Chapter 1

Practicalities

The subject of this book is the description, analysis, implementation, and application of collective communication algorithms on massively parallel parallel computers. In this chapter, we give an overview of the primary collective communications that will be treated. To facilitate analysis of the cost of the various operations, we introduce a simple parallel programming model to be used throughout the book. This will also allow us to put lower bounds on the cost of the operations, allowing us to evaluate the optimality of the various approaches. The cost of collective communications will be closely linked to the topology of the communication network. We will consider popular topologies, including hypercubes, linear arrays and rings, and higher dimensional meshes and tori. We have choosen to express our algorithms using the Message-Passing Interface, making the resulting codes portable among message-passing architectures. We give a rudimentary treatment of this interface at the end of this chapter.

1.1 A Model of Parallel Computation

The target architectures for our algorithms are distributed-memory multidimensional mesh multicomputers, including Multiple Instruction Multiple Data (MIMD) machines such as the Symult S2000, the Intel Touchstone Delta and Paragon systems, and the Cray T3D. For historic completeness, we also include hypercubes in our target architectures, which makes the results applicable to the Intel iPSC/860 and NCUBE/2.

Commercially available parallel systems change rapidly. Hence we will not explain our techniques as they are applied to particular machines, since the book would then be outdated before it was even published. Rather, we will present a simple programming model, which will be used to explain and evaluate the algorithms and issues.

1.1.1 Indexing of Processors

We assume the parallel architecture has p nodes. Often, we will index these nodes $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, regardless of the underlying architecture. Other times, it may become convenient to index them relative to a higher dimensional mesh. E.g., for two dimensional meshes, we may index the nodes $\mathbf{P}_{ij}$, where i and j indicate the row and column indices in the mesh, respectively.

1.1.2 Single-Ported Architectures

We assume that a node's network interface is single-ported. That is, at most one message can be sent and one message received at full node-to-network bandwidth. If k messages are received simultaneously, then the effective bandwidth into the node is reduced by a factor of k.

Many of the presented algorithms can be extended to multi-ported architectures, where more than one message can be sent of received simultaneously. Rather than treating these extensions in the book, we hope to present the techniques clearly enough that the reader can derive algorithms for multi-ported architectures.

1.1.3 Bidirectional Channels

Channels are assumed to be capable of transmitting a single message in each direction at full bandwidth. If k messages traverse a channel in the same direction, then the transmission time is increased by a factor of k.

1.1.4 Distance between two nodes

We will assume that messages between two nodes will always traverse the same prescribed path of physical links in the network. We will call the number of channels traversed between two nodes the *distance* between those two nodes.

1.1.5 Cost of Communication between Neighbors

We assume that the cost of sending/receiving a length n message to/from a physical neighbor can be modelled by:

$$\alpha + n\beta$$

where α is a fixed-cost message overhead or startup time independent of the length of the message, and β is the transmission cost per unit of data.

1.1.6 Pipelining of Messages

Let us assume that we wish to send a message between two nodes that are not physically neighbors, i.e., they are a distance s apart. Routing a message from between these two nodes could be accomplished by sending the message from neighboring node to neighboring node along the route prescribed by the routing algorithm, treating the message as a unit. The cost for a message of length n would become

$$s\alpha + sn\beta$$

For long message, we can do considerably better: First, the message is broken into k pieces where for our explanation it suffices to assume n is an integer multiple of k. Sending the message between the same two nodes can now be accomplished by *pipelining* the packets: The source node starts by sending the first n/k pieces of data to the first neighbor along the path. While this neighboring node passes the data to its neighbor, the root sends the next n/k pieces, etc. The time required for the first packet to reach the last node is $(s)(\alpha + n/k\beta)$, after which $k-1$ more packets must arrive, requiring time $\alpha + n/k\beta$ each, for a total time of

$$\begin{aligned} T_{\text{pipe}}(n,s,k) &= (s+k-1)\left(\alpha + \frac{n}{k}\beta\right) && (1.1) \\ &= (s+k-1)\alpha + \frac{s+k-1}{k}n\beta && (1.2) \end{aligned}$$

Notice that *if* the message is very large, the β term will start dominating and *if* k is picked large enough, $(p+k-1)/k \rightarrow 1$, so that asymptotically the cost of broadcasting the message becomes $n\beta$. We can compute the optimal number of packets, k_{opt} by pretending that the above equation is a continuous function of k, and differentiating with respect to that variable.

$$\frac{dT(n,s,k)}{dk} = \alpha - \frac{s-1}{k^2} n\beta$$

Observing that for $k > 0$ the function is convex, and hence has a unique minimum, allows us to solve for the optimal k by setting the derivative to zero:

$$\alpha - \frac{s-1}{k_{\text{opt}}^2} n\beta = 0 \Rightarrow k_{\text{opt}} = \sqrt{(s-1)n\frac{\beta}{\alpha}}$$

