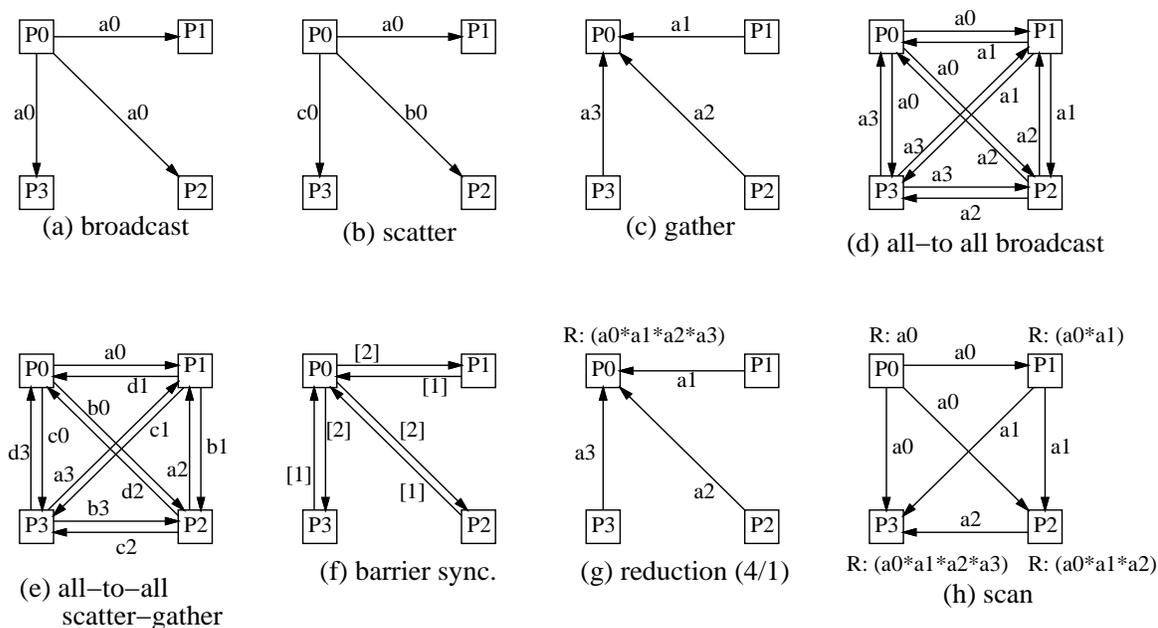


Figure 1. Semantics of various collective operations among four processes

It is important to distinguish between the *process view* and the *node view* of a collective operation. For example, when one node in a network sends a message to a proper subset of the nodes, this is usually referred to as *multicast* [8]. However, the MPI standard does not explicitly discuss multicast. Rather, MPI would describe multicast as a *broadcast* to a group of processes that happen to reside on only a proper subset of the nodes in the network. While defining collective operations in terms of process groups is very useful for studying their semantics, it is less appropriate for the study of their performance, which depends on the physical relationships between the group members and the system architecture. For example, the number of processes per node and the distribution of processes in the network affect the performance of collective operations executed within that group. Therefore, in this paper, we will discuss collective implementations in terms of the physical network architecture and the specific messages that constitute the operation.

**Uses of Collective Operations.** Collective communication is used in all three major methods used to program distributed-memory computers: message-passing, translation of shared-memory code to message-passing code, and execution of shared-memory code atop distributed shared-memory.

Collective communication operations are found in many algorithms designed for message-passing systems. In fact, it was their frequent use that led to their inclusion in several commercial communications libraries, and eventually to the standardization of their syntax and semantics in MPI [5]. Collective operations are used in numerous sorting, graph, and search algorithms [9]. Perhaps the largest class of message-passing applications that can take advantage of efficient collective operations is parallel numerical algorithms. Collective operations are used in a wide variety of matrix-related algorithms, including solving of linear systems, finding eigenvalues, and performing transform operations [9]. Many of these numerical algorithms have themselves been organized into libraries. For example, the ScaLAPACK project [10] is targeted to producing a distributed-memory version of LAPACK, a popular sequential library for problems in numerical linear algebra. In order to improve portability, ScaLAPACK uses a library called BLACS (Basic Linear Algebra Communication Subprograms) [11], which provides basic matrix-related communications operations. In turn, BLACS can be implemented using architecture-specific implementations of collective operations.

In spite of the presence of communication libraries and numerical libraries, many users prefer a shared-memory programming paradigm to a message-passing paradigm. A compiler translates a program written in a data parallel language into lower-level communication operations. Li and Chen [3] have studied this translation process. The communication operations generated in their approach include all those listed in Table 1, as well as others, such as permutation. More recently, a coalition of industrial and academic groups has standardized High Performance Fortran [2], which is designed for distributed-memory platforms. When an HPF program is compiled, many of those communication operations generated by the compiler may be collective in nature. For example, HPF contains so-called *intrinsic*s, which perform reduction and scan, rearranging and reshaping, and scatter and gather operations on data arrays. HPF also allows dynamic redistribution of arrays which, depending on their present distribution, may involve any of broadcast, scatter, gather, and their all-to-all counterparts. Furthermore, since entire arrays may be operated on without explicitly looping across the elements, even a statement as common as  $A = B + C$ , where  $A$ ,  $B$ , and  $C$  are arrays, may produce underlying reduction and scatter operations when compiled and executed.

Finally, it is possible to execute a shared-memory program directly atop a distributed-memory platform, so long as the system dynamically translates remote memory accesses into communication operations. The *distributed shared memory* paradigm provides a virtual address space that is shared among processes located on processing nodes with their own local memories. In order to improve the performance of local reads, most systems allow data to be replicated across multiple nodes, which implies that the system must take certain measures in order to maintain memory coherence. A wide variety of coherence protocols have been proposed [12]; many of them can be classified as either *write-*

*update* or *write-invalidate*. Both approaches rely on broadcast to send invalidations and updates, respectively. In addition to being able to access shared memory, the other main form of interaction among processes is synchronization, such as barriers and semaphores. Again, efficient collective communication operations can be used to implement these constructs so that their overhead is minimized, thereby improving performance.

### 3 Architectural Issues

Varying levels of support for collective operations may be provided in an MPC. In some systems, certain collective operations are supported directly in hardware, in which single dedicated instructions may be invoked by a process executing on a node. Many existing MPCs, however, support only unicast communication in hardware. In these environments, all communication operations must be implemented in software by sending one or more unicast messages; such implementations are called *unicast-based* [8]. Finally, operations may be *partially* supported in hardware. For example, a system may provide some elementary collective operations, such as multicast, in hardware, with several instances combined in software to implement a more complex operations. Whether implemented in hardware, software, or a combination of the two, the design and performance of collective operations is influenced by several characteristics of the system, as described below.

**Switching and Network Latency.** One of the most important architectural characteristics, which affects all types of communication, is the switching strategy, which determines how data is removed from one channel and placed on another channel along the path from source to destination. In store-and-forward switching, which was used in early hypercube systems, this task was relegated to the local processors along the path from the source to the destination. Each intermediate node received the entire message before forwarding it on towards the destination. In wormhole-routed systems, on the other hand, all such communication-related tasks are handled by a separate *router*, or *communications co-processor*, located at each node. As shown in Figure 2, the message is divided into small pieces, called *flits*, that are pipelined through the network by way of small buffers at the routers. The presence of routers allows any set of local processors to communicate among one another without affecting any of the local processors outside the set. As we shall see later, this capability allows much more flexibility than store-and-forward switching in the design of collective operations.

The switching strategy directly affects the *network latency*,  $t_n$ , which equals the elapsed time after the head of a (unicast) message has entered the network at the source until the tail of the message emerges from the network at the destination. In store-and-forward switched systems,  $t_n$  was linear in the path length between the source and destination. In wormhole-routed systems, network latency is given by  $t_p d + \ell t_f$ , where  $t_p$  is the delay at the individual nodes encountered on the path,  $d$  is the number of nodes traversed, or distance,  $\ell$  is the length of the message, and  $t_f$  is the time required to

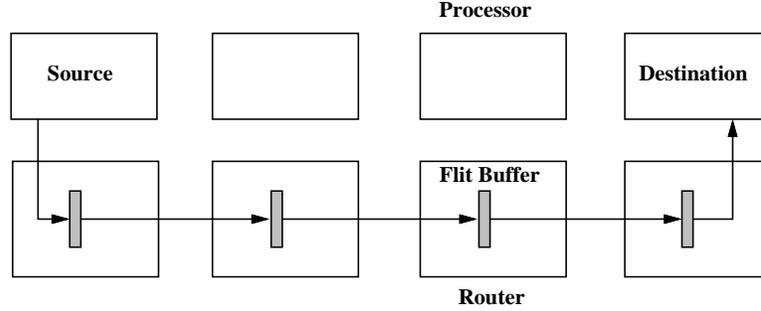


Figure 2. Pipelining operation of wormhole routing

transmit a flit between routers. For relatively long messages, the value of  $t_p d$  becomes small compared to that of  $l t_f$  [7].

The *start-up latency*,  $t_s$ , is the time required for the system to handle the message at both the source and destination nodes. Its value is mainly dependent on the design of system software and the interface between local processors and routers. Startup latency can be further decomposed into *sending latency*,  $t_{snd}$ , and *receiving latency*,  $t_{rcv}$ , the start-up latencies incurred at the sending node and the receiving node, respectively. For some systems,  $t_s$  may be an order of magnitude greater than  $t_n$  for small messages [8]. Thus, the latency of short messages is distance-insensitive due to startup latency, while the latency of long messages is distance-insensitive due to the pipelining effect of wormhole routing. Under these circumstances, all pairs of nodes can be considered to be essentially equidistant apart with respect to time, which allows a great deal of flexibility in the definition and scheduling of constituent messages of collective operations. Therefore, when evaluating the performance of collective operations in this paper, we will often ignore the contribution of  $t_p d$  to network latency.

Despite this property, a wormhole-routed network may not be modeled as a complete graph, especially in the context of collective communication, because messages sent concurrently through the network may contend for communication channels. Collective operations must be designed so that they not only minimize the number of message-passing steps, but also minimize, or preferably eliminate, contention among constituent messages. This task involves both the physical network topology and the underlying hardware routing algorithm.

**Network Topology.** Several MPC topologies are depicted in Figure 3. The topologies of many commercial MPCs are special cases of either  $n$ -dimensional meshes or  $k$ -ary  $n$ -cubes. These classes of topologies, which include hypercubes, meshes, and tori as special cases, are popular in part because they lend themselves to very simple routing algorithms. Early systems that used store-and-forward switching often adopted a hypercube topology because of the relatively dense interconnection network, which resulted in shorter message paths. With the advent of wormhole routing, in which internode distance has less effect on communication delay, low-dimensional meshes and tori have attracted larger followings due to their simpler physical layouts and better scalability [7].

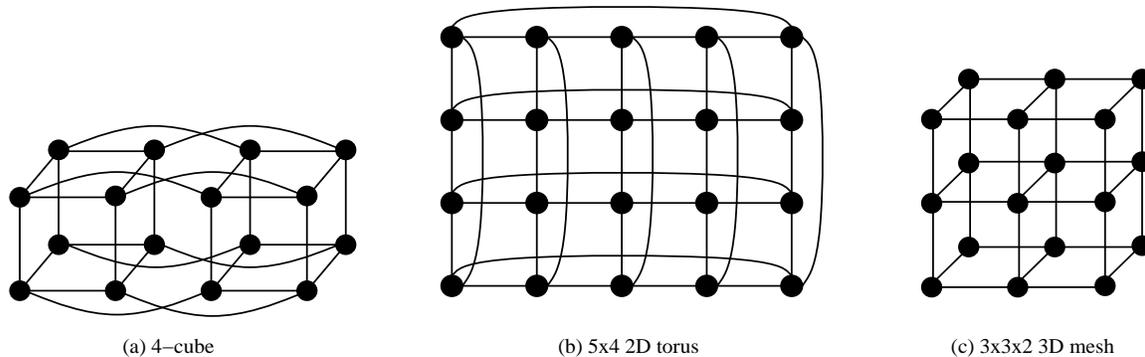


Figure 3. MPC topologies

Notable exceptions to mesh-based direct networks include the *fat tree*, which is used in the TMC CM-5, and switch-based interconnection networks, as used in the IBM SP1. The issue of collective communication is also important in these architectures and has recently drawn attention from the research community; we review some of that work later in the paper. However, we will concentrate primarily on wormhole-routed systems with hypercube, mesh, and torus topologies, for which the solutions to collective communication design possess a certain degree of similarity. Commercial and research MPCs using these network topologies include the nCUBE-2 (hypercube), the Intel Paragon (2D mesh), the MIT J-machine (3D mesh), and the Cray T3D (3D torus).

**Hardware Routing Algorithm.** The *routing algorithm* determines the path selected by a message in order to reach its destination. Since wormhole-routed messages can hold some channels while waiting for others, the routing algorithm is designed to prevent deadlock among messages. Although research in adaptive wormhole routing is very promising, most routing algorithms presently used in commercial systems are deterministic, that is, the path between a given source and destination is fixed. For example, *dimension-ordered routing*, which has been adopted in many wormhole-routed  $n$ -dimensional mesh systems, avoids deadlock by enforcing a strictly monotonic order on the dimensions of the network traversed by each message. Designing collective operations that avoid contention among constituent messages is complicated by the use of deterministic routing, since contending messages cannot be dynamically rerouted along alternate paths.

**Port Model.** Each router is connected to its local processor/memory by one or more pairs of *internal* channels, or *ports*. One channel of each pair is for input, the other for output. The *port model* of a system refers to the number of internal channels at each node. If each node possesses exactly one pair of internal channels, then the result is a so-called “one-port communication architecture.” Figure 4(a) shows a one-port node/router pair in a 2D mesh. A major consequence of a one-port architecture is that the local processor must transmit (receive) messages sequentially. Architectures with multiple ports reduce this bottleneck. In the case of an *all-port* system, shown in Figure 4(b), every external

channel has a corresponding internal channel, thus allowing the node to send and receive on all external channels simultaneously. It is also possible for a system to possess a  $k$ -port architecture, where the number of ports is greater than one but less than the number of external channels. The port model of the system can be important to collective operations, most of which involve sending multiple messages, receiving multiple messages, or both.

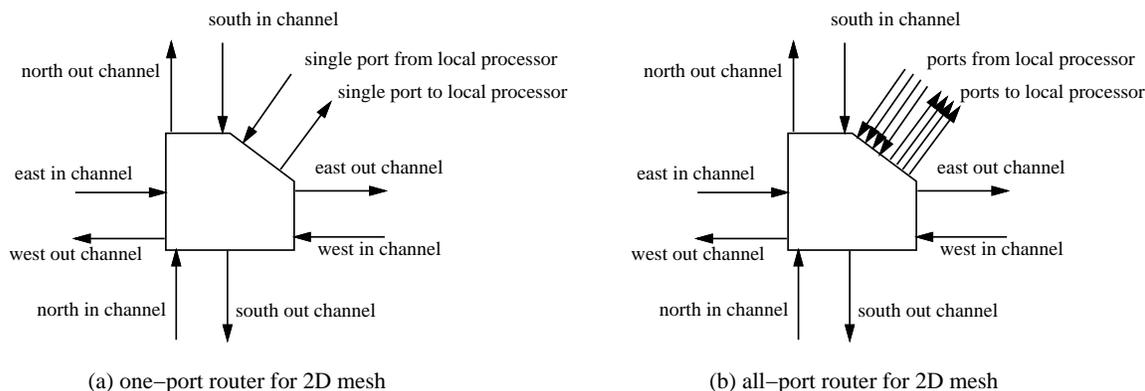


Figure 4. Two different port models for a 2D mesh

**Virtual Channels.** In meshes, the use of dimension-ordered routing is sufficient to prevent deadlock. In torus networks, however, the presence of wraparound channels can lead to routing cycles among messages, even under dimension-ordered routing. Channel-dependence cycles can be broken by multiplexing *virtual channels* on a single physical communication channel. Each virtual channel has its own flit buffer and control lines. Virtual channels may be used in a variety of ways to eliminate deadlock, but how they are used determines potential message contention, thereby affecting the design of efficient collective communication operations.

**Intermediate Reception.** The *intermediate reception* (IR) capability, also referred to as *path-based routing*, is a hardware feature that allows a router to deliver an incoming message to the local host while simultaneously forwarding it to another router, as depicted in Figure 5. Since implementing IR requires a relatively minor modification to existing routers used in MPCs, it is receiving increasing attention, particularly among researchers involved in collective communication. IR can be used to deliver a message to multiple destinations in nearly the same time that is needed to send a message to a single destination. Application of IR to multicast, broadcast, and other collective operations will be described later.

**Software-Supported Collective Communication.** Even if collective operations are implemented in software, their performance can often be improved by arranging the constituent messages so as to make better use of the underlying hardware features. Many collective operations can be

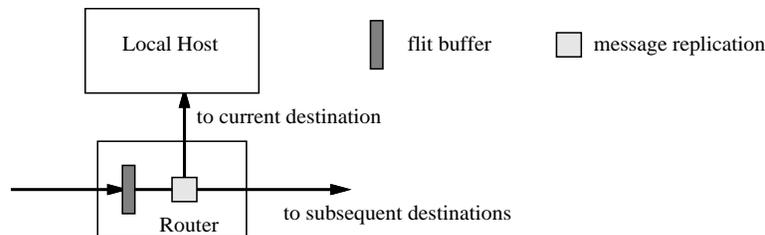


Figure 5. Operation of intermediate reception

implemented using a tree of messages. Distribution-type operations, such as broadcast, multicast, and scatter, involve sending messages from the root of the tree to its leaves. In receiving-type operations, such as gather and reduction, messages are sent towards the root from the leaves. If the communication patterns of two operations are essentially identical except that the direction of the messages is reversed, then these operations are called *duals* of one another [9]. Which type of tree to use depends on the underlying architecture and which nodes are involved in the operation. Let us define a *broadcast tree* as one that involves all nodes in the network, and a *multicast tree* as one that involves only a partial subset of the nodes. In the next three sections, we describe various tree structures and the architectures and situations for which they are best suited.

## 4 Broadcast Trees in Hypercubes

Since the introduction of the first hypercube systems in the 1980's, the study of collective communication primitives for this topology has been extensive. Many of these results concern broadcast communication, and hence several types of broadcast trees have been proposed. Since first-generation systems used store-and-forward switching, many such proposals are based on nearest-neighbor communication, although the distance-insensitivity of wormhole routing allows this constraint to be relaxed.

**Nearest-neighbor trees.** Perhaps the simplest broadcast “tree” in any network is a Hamiltonian path. From a given source node in a hypercube, a Hamiltonian path can be constructed by using the reflected gray code and the exclusive-or of the source address. Figure 6(a) illustrates such a Hamiltonian path, indicated with bold arrows, in a 4-cube. The path begins with node 0000 and ends with node 1000. In either store-and-forward or wormhole-routed networks, the time needed to complete this operation is linear in the total number of nodes, since each intermediate node must receive the entire message before forwarding it to the next node in the path. Specifically, the delay is  $N \times t_u$ , where  $N$  is the number of nodes in the network and  $t_u$  is the time required to send a unicast message between two neighboring nodes.

An approach that makes better use of the dense interconnection of the hypercube topology is the well-known spanning binomial tree (SBT) algorithm [13], which is illustrated for a 3-cube in

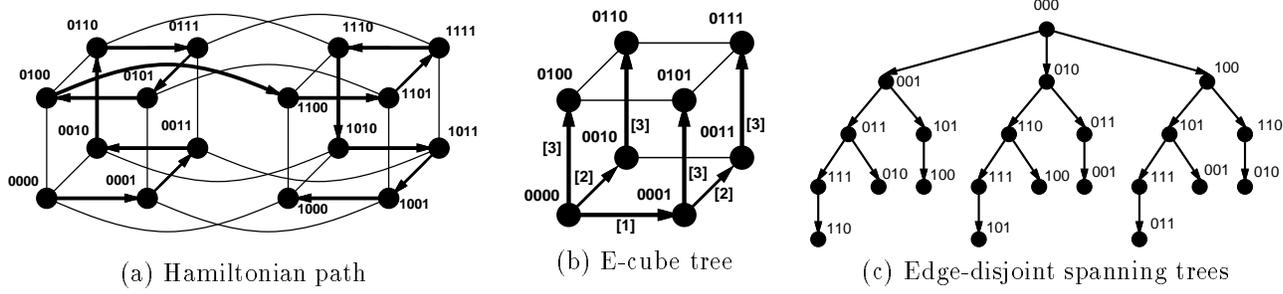


Figure 6. Hypercube broadcast trees based on nearest-neighbor communication

Figure 6(b). In the first step of the algorithm, the source node sends the message to its neighbor whose address differs from its own in the lowest (alternatively highest) bit position. Next, these two nodes send to their respective neighbors in the second dimension. This *recursive doubling* process continues until, in the last step, half of the nodes in the network forward the message to the other half through the highest dimension. This algorithm requires  $n$  message-passing steps to reach all nodes in an  $n$ -cube, with the last node receiving the message at time  $nt_u$ .

In order to reduce broadcast latency in hypercubes, Johnsson and Ho [14] proposed a tree structure that uses  $n$  edge-disjoint spanning trees (EDST) of the hypercube. In order to implement a broadcast operation using this method, the message is partitioned into  $n$  segments, each of which is transmitted along a different spanning tree, as illustrated in Figure 6(c). This algorithm employs the channels more efficiently than the SBT broadcast algorithm. The algorithm completes the broadcast of a message of length  $\ell$  in time of  $O((\ell/n) \log_2 N) = O(\ell)$ . Since the spanning trees are disjoint, this algorithm does not incur channel contention. However, the algorithm does require that the message be reconstructed at every destination. Moreover, if the architecture does not support a sufficient number of input and output ports, then this approach can incur contention at the ports. For example, in Figure 6(c), node 111 may be required to receive three message segments at approximately the same time.

**Trees designed for wormhole routing.** In the case of one-port systems, wormhole routing has little effect on the performance of broadcast trees, since a node can send (receive) only one message at a time; implementations using only nearest-neighbor communication can distribute (gather) data as fast as any other algorithm. Even the presence of multiple ports may not necessarily improve the overall performance of an operation that uses only nearest-neighbor communication. For example, Figure 7 shows the steps of the SBT algorithm in a 4-cube (either store-and-forward or wormhole-routed). If the system is a one-port architecture, then the four step solution shown in Figure 7(a) is optimal; given that a node can send only one message at a time, four steps are necessary and sufficient to reach all other nodes. If the SBT algorithm is executed on a 4-port 4-cube, as shown in Figure 7(b), some nodes receive the message earlier than in a one-port architecture. This result occurs because nodes are able to transmit multiple messages in parallel as long as they are routed through different

dimensions. However, the algorithm still requires four steps to reach all the nodes. The maximum broadcast latency is the same in both cases,  $4t_u$ , because the SBT algorithm was not designed to take advantage of either multiple ports or wormhole routing.

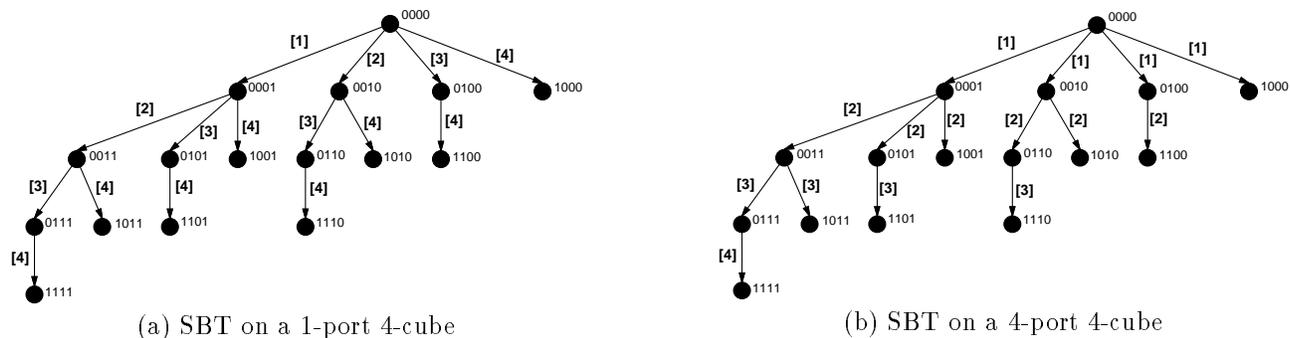


Figure 7. SBT forwarding in 4-cubes

For multiple-port systems, however, there exist better solutions that use other than nearest-neighbor communication. McKinley and Trefftz [15] proposed a simple variation on the SBT called the *Double Tree* (DT) algorithm, which is designed for all-port, wormhole-routed hypercubes. The DT algorithm begins with the source node  $s$  sending the message to node  $\bar{s}$ , whose address is the bitwise complement of  $s$ . Subsequently, nodes  $s$  and  $\bar{s}$  become the roots of *partial* spanning binomial trees. In this manner, the number of message passing steps required to reach all nodes in an  $n$ -dimensional hypercube is  $\lceil n/2 \rceil$ . Figure 8 illustrates the steps of the DT algorithm as executed on a 4-cube; for clarity, some links are not shown. Node 0000 (0) first sends the message concurrently to neighboring nodes 0001, 0010, 0100, and to node 1111, which is reached by way of routers at nodes 1000, 1100, and 1110. Nodes 0000 and 1111 then become the roots of forward and backward trees, respectively. Addresses in the first tree are resolved by changing 0's to 1's, while addresses in the backward tree are resolved by changing 1's to 0's. On a 4-port 4-cube, the broadcast is complete after only two message-passing steps, which is optimal without partitioning the message. The DT algorithm requires three steps in a 6-cube, which is also optimal. Experiments on an nCUBE-2 hypercube, which possesses a multiple-port architecture, confirm the advantage of this approach over a spanning binomial tree, particularly for large messages [15].

Although the DT algorithm reduces broadcast latency compared to the SBT algorithm, it is not optimal for larger hypercubes. Ho and Kao [16] discovered a more general solution in which the network is recursively divided into subcubes of nearly equal size; the DT algorithm is used to finish the broadcast operation in small subcubes. The approach uses the concept of a dimension-simple path. A path  $P = v_0, v_1, \dots, v_d$  in an  $n$ -dimensional hypercube is called *dimension-simple* if there exists a sequence  $i_1, i_2, \dots, i_d$  of distinct cube dimensions such that for all  $v_j, j \geq 1, v_j$  is obtained from  $v_{j-1}$  by complementing the bit at dimension  $i_j$ . The path  $P$  is called *ascending* (respectively, *descending*)

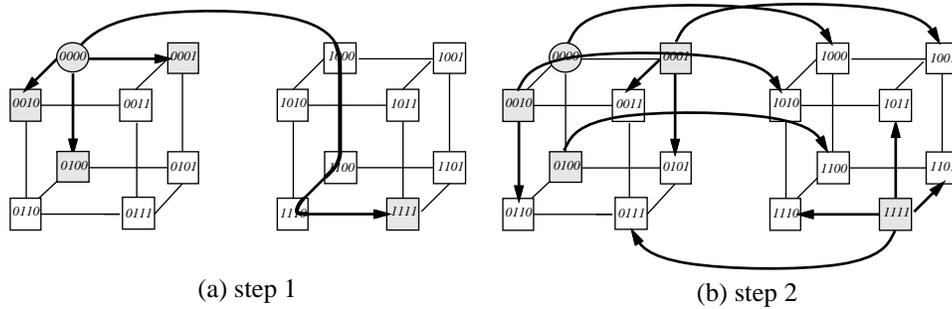


Figure 8. Example of DT broadcast in a 4-cube

if  $i_1 < i_2 < \dots < i_d$  (respectively,  $i_1 > i_2 > \dots > i_d$ ). For example, an ascending dimension-simple path from node 0000000 in a 7-cube is

$$0000000 \rightarrow 0000001 \rightarrow 0000011 \rightarrow 0000111 \rightarrow 0001111 \rightarrow 0011111 \rightarrow 0111111 \rightarrow 1111111$$

This particular path happens to traverse all the dimensions of the hypercube, that is,  $d = 7$ . In an all-port wormhole-routed hypercube in which dimension-ordered routing is performed by resolving addresses from top-to-bottom, a single node can send a message to all the nodes along an ascending path simultaneously, except for any message staggering due to sending latency. In this manner, an  $n$ -dimensional cube can be partitioned into  $n + 1$  subcubes such that each subcube contains one node on the dimension-simple path. Ho and Kao [16] have shown that any cube can be partitioned equally or nearly equally, which implies that the number of steps required by their algorithm is optimal to within a multiplicative constant, specifically, the time required is  $\Theta(\frac{n}{\log_2(n+1)})$ .

Figure 9 depicts the operation of the Ho-Kao broadcast algorithm in a 7-cube. In the first step, the source node 0000000 sends a copy of the message to the seven nodes that lie along an ascending dimension-simple path. Each of these nodes, plus the source, lies in a disjoint 4-cube; those eight 4-cubes contain all the nodes in the network. The eight intermediate destination nodes denoted in the figure complete the broadcast operation using the DT algorithm within their individual 4-cube.

**Support for Other Collective Operations.** As we mentioned earlier, tree structures are suitable for supporting other collective operations besides broadcast. For example, a tree can also be used to perform a scatter operation, as shown in Figure 10(a) for an 8-node system in which nodes have one-port architectures; the number in brackets represents the number of the message-passing step. Messages for individual nodes are sent from the source down the appropriate branch; each node stores its own messages and forwards the rest, as necessary, to its children. The dual operation, gather, is implemented by simply reversing the direction of message transmission, as shown in Figure 10(b). The same structure can be used to implement all-to-one reduction, in which case the reduction operation

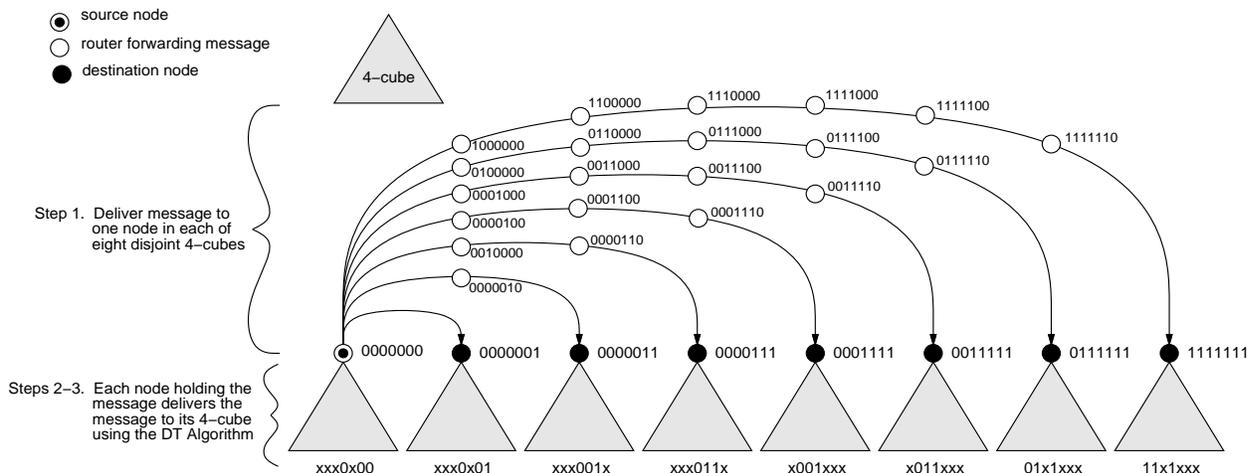


Figure 9. Ho-Kao broadcast algorithm, as executed on an all-port 7-cube

is performed pairwise on data items as they proceed towards the root. Should the result need to be distributed back to all the nodes, then the tree structure in Figure 10(a) could be used, but with only the single result message being broadcast. Other approaches to such  $N/N$  reduction operations are described in Section 7. Finally, barrier synchronization can also be implemented using both trees, as shown in Figure 10(c). The root processor collects notices from nodes that they have reached the barrier and, after hearing from all other nodes, informs the nodes that they may proceed past the barrier.

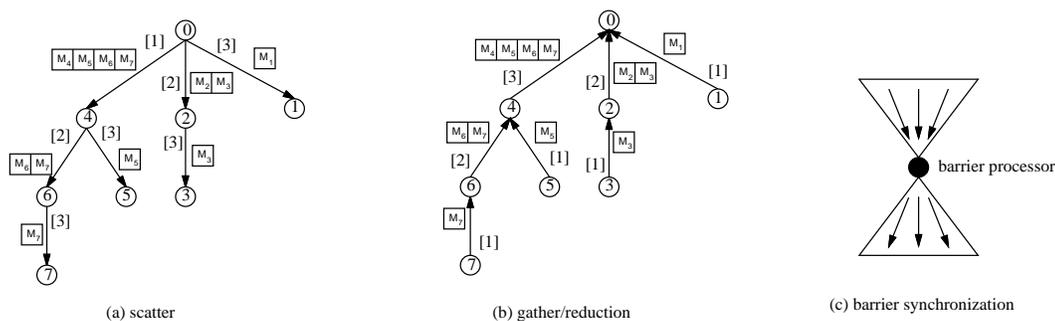


Figure 10. Use of broadcast trees in other collective operations

## 5 Broadcast and Multicast in One-Port Meshes and Tori

Many of the studies of broadcast communication in wormhole-routed meshes and tori are based on counterparts of hypercube algorithms, but exploit wormhole routing to send messages across multiple-hop paths in a single step. As with hypercubes, the solution to such problems depends on the port

model. In this section, we describe algorithms for both broadcast and multicast in one-port systems; multiple-port architectures are addressed in Section 6.

**Dimensional Broadcast Tree Approach.** The one-to-all broadcast problem in one-port wormhole-routed mesh networks was first studied by Barnett *et al* [17]. The algorithm presented in [17] operates in a recursive manner, similar to the spanning binomial tree, within each dimension of the mesh. Suppose, for example, that the source node is the leftmost node in a linear array topology, as illustrated in Figure 11. In the first message-passing step, the source node sends the message halfway across the linear array, partitioning the network into two subnetworks. In subsequent steps, each node holding a copy of the message forwards it to a node in its partition that has not yet received the message. By employing non-nearest-neighbor communication, this approach takes advantage of the pipelining effect of wormhole routing.

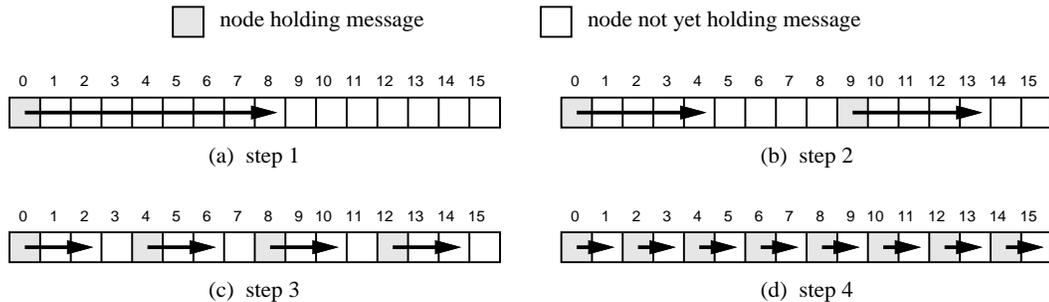


Figure 11. Recursive doubling broadcast in a 16-node wormhole-routed linear array

The algorithm may be generalized to higher-dimensional meshes by applying the above procedure to one dimension at a time. As illustrated in Figure 12, which depicts the operation of this algorithm in an  $8 \times 8$  2D mesh, the source node need not be positioned at the end or corner of the mesh. An important feature of this algorithm is that by partitioning the network in this manner, channel contention among the messages is avoided. Therefore, in the absence of contention from other network traffic, the total time required for broadcast in a  $2^r \times 2^c$  2D mesh is  $(r+s)(t_s + \ell t_f)$ . The same approach can be applied to meshes of higher-dimension as well. In general, this algorithm requires  $\sum_{i=0}^d \lceil \log w_i \rceil$  message-passing steps in a  $d$ -dimensional mesh in which the width of dimension  $i$  is  $w_i$ .

**Contention-Free Multicast.** In the examples above, the width of each dimension is assumed to be a power of 2. This assumption does not hold in many commercial systems, however. If one or more of the widths is not a power of two, then direct application of the above approach can lead to contention among the constituent messages. It is desirable to find an algorithm that achieves the lower bound on the number of steps needed to broadcast on a one-port architecture (without partitioning the message); that lower bound is  $\lceil \log_2(\prod_{i=0}^d w_i) \rceil = \lceil \log_2(N) \rceil$ .

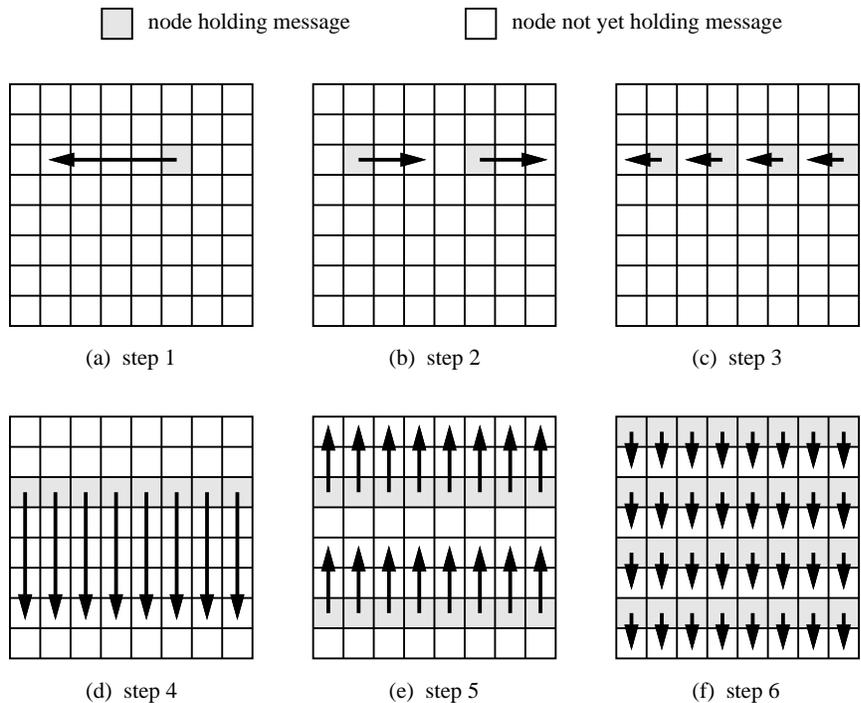


Figure 12. Dimensional broadcast in an  $8 \times 8$  mesh

It turns out that the broadcast problem is solved by a suite of algorithms that were originally designed for the general multicast problem, but which can also be applied to the degenerate case of broadcast. We now describe these multicast algorithms. As mentioned in Section 2, collective operations are often performed among a proper subset of the nodes in the network, rather than among all the nodes. Such situations arise whenever the operation involves only a subset of the processes in the application, or when the nodes allocated to a particular application do not form a contiguous, regular subnetwork. Designing a multicast tree is more difficult than designing a broadcast tree, since the latter is a special case in which any node may be called upon to relay messages. A multicast tree, on the other hand, should not involve any local processors other than the source and destinations; although the routers at other nodes may be required to route messages, interrupting their local processors wastes computing resources.

The multicast algorithms described below are all based on recursive doubling and apply to one-port,  $n$ -dimensional hypercubes, meshes, and tori that use dimension-ordered routing. For each topology, there is a corresponding algorithm, and hence the algorithms are named *U-cube*, *U-mesh*, and *U-torus*. Recursive doubling can be viewed in many ways; one possible view is depicted in Figure 13, where we assume for simplicity that  $m$ , the number of nodes involved in the multicast, is a power of 2. Figure 13 shows all unicasts occurring in the first three steps of an optimal multicast, as well as two of the unicasts that occur in the final step.

The key to avoiding contention among the constituent messages is the ordering of the destinations.

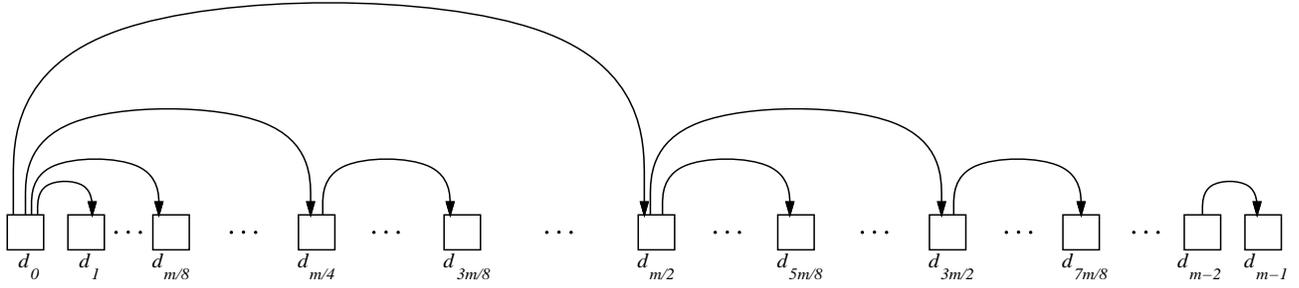


Figure 13. A minimum-time multicast

The multicast algorithms take advantage of several simple results regarding message contention and lexicographical ordering, by dimension, of arbitrary sets of nodes. In a 2D mesh, for example, such an ordering is simply a column-wise or row-wise listing of the node addresses. Such an ordering is also called *dimension order*, denoted by the symbol  $\prec_d$ . A sequence of node addresses that are ordered with respect to dimension order is referred to as a *dimension-ordered chain* [8]. Three properties of dimension-ordered chains, depicted in Figure 14, are used in these multicast algorithms. Consider any four nodes,  $u, v, x$ , and  $y$ , in a hypercube or mesh. If  $u \prec_d v \prec_d x \prec_d y$ , then dimension-ordered routes  $P(u, v)$  and  $P(x, y)$  in Figure 14(a) have been shown to be arc-disjoint [8]. That is to say, the routes do not share any channels (although they may use two, oppositely directed channels which, in some systems, may be multiplexed on the same link). Two other pairs of routes, depicted in Figures 14(b) and 14(c), are also arc-disjoint under dimension-ordered routing.

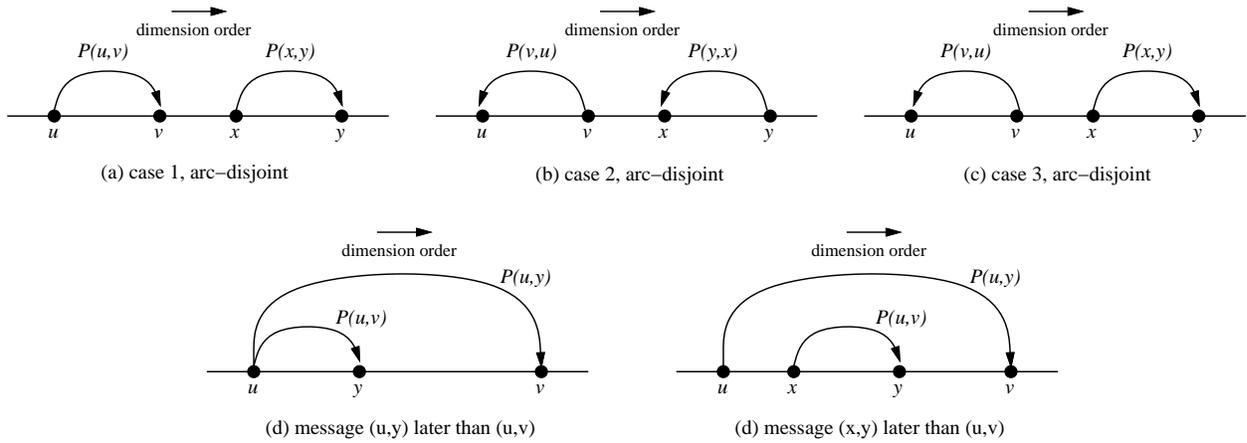


Figure 14. Arc-disjoint paths in dimension-ordered chains

If we assume that the source and destinations in Figure 13 are ordered lexicographically from left to right, then we can see that all the messages generated by the algorithm are contention-free, as follows: Since all pairs of messages sent in the same step match the pattern shown in Figure 14(a), they are guaranteed to be arc-disjoint, and therefore contention-free. Messages sent in different steps are contention-free by the nature of the algorithm when executed on a one-port architecture. Specifically,

all pairs of messages that do not match the pattern in Figure 14(a), must match one of the patterns in Figure 14(d) or 14(e). In either case, transmission of the inner message must occur after that of the outer message, so the two messages are contention-free. As described above, this *chain algorithm* is only applicable to those cases in which the source address is less than or greater than (with respect to  $<_d$ ) all the destination addresses. Clearly, this situation is not true in general, however, minor variations on this approach solve the problem in each of the three topologies.

For a hypercube network in which E-cube routing is used, the symmetry of the topology effectively allows the source node to play the role of the first node in a dimension-ordered chain. The exclusive-or operation, denoted  $\oplus$ , is used to carry out this task. A sequence  $d_1, d_2, \dots, d_{m-1}$  of hypercube addresses is called a  $d_0$ -relative dimension-ordered chain if and only if  $d_0 \oplus d_1, d_0 \oplus d_2, \dots, d_0 \oplus d_{m-1}$  is a dimension-ordered chain. The source can easily sort the  $m - 1$  destinations into a  $d_0$ -relative dimension-ordered chain,  $\Phi = d_1, d_2, \dots, d_{m-1}$ . The source may then execute the chain algorithm using  $\Phi$  instead of the original addresses. An example is shown in Figure 15. Notice that this U-cube multicast algorithm degenerates to the well known spanning binomial tree [14] for the special case of broadcast.

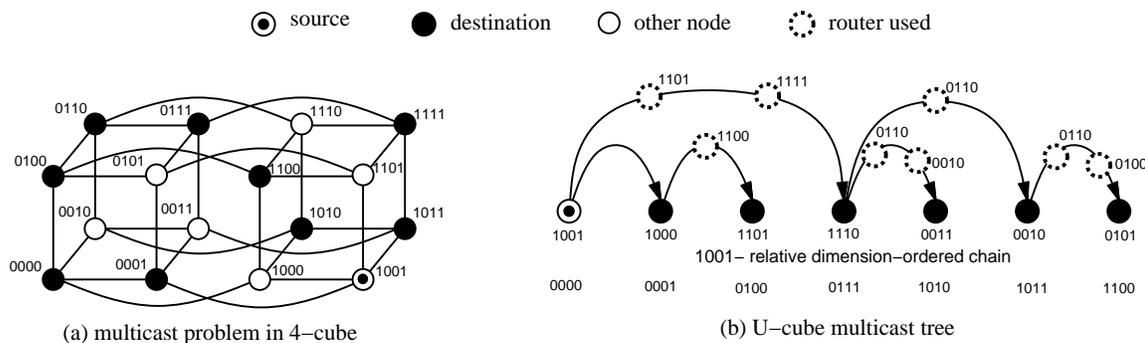


Figure 15. U-cube multicast solution in a 4-cube

In meshes, symmetry cannot be used for cases in which the source address lies in the middle of a dimension-ordered chain. However, another relatively simple method, based on Figure 14(c), may be used to address this problem. In the *U-mesh* algorithm, the source and destination addresses are sorted into a dimension-ordered chain, denoted  $\Phi$ . The source node successively divides  $\Phi$  in half. If the source is in the lower half, then it sends a copy of the message to the smallest node (with respect to  $<_d$ ) in the upper half. That node will be responsible for delivering the message to the other nodes in the upper half, using the chain algorithm. If the source is in the upper half, then it sends a copy of the message to the largest node in the lower half. The source continues this procedure until  $\Phi$  contains only its own address. Figure 16(a) depicts a multicast problem in a  $6 \times 6$  2D mesh.

The U-mesh algorithm is not contention-free when executed on a torus; however, a simple extension of the dimension-ordered chain can be used to avoid contention in a torus (and, incidentally, also

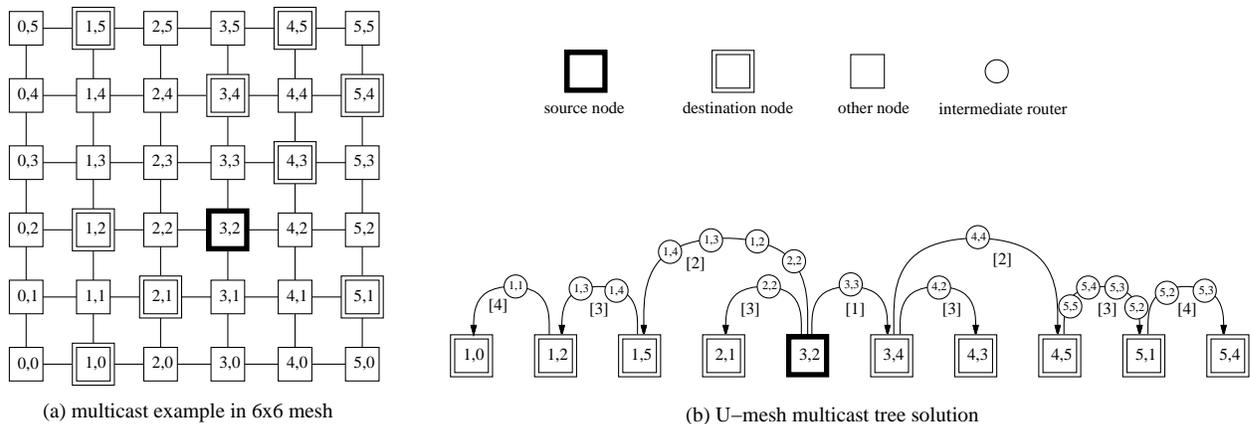


Figure 16. U-mesh multicast solution in a  $6 \times 6$  mesh

in a mesh) [18]. The source  $x_s$  and destinations are sorted into a dimension-ordered chain  $\Phi = \{x_0, x_1, x_2, \dots, x_s, \dots, x_{m-1}\}$ . Next,  $\Phi$  is “rotated” so that  $x_s$  becomes the first element in the chain. The resultant chain  $\Phi' = \{x_s, x_{s+1}, \dots, x_{m-1}, x_0, x_1, \dots, x_{s-1}\}$  is called an *R-chain with respect to  $x_s$* . The U-torus algorithm takes an R-chain as input and uses the recursive doubling algorithm described earlier. Figure 17 illustrates the application of the U-torus algorithm. In this example, the  $6 \times 6$  torus shown in Figure 17(a) is assumed to be a *unidirectional* torus; messages may be routed only towards the right or the top of the torus. The U-torus algorithm also applies to *bidirectional* tori, in which messages can be sent in either direction in each dimension [18].

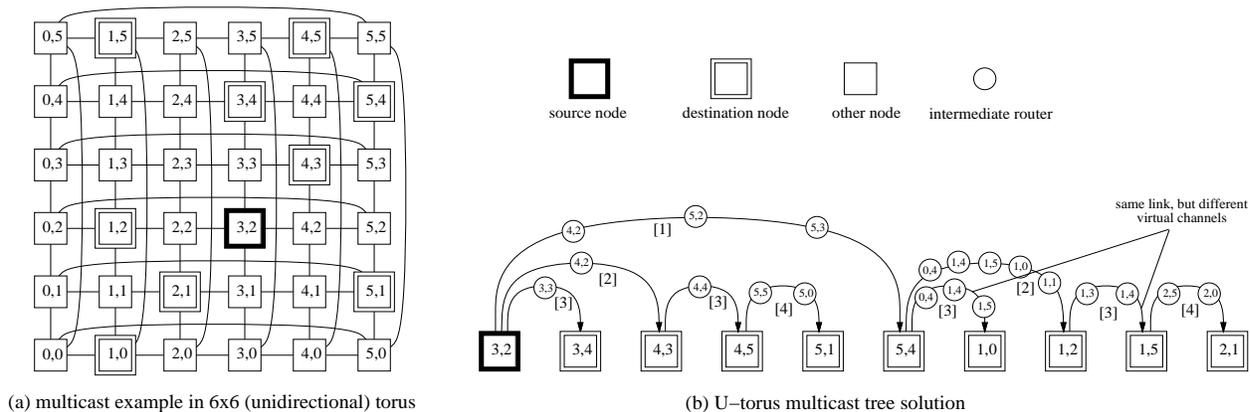


Figure 17. Example multicast using the U-torus algorithm

The reader may notice that two of the messages sent in step three appear to require the same channel from node (1, 4) to node (1, 5). However, not shown in Figure 17 are the virtual channels required to prevent deadlock in a unidirectional torus. Figure 18 shows the virtual channels connecting the nodes in the second column of the torus. The  $p$  channels are used by message that will eventually use the wraparound channel, in this case  $c_{p5}$ . The  $h$  channels are used by messages that will not

require the wraparound link and by messages that have already used the wraparound link. In the example shown in Figure 17, the message from  $(5, 4)$  to  $(1, 0)$  uses the  $p$  channel on link  $((1,4),(1,5))$ , while the message from  $(1,2)$  to  $(1,5)$  uses the  $h$  channel on that link.

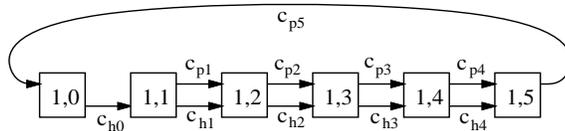


Figure 18. Virtual channels for second column in  $6 \times 6$  torus

The U-cube, U-mesh, and U-torus algorithms exhibit several desirable properties. They require  $\lceil \log_2(m+1) \rceil$  message passing steps to reach  $m$  destinations, regardless of their location, the dimension of the topology, or the shape of the topology (in the case of mesh and torus). When used for the special case of broadcast, each requires  $\lceil \log_2(N) \rceil$  message-passing steps, where  $N$  is the number of nodes in the network. Moreover, they avoid channel contention among the constituent messages. As with the broadcast trees described earlier, these algorithms can be used to implement other collective operations, such as scatter, gather, reduction, and barrier synchronization, among arbitrary subsets of nodes. Another advantage is that these algorithms do not require partitioning of the message, resulting in simpler code and fewer communication startups. For very long messages, however, message partitioning may be worthwhile, as described below.

**Message Partitioning Approaches to Broadcast.** As in the EDST broadcast algorithm described in Section 4, it is possible to take advantage of the fact that all nodes participate in the broadcast operation and are available to relay and replicate the message. Such approaches involve partitioning the message and sending parts of it along separate subnetworks, with every node eventually collecting all the pieces. These algorithms exploit redundancy in the network topology in order to reduce the time of the operation. An extension of this approach is pipelining, in which each piece of the message assigned to a particular distribution tree is further partitioned as it is sent through the tree.

One algorithm that uses this approach in 2D mesh networks is similar to the EDST algorithm for hypercubes, and is appropriately named *edge-disjoint spanning fences* (EDSF) [19]. The EDSF algorithm embeds two disjoint spanning trees, or fences, in the mesh, as illustrated in Figure 19. The source node partitions the message and alternately pipelines the pieces through the two fences. For clarity, the source in Figure 19 is the node in the upper-left corner. Beginning in step 0, the source node alternates between sending segments east and south, filling the two pipelines. The nodes can be considered to be colored black and white in a checkerboard pattern. Black nodes forward segments to the east in even numbered steps, and to the south in odd-numbered steps. Conversely, white nodes forward segments to the south in even numbered steps, and to the east in odd-numbered steps. To

accommodate any node as the source, the mesh is actually considered as a logical torus, with some segments being wormholed across the network to the west and north. (Therefore, the same algorithm could of course be used on an *actual* torus, as well.) Compared to the single tree approaches described earlier, this approach involves many more message startups that accompany message partitioning. Despite these additional costs, however, this approach has been shown to provide better performance, for relatively long messages, than a non-pipelined tree [19].

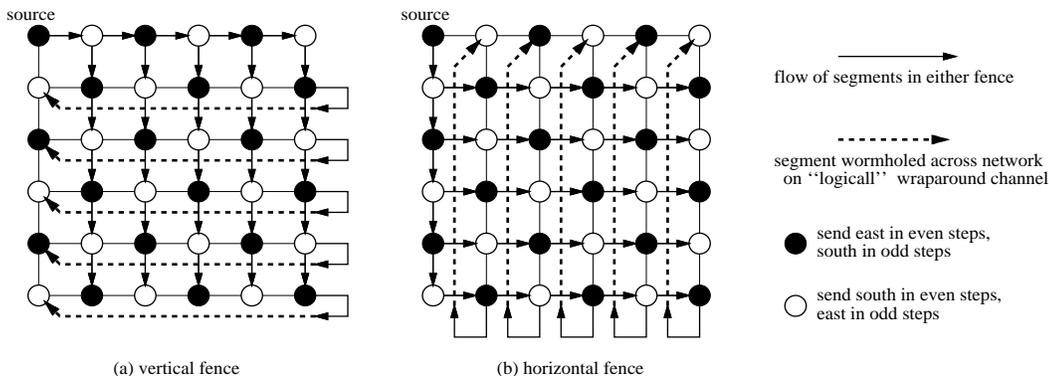


Figure 19. Two edge-disjoint spanning fences in a  $6 \times 6$  mesh

Another broadcast method that involves message partitioning, called *scatter-collect*, is also proposed in [19]. Figure 20 illustrates the operation of this algorithm in an  $8 \times 8$  2D mesh. As the name implies, the source node partitions the message and performs a scatter operation across its row, with each node receiving a particular segment of the message. (This operation could be implemented using the approach in Figure 10, which would require 3 steps to reach 8 nodes.) Next, each node in the row of the source performs a scatter in its column. Finally, every node collects the remaining segments. In the 2D mesh, the collect operation is implemented as two operations: collecting in rows and then in columns. Specifically, the nodes in each row form a logical ring; nodes circulate segments around the ring until all nodes hold all segments in that row. Next, the nodes in each column form a logical ring, and nodes circulate segments around the ring until all nodes hold all segments. This algorithm avoids channel contention. As with the EDSF algorithm, the pipelining of messages provides good performance for broadcast of long messages [19].

The scatter-collect algorithm can also be used to implement multicast in hypercubes, meshes, and tori, with the aid of the multicast tree algorithms described earlier. Specifically, the source node can use a multicast tree to scatter the data to all the nodes in the group. Next, the nodes in the group can form a logical ring, according to a dimension-ordered chain, and circulate the message segments until all destinations have received all segments. For example, suppose that the U-mesh tree depicted in Figure 16 were used to scatter segments of the message to the destinations, by recursively partitioning the message as in Figure 10(a). The destinations could then circulate the segments around a logical ring, as shown in Figure 21, until all destinations hold all the segments.

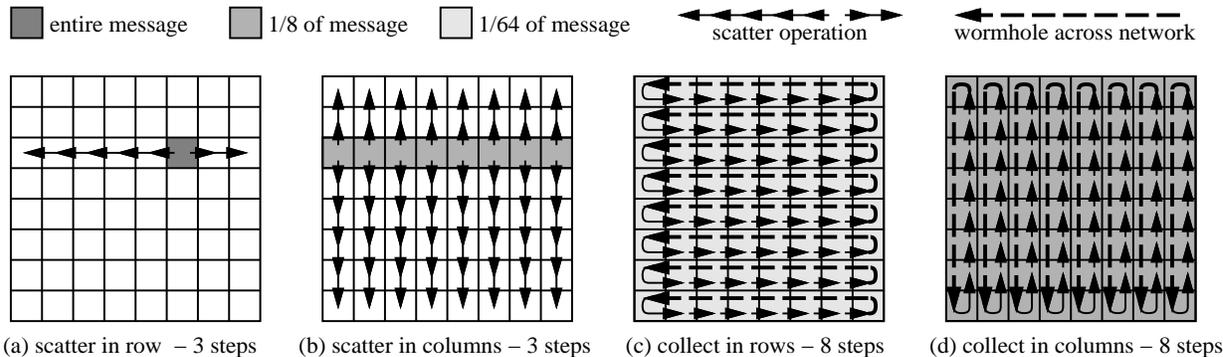


Figure 20. Scatter-collect broadcast operation in an  $8 \times 8$  2D mesh

This circulation process is contention-free, although some pairs of messages may simultaneously use two oppositely-directed channels on the same link. As with the broadcast operation, this approach may provide better performance than simple recursive doubling for long messages.

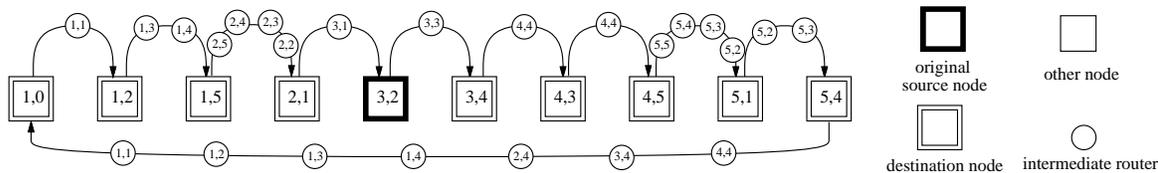


Figure 21. Collect phase of scatter-collect multicast (see Figure 16)

## 6 Broadcasting in Multi-Port Meshes and Tori

In addition to the pipelining effect of wormhole routing, we saw in Section 4 that multiple ports can be exploited in broadcast implementations for hypercubes. Similar techniques can be applied in the design of unicast-based collective communication operations in multi-port meshes and tori. Multiple ports allow a local processor to simultaneously transmit (receive) messages to (from) any of its external channels. Of course, high sending latency can cause message transmission to be staggered, even serialized if the message is small enough. However, our experiences with the nCUBE-2 [15] have shown that its all-port architecture allows considerable overlap among messages sent in succession from a given node, even though the startup latency of the nCUBE-2 is relatively high, between 80 and 100 microseconds for sending messages. Moreover, newly announced wormhole-routed systems, such as the nCUBE-3 and Cray T3D, claim much lower startup latencies of a few microseconds.

**The EDN Approach.** The *Extended Dominating Node* (EDN) model [20] has been developed recently to systematically construct unicast-based collective operations for multi-port wormhole-routed networks. The model is based on the concept of dominating sets from graph theory. A *dominating set*

$D$  of a graph  $G$  is a set of vertices in  $G$  such that every vertex in  $G$  is either in  $D$  or is adjacent to at least one vertex in  $D$ . Figure 22 illustrates the concept of node domination, using a 2D mesh. Figure 22(a) shows a set of 4 dominating nodes for the  $4 \times 4$  mesh. As illustrated with bold arrows, these four nodes can send a message to the remaining 12 nodes in a single message-passing step, by sending to their appropriate neighbors. Figure 22(b) shows an abstract representation of the dominating nodes. The EDN model broadens the concept of node domination to include nodes reachable in a single communication step under a given routing algorithm (XY routing in the case of 2D mesh networks). Figure 22(c) illustrates how four different nodes can dominate the other 12 nodes in the  $4 \times 4$  mesh. The EDN approach uses multiple *levels* of EDNs to define multiple message-passing steps of collective communication algorithms. The key issue in applying the EDN method to the development of collective operations lies in finding levels of EDNs that form regular and recursive patterns.

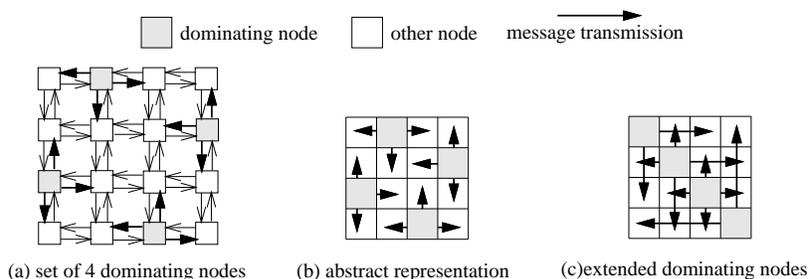


Figure 22. Dominating nodes in a  $4 \times 4$  2D mesh

Figure 23 illustrates how the EDN method can be used to implement broadcast in a  $16 \times 16$  2D mesh network under XY routing [20]. In the first two message-passing steps, called *startup* steps, the source node delivers the message to the highest-level EDNs, that is, the level-3 EDNs; there are four such nodes. The level-3 EDNs proceed to “dominate” the twelve level-2 EDNs by delivering the message to them in step 3. By the end of step 4, every level-1 EDN has received the message from either a level-2 or level-3 EDN. In step 5, which is not shown, the message is delivered to all remaining nodes by having the EDNs send to their appropriate neighboring nodes following the communication pattern shown in Figure 22(a). In general, this method can complete the broadcast operation in a mesh of size  $2^k \times 2^k$  in at most  $(k + 2)$  steps, while avoiding channel contention among the constituent unicast messages.

By taking advantage of multiple ports, this method is able to approach a lower bound on the number of message-passing steps needed for broadcast in an all-port 2D mesh networks. Specifically, the maximum number of nodes that can hold the message after  $t$  steps is  $5^t$ . Given a mesh of size  $n \times n$ , a lower bound on the time complexity is derived to be  $T \geq \lceil \log_5(n^2) \rceil$ . (It should be noted that in most cases this bound is not achievable because of the limitations imposed by XY routing.) Figure 24 compares this method to a recursive doubling algorithm (U-mesh) when executed on an all-port architecture 2D mesh. Figure 24(a) compares the algorithms in terms of the number of steps on meshes with up to  $2^{16}$  nodes. Using a recursive doubling approach, the number of steps required

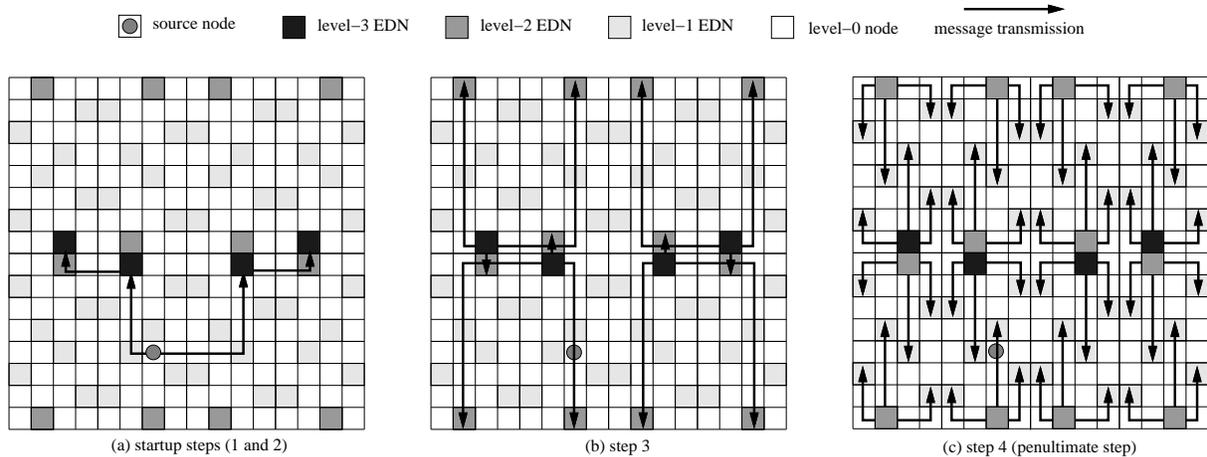


Figure 23. EDN broadcast in a  $16 \times 16$  mesh

is equal to  $\lceil \log_2(n^2) \rceil$ , regardless of the location of the source node. The number of steps required by the EDN algorithm depends on the source, but the maximum number of steps is at most one more than the minimum number. As shown in Figure 24(a), in some cases ( $n = 5, 6, 7, 12, 14, 28, 56$ ) the EDN algorithm actually achieves the lower bound. Figure 24(b) compares the maximum broadcast latency of the EDN algorithm and a recursive doubling broadcast algorithm for meshes of size  $8 \times 8$ ,  $16 \times 16$ , and  $32 \times 32$ . The message size is varied from 32 bytes to 2048 bytes. Although the recursive doubling algorithm will sometimes inadvertently take advantage of the multiple-port architecture, the advantage of the EDN algorithm is clear.

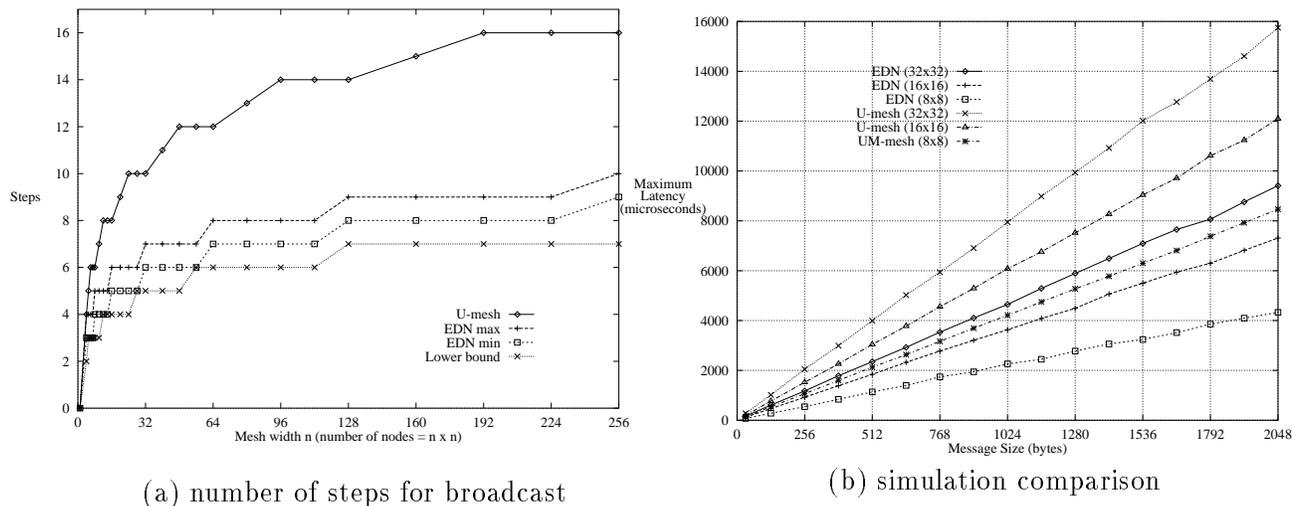


Figure 24. Performance of EDN and recursive doubling broadcast algorithms

Besides broadcast, the EDN model can be applied to other collective operations besides broadcast. For example, the EDN approach has also been applied to reduction and gather, which are duals of broadcast, to the problem of matrix transposition, an example of a permutation operation [20], and

to other topologies, including higher-dimension meshes and tori. Other research has addressed the problem of multicast in all-port hypercubes [21]. However, numerous problems in the area of collective communication in multi-port systems remain to be explored.

## 7 All-to-All Communication

The complex collective operations, such as complete exchange and all-to-all broadcast, can be implemented by having multiple nodes simply invoke other operations, such as broadcast, scatter, and gather. However, more efficient solutions that involve less channel contention are possible when the complex operation is treated as an atomic unit in which all group members participate in the entire operation, rather than just a suboperation. Several new algorithms, designed for wormhole-routed architectures, have been proposed for these operations.

**Complete Exchange.** The complete exchange operation requires that each node transmit a unique message to each of the other nodes; in other words, every pair of nodes must exchange messages. Complete exchange is sometimes referred to as *all-to-all personalized communication* or *all-to-all scatter-gather*. The operation involves  $N^2$  messages,  $M_{i,j}$ , for  $0 \leq i, j \leq N - 1$ . Usually, it is assumed that all messages are the same length,  $\ell$ . Initially, each message  $M_{i,j}$  is stored at node  $i$ ; after the operation,  $M_{i,j}$  is to be stored at node  $j$ .

In order to realize a complete exchange in store-and-forward systems, the algorithm must utilize indirect communication, in which one or more intermediate nodes receive and then relay a message traveling between two non-adjacent nodes. To save communication startup time, the algorithm may combine messages into larger message blocks to be transmitted as a single unit between adjacent nodes. The *standard exchange* algorithm for hypercubes [14] is an example of a complete exchange algorithm designed for one-port, store-and-forward systems. As shown in Figure 25, the standard exchange algorithm consists of  $n$  steps; during each step, the cube is recursively divided into halves, and messages are exchanged across this new division, between pairs of corresponding nodes. Not shown in the figure is the local rearrangement of data between each step, which allow each message to be transmitted from a contiguous block of memory. In the first step, node  $i$  sends to node  $(i \oplus 2^0)$  all messages residing at node  $i$  that are destined for the half-cube containing node  $(i \oplus 2^0)$ . During the next step node  $i$  sends to node  $(i \oplus 2^1)$  all messages residing at node  $i$  that are destined for the quarter-cube containing node  $(i \oplus 2^1)$ . This process continues recursively. By combining individual messages into larger message blocks, the standard exchange algorithm is able to accomplish a complete exchange in only  $n$  communication steps while using only nearest-neighbor communication. Each transmitted message is of size  $(N/2)\ell$ , where  $\ell$  is the size of each message and local data permutation is required after each step.

The same message-passing pattern can be used for other operations involving all nodes in the

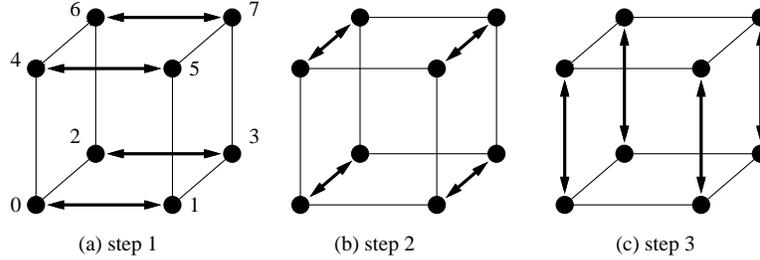


Figure 25. Standard exchange in a 3-cube

hypercube, although the size of the messages in each step differs [9]. In each step of all-to-all broadcast, every node sends its own message as well as all those it has received from other nodes; the message size in step  $i$  is  $m2^{i-1}$ . For  $N/N$  reduction, every node performs a reduction after each step and forwards the result in the next step; therefore the size of the message sent in every step is  $\ell$ . After the  $n$ th step, all nodes will hold the reduced value. Since the standard exchange algorithm uses only nearest-neighbor communication, its performance will be approximately the same on either a store-and-forward or wormhole-routed architecture.

The *direct algorithm* for hypercubes [22], is a complete exchange algorithm designed to take advantage of the distance-insensitive communication of one-port, wormhole-routed hypercubes. The algorithm is very simple: there are  $N - 1$  communication steps; in each step  $k = 1, 2, \dots, N - 1$ , node  $i$  sends message  $M_{i,j}$  to node  $j = i \oplus k$  and receives message  $M_{j,i}$  from node  $j$ . Thus, all communication occurs directly between the original source node of a message and the final destination node of that message. For communication between non-adjacent nodes only the routers are required to relay the message at the intermediate nodes on the path. The direct algorithm is stepwise contention-free, in that no two messages transmitted during any particular step will require a common communication channel.

In contrast to the standard exchange algorithm, each node in the direct algorithm transmits only a single message during each step. The direct algorithm is thus more efficient in terms of pure message transmission time, requiring each node to send (and receive)  $\ell$  bytes during each of the  $N - 1$  steps, for a total of  $(N - 1)\ell$  bytes per node; the total time for the operation is  $(N - 1)(t_s + \ell t_f)$ . The standard exchange algorithm, on the other hand, requires that each node send (and receive)  $(N/2)\ell$  bytes during each of  $n$  steps for a total of  $n(N/2)\ell$  bytes per node and a total execution time of  $n(t_s + (N/2)\ell t_f)$ . Also, where the standard exchange algorithm must completely “shuffle” the local data after each communication step, no local data permutation is required by the direct algorithm. Which algorithm is faster will depend on the ratio of the startup latency to the network latency, as well as the message size  $\ell$ . Shorter messages and larger startup-to-network latency ratios will favor standard exchange while longer messages and smaller ratios will result in better performance for direct exchange. Also, the time required to permute the data after each step of the standard exchange algorithm should not be ignored.

In a variation of the direct algorithm for hypercubes, the *direct algorithm for 2D meshes* [23] performs the complete exchange in a wormhole-routed 2D mesh through a combination of pairwise message exchanges. Like its hypercube counterpart, the direct algorithm for 2D meshes is a simple algorithm: the nodes of the mesh are assigned the ordinal numbers 0 through  $N - 1$  in a row-major fashion. During step  $k$ , for  $k = 1, 2, \dots, N - 1$ , the node with ordinal number  $i$ , for  $0 \leq i \leq N - 1$ , sends message  $M_{i,j}$  to, and receives message  $M_{j,i}$  from, the node whose ordinal number is  $j = i \oplus k$ . Figure 7 shows the communication steps of the direct algorithm on a 2D ( $2 \times 4$ ) mesh using dimension-ordered routing. Notice that in steps 2, 3, 6, and 7, messages simultaneously require common communication channels, leading to stepwise channel contention. Nonetheless, on systems with relatively high startup latency and, therefore, excessive communication capacity, this factor may not significantly degrade performance [23]. The algorithm can be generalized to accommodate arbitrary mesh sizes [23].

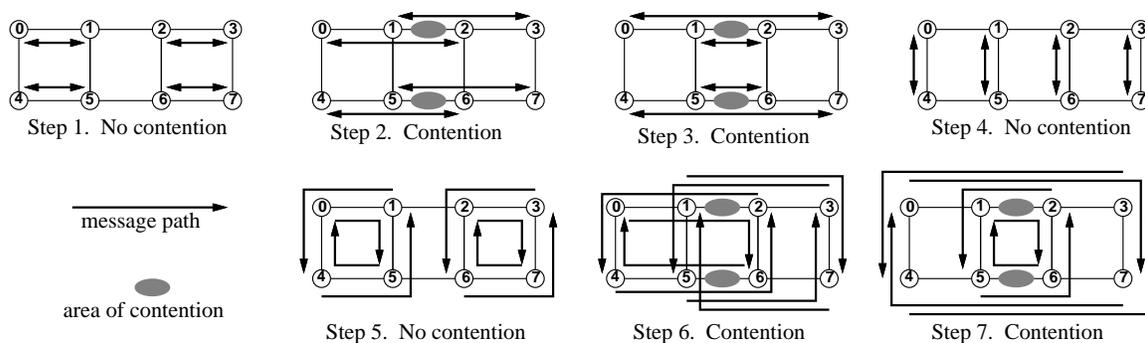


Figure 26. Direct pairwise exchange on a 2D mesh [23]

Another algorithm for complete exchange on a wormhole-routed 2D mesh, the *binary exchange algorithm for 2D mesh* [24, 25], is derived from the standard exchange algorithm for hypercubes. Whereas the standard exchange recursively halves the hypercube during each *phase*, the binary exchange algorithm recursively halves the 2D mesh during each phase, alternately in the  $X$  and  $Y$  dimensions. Thus,  $2d = \log_2 N$  communication phases are required for a  $2^d \times 2^d$  mesh; these phases are illustrated in Figure 27 for an  $8 \times 8$  mesh. As with standard exchange, blocks of  $N/2$  messages are exchanged between pairs of nodes, and local message permutation is required after each communication phase. During the first phase, each node exchanges messages with the corresponding node in the other half of the mesh, based on a vertical division of the mesh; in the second phase, communication occurs between corresponding nodes in opposing quadrants, based on the new horizontal division of the mesh. As Figure 27 illustrates, all communication occurs between pairs of nodes in the same row or in the same column. Since this algorithm incurs channel contention in each phase, we have labeled each phase with the actual number of message-passing steps required.

In contrast, the *quadrant exchange algorithm* [24] was designed specifically to account for wormhole routing and mesh topologies. During each stage of the algorithm, quadruples of nodes exchange messages among themselves in a series of three communication steps, as shown in Figure 28(a). Af-

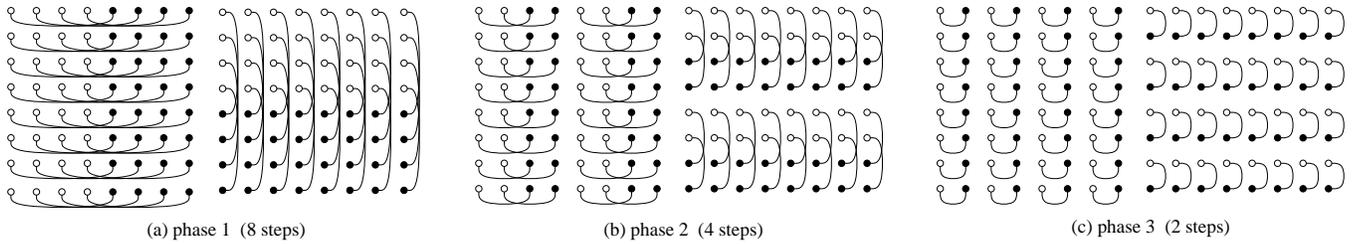


Figure 27. Binary exchange algorithm on a 2D mesh [25]

ter completing these three steps, the four nodes have then completely exchanged messages among themselves. Due to the distance-insensitive nature of wormhole routing, the four nodes shown in Figure 28(a) need not be adjacent; they need only form the corners of a rectangle in the mesh.

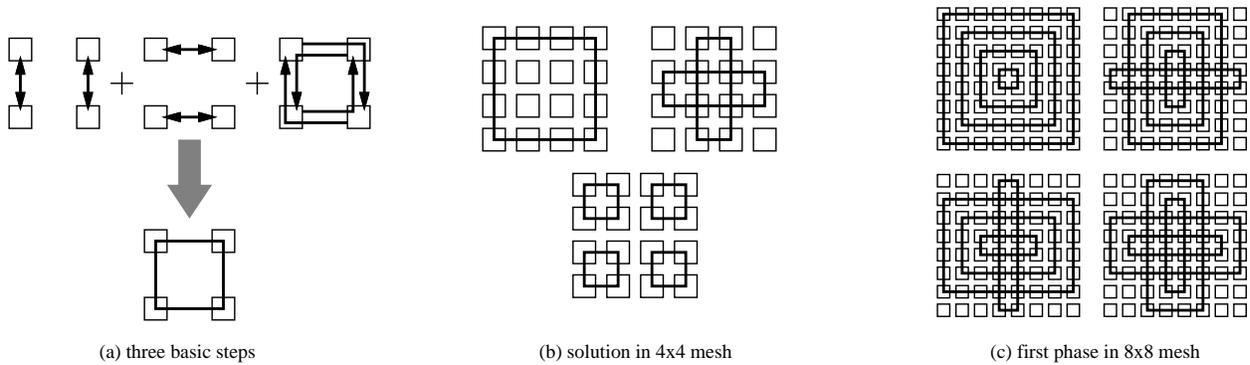


Figure 28. The quadrant exchange algorithm on a 2D mesh

The quadrant exchange performs the complete exchange by “quartering” the mesh; this process involves performing all the necessary inter-quadrant exchanges in a series of stages. Figure 28(b) shows complete exchange operation on a  $4 \times 4$  mesh. In the top two steps, the data at every node is copied to one node in each of the other three  $2 \times 2$  quadrants. In the bottom step, the operation is performed on each of the quadrants. Figure 28(c) shows the first phase (four steps) on an  $8 \times 8$  mesh; these operations quarter the mesh. To complete the total exchange, the algorithm is then concurrently applied to each of the four  $4 \times 4$  quadrants. The selection of the quadruples within each stage is made so that communication is stepwise contention-free. In general,  $2^{j-1}$  stages are required to quarter a  $2^j \times 2^j$  mesh, thus,  $\sum_{i=1}^j 2^{j-i} = 2^j - 1 = \sqrt{N} - 1$  stages are needed. Since each stage consists of three communication steps, then a total of  $3(\sqrt{N} - 1)$  communication steps are needed for the quadrant exchange.

The quadrant exchange algorithm falls somewhere between a strictly store-and-forward algorithm and a direct exchange. Although messages are transmitted in blocks rather than individually, direct communication often occurs between non-adjacent nodes. During each stage of the algorithm, a node  $x$  involved in a quadruple must send to the other three nodes of the quadruple all messages currently at node  $x$  that are destined for the three quadrants of the mesh not containing node  $x$ . A node in

one such quadrant is accessed by  $x$  during each of the three communication steps of the stage. Thus, blocks of  $N/4$  messages are exchanged between pairs of nodes during each communication step.

Table 2 compares the number of steps and individual messages per transmitted message block for each of the three 2D mesh total exchange algorithms described above. The quadrant exchange algorithm, designed to balance the extremes of the binary exchange and direct algorithms, clearly falls between these two algorithms in both the number of communication steps and the size of the message blocks that are exchanged during each step. The product of these two values, which represents the total number of messages sent (and received) by a single node during the total exchange, is also balanced by quadrant exchange between the extremes of the other two algorithms. Only quadrant exchange generates messages that are stepwise contention-free. Thus, the number of communication steps listed for the other two algorithms is, in a sense, optimistic, since contention for communication channels will cause some of these communication steps to be delayed.

Algorithm	Direct	Binary exchange	Quadrant exchange
Comm. steps	$N - 1$	$\log_2 N$	$3(\sqrt{N} - 1)$
Message size	$\ell$	$(N/2)\ell$	$(N/4)\ell$
Msgs per node	$N - 1$	$(N/2)(\log_2 N)$	$(3N/4)(\sqrt{N} - 1)$

Table 2. Communication steps and message block size for 2D mesh total exchange algorithms

**All-to-All Broadcast.** All-to-all broadcast can be regarded as a degenerate case of complete exchange in which each node sends the same message to every other node. Although this operation may be implemented as multiple, separate one-to-all broadcasts, network contention could significantly delay the operation. Most algorithms for store-and-forward networks use a pipelining approach, in which nodes forward each other's messages across the dimensions of the system. Unlike other collective operations, the availability of wormhole routing has thus far yielded relatively little improvement over store-and-forward algorithms. In one-port hypercubes the standard exchange method described earlier achieves the same performance on both store-and-forward and wormhole-routed systems, since it uses only nearest-neighbor communication. Since the message size doubles with each of the  $n$  steps, the time required is  $nt_s + (N - 1)\ell t_f$ . In one-port meshes and tori, a generalization of this method can be used, whereby the nodes forward the messages around a logical ring in each dimension. The one benefit of wormhole routing is that the message can be wormhole-routed across the mesh; there is no difference for the torus. This approach is identical to the collect phase of the scatter-collect broadcast algorithm described earlier. For example, the suboperations depicted in Figures 20(c) and 20(d) are actually performing an all-to-all broadcast on the data that had been scattered about the network. This operation requires  $w_i$  steps in each dimension  $i$  of a  $n$ -dimensional mesh or torus. The message size increases by a factor of  $w_i$  with each dimension. The total time is  $\sum_{i=0}^{n-1} (w_i t_s + (n - i)\ell t_f)$ .

For all-port hypercubes, Bertsekas *et al* [26], developed algorithms for all-to-all broadcast, scatter, and complete exchange that were shown to be optimal under both store-and-forward as well as wormhole-routed switching. These algorithms require relatively close synchronization among the nodes. The fundamental concept is to use multiple disjoint spanning trees and to schedule message transmission such that broadcasts from different nodes avoid one another in the network. Since the port utilization at some nodes is 100% over the entire operation, the network switching technique does not affect performance.

**Reduction and Scan.** We mentioned previously that the standard exchange algorithm (Figure 25) can be used to implement  $N/N$  reduction in hypercubes. As described by Kumar [9], the same approach can also be used to implement scan operations by adding an additional “result” buffer at each node [9]. The result buffer at each node is initialized with the data at that node. The message passing procedure is the same as for  $N/N$  reduction, except for one additional operation: at the end of each communication step, the result buffer is updated only if the message in that step arrived from a processor with a lower label than the recipient processor. At the end of  $n$  steps, each node’s result buffer will hold the appropriate value.

In a similar manner, the same message passing pattern used in all-to-all broadcast in meshes and tori can also be used to implement reduction and scan operations in those topologies. In the case of scan, some care must be taken to ensure that the final result buffer at every node is based only on data from lower-labeled nodes. Finally, if only a subset of nodes in the network is involved in the operation, then a dimension-ordered chain can be used to circulate reduction/scan data, in the manner shown in Figure 21 for the collect phase of scatter-collect multicast. These approaches are straightforward extensions of the algorithms described earlier; hence, their details are omitted.

## 8 Hardware Support for Collective Communication

Until now, we have considered how to implement collective communication in systems that provide only unicast communication in hardware. Such algorithms can improve their performance by taking into account several characteristics of the underlying architecture, such as the wormhole switching and the port model, even though those features were intended primarily to support unicast communication. An alternative approach is to provide more direct support for collective communication in the system. Two main approaches have been studied. In the first, collective communication is implemented in a network other than the primary data network. In the second, the data network is enhanced to better support certain collective operations. Each of these approaches is discussed in turn.

**Support Outside the Data Network.** Two MPCs that supplement the data network with additional hardware for various collective operations are the TMC Connection Machine CM-5 and the

Cray T3D. In the following, we briefly describe these systems and how they support collective communication. However, the readers are referred to the papers cited in the following text for many of the details that we omit due to space limitations.

The CM-5 [27] is a descendent of the CM-2, a fine-grained SIMD system. The CM-5 is designed to support *synchronized MIMD* computing, whereby nodes are allowed to execute independently of one another, with special hardware provided to support relatively fine-grained synchronization and communication among nodes. The CM-5 actually has three independent networks: a data network, a diagnostic network, and a control network. The wormhole-routed data network for unicast communication is based on the fat-tree topology, which offers several paths between every pair of nodes and a routing function that “pseudo-randomly” selects among the possible paths. Figure 8 shows an abstract representation of the data network for a 64-node CM-5. The data network interface is memory-mapped in order to reduce startup latency. The diagnostic network uses a binary tree topology and permits “back door” access to all system hardware. The control network, which also uses a binary tree with processors located at the leaves, provides direct hardware support for three classes of collective operations: broadcast, reduction, and barrier synchronization. The interfaces to the control network are memory mapped, and the operations must be executed on small, fixed-length messages; operations on larger messages may be implemented by cascading multiple operations.

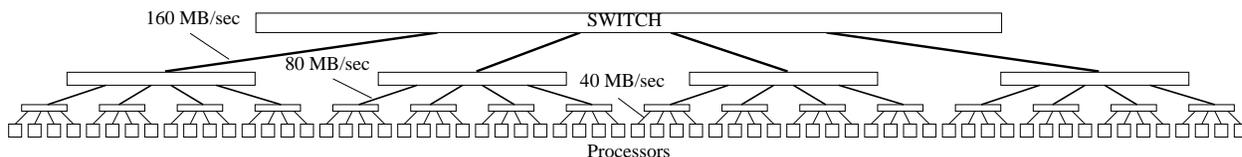


Figure 29. A 64-node CM-5 data network fat-tree [28]

Since the CM-5 data network supports adaptive routing, the issues associated with the design of unicast-based collective operations for this system are much different than those of deterministically-routed  $k$ -ary  $n$ -cube systems, on which this paper has focused. Ponnusamy *et al* [29] have experimented with various collective operations and parallel algorithms on the CM-5, concentrating primarily on the effect of contention in the network. Bozkus *et al* [30] have compared the performance of collective operations as implemented on the CM-5 with the Intel Touchstone Delta. The CM-5 implementations take advantage of the control network, while the Delta implementations rely strictly on software. The CM-5 wins easily for operations on small messages. However, for large messages, the software overhead of the Delta is sufficiently amortized over the network latencies of the messages that the operations execute faster on the Delta than on the CM-5. Finally, Brewer and Kuszmaul [28] suggest that the data and control networks of the CM-5 may be used in tandem to implement fast collective operations. In particular, barriers are used within collective data operations in order to reduce contention within and between different phases of the operation, greatly improving performance.

The Cray T3D is an MPC that is designed to run as a back-end processor to a Cray Y-MP or Cray

C90 host system. The data network of the T3D is a wormhole-routed, bidirectional 3D torus. Unlike the other systems considered in this paper, the architecture of the T3D supports distributed shared memory. Memory references are translated into small packets by the network interface hardware; memory consistency is the responsibility of software. Besides the data network, the T3D provides additional hardware support for two forms of collective communication; both involve synchronization among the nodes. First, each processor has access to two 8-bit registers. Each bit is connected to a separate tree-wired barrier synchronization circuit. A processor invokes a barrier by setting the corresponding bit; when all processors involved have set their bits, the circuit will reset all the bits to 0. A processor may either busy-wait or be informed by an interrupt when the barrier condition is met. Second, each node contains two fetch-and-increment registers, which are globally accessible to all other processors. Since many programs distribute work segments to processes according to data array indices, these registers provide an efficient mechanism for a process to seize a particular piece of the computation without using a server process. These choices of functionality appear to provide a good balance between limiting extra hardware and providing fast support for synchronization operations most frequently used under the shared-memory programming paradigm.

**Support Within the Data Network.** An alternative to implementing a separate network is to enhance the primary data network with somewhat more generic functionality that can be used to support several collective operations. To improve collective communication performance and reduce software overhead, two such enhancements to network routers have been proposed and implemented: *message replication* and *intermediate reception*. Message replication refers to the ability to duplicate incoming messages onto more than one outgoing channel, while intermediate reception is the ability to simultaneously deliver an incoming message to the local processor/memory and to an outgoing channel.

Unicast-based collective operations rely on local processors to replicate and relay messages. A seemingly natural extension of a multicast or broadcast tree is to have the router itself replicate the message, flit-by-flit, and forward it onto multiple outgoing links, effectively producing a tree-like message worm. Figure 30(a) shows how message replication can be used to solve a multicast problem in a  $6 \times 6$  2D mesh. The routers at nodes (1,2), (3,2), (4,2), and (5,2) are required to replicate the message; the heads of the message follow XY paths. A difficulty with this approach is that when any branch of the tree encounters a required channel that is unavailable, the entire message tree must be blocked, rendering all channels used by the tree unavailable for other communication. Moreover, the dependencies among branches of the tree can lead to deadlock. The nCUBE-2 hypercube includes hardware support for message replication, which is used to implement tree-based broadcast and limited multicast where the destinations form a subcube, but this mechanism is not deadlock-free [31].

In order to avoid the disadvantages of the tree-like worms produced by message replication, and yet apply hardware support to the implementation of collective communication operations, the technique

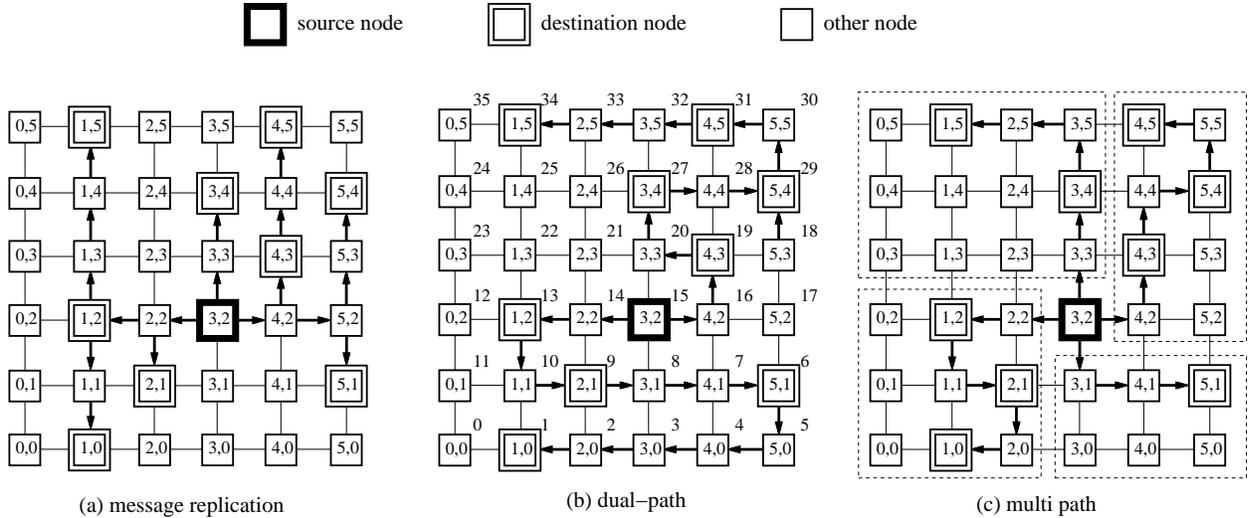


Figure 30. Message replication and path-based routing

of intermediate reception (IR) has been proposed. A router possessing IR capability is able to copy the flits of a message to the memory of the local processor as the message passes through the router enroute to other destinations. In this way, a message originating at a source node can be routed as a single worm through several destination nodes, depositing a copy of the message at each of the intermediate destinations as it passes through. Such communication methods are termed *path-based*, while the constituent messages are called *multi-destination worms*. Intermediate reception has been used in the University of Michigan HARTS machine, which uses a wraparound hexagonal mesh topology, and in a version of the Caltech Mesh-Routing Chips.

The routing rules used for path-based routing must be liberal enough to allow flexibility in message routing, so that many intermediate destination nodes can be visited by a single message. However, these rules must still avoid deadlock. One way in which deadlock can be avoided in path-based routing is by ensuring that there are no cycles of channel dependency allowed by the routing algorithm. Ensuring an absence of channel dependence cycles, and thus freedom from deadlock, can be accomplished by means of a *Hamiltonian Path* (HP). For example, in Figure 30(b), the numbers near the upper-right corner of each node correspond to one (of many) HPs through the network. If the routing algorithm can be designed so that messages always visit nodes (including the destination nodes and the intermediate nodes at which only the router is used) in the same order as those nodes appear on a particular HP, then cyclic dependences will be impossible, and deadlock will be prevented.

Lin, *et al* [31] have used an HP similar to that represented in Figure 30(b) to develop a family of path-based multicast routing algorithms for 2D mesh networks. In the most basic of these algorithms, termed *dual-path*, the source node of a multicast generates at most two multi-destination worms. One worm travels forward from the source, visiting all destination nodes that appear *after* the source on the HP. Another multi-destination travels backwards on the HP, reaching those destination nodes

that appear prior to the source node on the HP. Figure 30(b) shows the two message worms generated by the dual-path algorithm for an example multicast operation. Notice that the algorithm allows messages to “skip over” some of the nodes on the HP, provided that the nodes visited by any message worm appear in the same relative order as defined by the HP.

To take advantage of the communication parallelism available in systems with all-port architecture, this algorithm can be extended to generate up to four concurrent multi-destination message worms in a 2D mesh, as shown in Figure 30(c). Each worm reaches the destination nodes in the corresponding quadrant of the mesh, with respect to the source node. Again, each worm visits nodes in an order that is consistent with the HP represented in Figure 30(b), thereby ensuring deadlock-free communication. Besides 2D meshes, the path-based approach can be applied to hypercubes and tori, as well.

In path-based routing for operations with arbitrary destination sets, such as multicast, the message worm is generally prefixed with a list of destination node addresses, or offsets between destinations. The order of these addresses corresponds to the order in which the destination nodes will be visited. Once the message header reaches the first destination node in the list, the router at that node strips its own address from the head of the list, and forwards the remainder of the message worm toward the next destination node, while simultaneously copying the message contents to the local host memory.

The port model of the system also has a large effect on the design and performance of path-based collective communication operations. Path-based routing typically requires multiple input ports, also called *consumption channels*, else deadlock may occur when two or more path-based collective operations are issued concurrently. Figure 31 illustrates a scenario in which two multi-destination message worms are each attempting to deliver a message to nodes  $x$  and  $y$ . However, each message is holding the single input port at one node, while waiting to use the input port at the other node, resulting in communication deadlock.

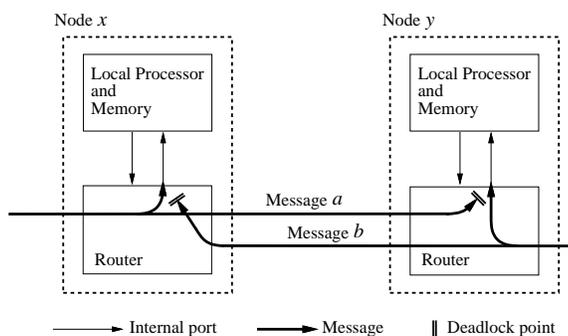


Figure 31. Port-induced communication deadlock in a one-port system

Finally, multiple path-based operations may be combined to form another collective operation. Even broadcast may be implemented more quickly using multiple, multidestination worms than using a single worm, due to the excessive length of the latter. In the worst case, a single worm of length  $m$  requires time  $t_s + (N - 1 + m)t_f$ , where  $N$  is the number of nodes in the network and  $t_f$  is the time

to transmit a flit between two routers. In a 2D mesh, however, if the source multicasts to all nodes in its row, and each node in the row then multicasts to all nodes in their respective columns, then the time required is  $2(t_s + (\sqrt{N} - 1 + m)t_f)$ . Panda and Singal [32] have proposed just such an approach to broadcast in a mesh, which can achieve much better performance than a single-path approach in large meshes. Ho and Kao [33] have also proposed a path-based version of the broadcast algorithm described in Section 4. The routing is identical to that of the algorithm in Section 4, except that each node sends to all its children along a single path in one step. It would appear that path-based collective communication has merits, although this area has received relatively little attention thus far [32].

## 9 Conclusions

As the supercomputing industry has migrated towards parallel architectures, many with distributed memory, wormhole-routed switching has played a major role in reducing the cost of unicast communication among nodes. As this paper has shown, many new algorithms have also been proposed to implement the collective communication operations in wormhole-routed networks. In many cases, these solutions differ fundamentally from their store-and-forward counterparts because they attempt to exploit the distance insensitivity of wormhole routing. Other architectural characteristics, such as the topology, port model, and hardware routing algorithm, must also be accounted for in order to achieve good performance.

This paper has attempted to elucidate several key issues in the design of collective operations in wormhole-routed networks and has highlighted some of the extensive research that has already been conducted in this area. We have focused primarily on software solutions that take advantage of underlying architectural characteristics. In order to keep the survey manageable, however, several related topics were intentionally omitted, such as adaptive routing, fault-tolerance, and switch-based interconnection networks. These are important topics in their own right, and surveys of the research and commercial projects in these areas would likely be of great interest to the community.

Several of the topics mentioned in the paper have only recently begun to receive attention from the research community and merit further study. First, as adaptive routing algorithms mature and begin to appear in research prototypes and commercial systems, the need arises to study how this capability can be exploited by collective operations. For example, it may be possible to design collective operations so that the constituent messages adapt to one another in a desired manner. Second, the study of collective operations in multi-port systems and in systems that support intermediate reception has only just started. Many subtopics have not yet been addressed at all, even through analysis and simulation. In addition, research prototypes that possess these features are needed for experimental studies. A particularly intriguing subject is multiphase collective operations based on underlying path-based messages. Third, the mapping between higher-level programming paradigms

and collective operations is becoming a research area in its own right. Although a good deal of research is presently studying the use of collective communication in numerical libraries [10], and the relationship between data distribution patterns and the collective communication generated by compilers for data parallel languages [3], the potential payoffs from both areas are high, and increased research is needed. Finally, another trend in high-performance computing is towards clusters of workstations connected by high-speed switches. Many such environments offer some of the same features as wormhole networks, including pipelining of messages, concurrent input and output, concurrent message passing, and hardware-supported multicast communication. As such systems become more widespread, the need arises to explore how to best support collective operations in this new class of architectures [34].

## Acknowledgments

The authors would like to thank several faculty and students at Michigan State University whose work has either directly or indirectly contributed to this paper: Lionel M. Ni, Abdol-Hossein Esfahanian, Betty H. C. Cheng, Xiaola Lin, Hong Xu, Christian Trefftz, Chengchang Huang, Dan Judd, and Jeremy Uniacke. We would also like to thank those many researchers who have made contributions to this area and who have helped to promote the benefits of efficient collective communication. This work was supported in part by the NSF grants MIP-9204066, CDA-9121641, CDA9222901, by DOE grant DE-FG02-93ER25167, and by an Ameritech Faculty Fellowship.

## Further Information

A number of related papers and technical reports of the Communications Research Group at Michigan State University are available via anonymous `ftp`. The site is `ftp.cps.msu.edu` and the directory is `pub/crg`.

## References

- [1] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315–339, Dec. 1990.
- [2] D. B. Loveman, "High Performance Fortran," *IEEE Parallel and Distributed Technology*, vol. 1, pp. 25–42, Feb. 1993.
- [3] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 361–375, July 1991.
- [4] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Communications of the ACM*, pp. 66–80, Aug. 1992.
- [5] Message Passing Interface Forum, "Document for standard message-passing interface," Tech. Rep. CS-93-214, University of Tennessee, Nov. 1993.
- [6] W. J. Dally and C. L. Seitz, "The torus routing chip," *Journal of Distributed Computing*, vol. 1, no. 3, pp. 187–196, 1986.

- [7] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, pp. 62–76, Feb. 1993.
- [8] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni, "Unicast-based multicast communication in wormhole-routed networks," in *Proc. of the 1992 International Conference on Parallel Processing*, vol. II, pp. 10–19, Aug. 1992.
- [9] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, California: Benjamin/Cummings, 1994.
- [10] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–127, IEEE CS Press, 1992.
- [11] J. Dongarra, R. van de Geijn, and R. Whaley, "Two dimensional basic linear algebra communication subprograms," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 347–352, 1993.
- [12] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *IEEE Computer*, vol. 24, pp. 52–60, Aug. 1991.
- [13] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine," in *Proceedings of the 4th Annu. Symp. Comput. Architecture*, vol. 5, pp. 105–124, Mar. 1977.
- [14] S. L. Johnsson and C.-T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *IEEE Transactions on Computers*, vol. C-38, pp. 1249–1268, Sept. 1989.
- [15] P. K. McKinley and C. Trefftz, "Efficient broadcast in all-port wormhole routed hypercubes," in *Proc. of the 1993 International Conference on Parallel Processing*, vol. II, pp. 288–291, 1993.
- [16] C. Ho and M. Kao, "Optimal broadcast on hypercubes with wormhole and e-cube routings," in *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, (Taipei, Taiwan), pp. 694–697, Dec. 1993.
- [17] M. Barnett, D. G. Payne, and R. van de Geijn, "Optimal broadcasting in mesh-connected architectures," Tech. Rep. TR-91-38, Department of Computer Science, The University of Texas at Austin, Dec. 1991.
- [18] D. F. Robinson, P. K. McKinley, and B. H. C. Cheng, "Optimal multicast communication in torus networks," in *Proc. of the 1994 International Conference on Parallel Processing*, Aug. 1994. accepted to appear.
- [19] M. Barnett, D. G. Payne, R. A. van de Geijn, and J. Watts, "Broadcasting on meshes with worm-hole routing," Tech. Rep. TR-93-24, Department of Computer Science, The University of Texas at Austin, November 2 1993.
- [20] Y.-J. Tsai and P. K. McKinley, "An extended dominating node approach to collective communication in wormhole-routed 2D meshes," in *Proceedings of the Scalable High Performance Computing Conference*, pp. 199–206, 1994.
- [21] D. F. Robinson, D. Judd, P. K. McKinley, and B. H. C. Cheng, "Efficient collective data distribution in all-port wormhole-routed hypercubes," in *Proceedings of Supercomputing'93*, pp. 792–803, November 1993.
- [22] S. R. Seidel, "Circuit switched vs. store-and-forward solutions to symmetric communication problems," in *Proceedings of the 4th Conference on Hypercube Computers and Concurrent Applications*, pp. 253–255, 1989.
- [23] R. Thakur and A. Choudhary, "All-to-all communication on meshes with wormhole routing," in *Proceedings of the 1994 International Parallel Processing Symposium*, pp. 561–565, 1994.
- [24] S. H. Bokhari and H. Berryman, "Complete exchange on a circuit switched mesh," in *Proceedings of the 1992 Scalable High Performance Computing Conference*, pp. 300–306, Apr. 1992.
- [25] S. Gupta, S. Hawkinson, and B. Baxter, "A binary interleaved algorithm for complete exchange on a mesh architecture," tech. rep., Intel, Beaverton, Oregon.

- [26] D. P. Bertsekas, C. Özveren, G. D. Stamoulis, and J. N. Tsitsiklis, “Optimal communication algorithms for hypercubes,” *Journal of Parallel and Distributed Computing*, vol. 15, no. 11, pp. 263–175, 1991.
- [27] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, “The network architecture of the connection machine CM-5,” in *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures*, pp. 272–285, June 1992.
- [28] E. A. Brewer and B. C. Kuszmaul, “How to get good performance from the CM-5 data network,” in *Proceedings of the 1994 International Parallel Processing Symposium*, pp. 858–867, 1994.
- [29] R. Ponnusamy, R. Thakur, A. Choudhary, K. Velamakanni, Z. Bozkus, and G. Fox, “Experimental performance evaluation of the CM-5,” *Journal of Parallel and Distributed Computing*, vol. 19, pp. 192–202, 1993.
- [30] Z. Bozkus, S. Ranka, G. Fox, and A. Choudhary, “Performance comparison of the cm-5 and intel touchstone delta for dataparallel operations,” in *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, (Dallas, Texas), pp. 307–310, December 1993.
- [31] X. Lin, P. K. McKinley, and L. M. Ni, “Deadlock-free multicast wormhole routing in 2D mesh multicomputers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 793–804, Aug. 1994.
- [32] D. K. Panda and S. Singal, “Broadcasting in k-ary n-cube wormhole routed networks using path-based routing,” Tech. Rep. TR36., Ohio State University, Sept. 1993.
- [33] C.-T. Ho and M. Kao, “Optimal broadcast in all-port wormhole-routed hypercubes,” in *Proceedings of the 1994 International Conference on Parallel Processing*, Aug. 1994. accepted to appear.
- [34] C. C. Huang and P. K. McKinley, “Communication issues in parallel computing across ATM networks,” *IEEE Parallel and Distributed Technology*, 1994. accepted to appear.