Naturally, $k_{\text{opt}} < 1$ or $k_{\text{opt}} > n$ doesn't make sense, so that the complete formula becomes

$$k_{\text{opt}}(n,s) = \begin{cases} 1 & \text{if } \sqrt{(s-1)n\frac{\beta}{\alpha}} < 1 \\ n & \text{if } \sqrt{(s-1)n\frac{\beta}{\alpha}} > n \\ \sqrt{(s-1)n\frac{\beta}{\alpha}} & \text{otherwise} \end{cases}$$

This gives a total time for sending the message a distance s of

$$T_{\text{pipe}}(n,s,k_{\text{opt}}(n,s)) = \begin{cases} s\alpha + sn\beta & \text{if } \sqrt{(s-1)n\frac{\beta}{\alpha}} < 1 \\ (s+n-1)(\alpha+\beta) & \text{if } \sqrt{(s-1)n\frac{\beta}{\alpha}} > n \\ (\sqrt{(s-1)\alpha} + \sqrt{n\beta})^2 & \text{otherwise} \end{cases}$$

Finally, if the distance s is fixed,

$$\lim_{n\to\infty} T_{\text{pipe}}(n,s,k_{\text{opt}}(n,s)) = n\beta$$

Thus for long messages, the cost of sending a message between two nodes using this kind of pipelining approach is essentially independent of the distance between the two nodes.

1.1.7 Model of Communication Cost

This is a good time to note that in current generation networks, there is a cost for introducing a message into the network, and a cost for forwarding a message at each hop, which is handled by hardware and is typically several orders of magnitude smaller. As a result, a more accurate model for sending a message a distance s using k packets is

$$T(k,n,s) = \alpha + (s+k-1)(\alpha_{\text{net}} + \frac{n}{k}\beta)$$

with $\alpha_{\text{net}} \approx 0$. In this case, $k_{\text{opt}} \approx 0$, and, since typically $\alpha >> \beta$ the time for sending a message a distance s becomes essentially $\alpha + n\beta$. This allows us to claim that if no network conflicts occur, we can model the time for sending a message between any two nodes in the network by

$$\alpha + n\beta$$

This assumption models networks that include hardware cut-through routing or worm-hole routing can be models quite accurately by this simple model.

1.1.8 Network Conflicts

Notice that since messages are assumed to be pipelined, leading to the cost model $\alpha + n\beta$, our model essentially assumes that the whole path between two nodes is occupied during the communication. Thus, if a single link accomodates more than one messages, e.g. if two paires of nodes are communicating, and the paths share this link, network conflicts occur. Thus, in the presence of network conflict, we will model the cost of sending a message between two nodes as

$$\alpha + kn\beta$$

where k equals the maximum over all links in the path of the number of conflicts on those links.

1.1.9 Cost of Computation

We will denote the time required for a typical arithmetic operation, e.g. a multiply or add, as γ.

1.1.10 Summary of Assumptions

1.2 Common Collective Communications

To describe the common collective communications that we will treat in this book, we start by introducing some notation.

Let x, $x^{(j)}$, and y be vectors of data of length n. For many of the operations, we will need to partition these vectors into p pieces, where p equals the number of nodes. Thus

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right), \quad x^{(j)} = \left(\begin{array}{c} x_0^{(j)} \\ \hline x_1^{(j)} \\ \hline \vdots \\ \hline x_{p-1}^{(j)} \end{array} \right) \quad \text{and} \quad y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

where x_i, $x_i^{(j)}$, and y_i are of length $n_i \geq 0$, with $\sum_{i=0}^{p-1} n_i = n$. Also, given a associative and commutative *element-wise* operator $\oplus$, we will assume that

$$y = x^{(0)} \oplus x^{(1)} \oplus \cdots \oplus x^{(p-1)}$$

Given this notation, the various collective operations can now be defined.

The common collective communications are now given by

Broadcast: Initially, one node, $\mathbf{P}_k$, owns a vector of data, x. After the broadcast, all nodes own a copy of x. This communication is often referred to as a one-to-all broadcast. The MPI call is `MPI_Bcast`.

Scatter: Initially, one node, $\mathbf{P}_k$, owns a vector of data, x. After the scatter, each node, $\mathbf{P}_i$, owns a piece of the data x_i, $i = 0, \ldots, p-1$. This communication is ofter referred to as a one-to-all personalized broadcast. The MPI call is **MPI_Scatter**.

Gather: Before the gather, each node, $\mathbf{P}_i$, owns a piece of the data x_i, $i = 0, \ldots, p-1$. Afterwards, one node, $\mathbf{P}_k$, owns the entire vector of data, x. The MPI call is **MPI_Gather**.

Collect: Initially, all nodes have a piece of the data: x_i is owned by $\mathbf{P}_i$. After the collect, all nodes own all of the date, x. This communication is often referred to as an all-to-all (simultaneous) broadcast, or a simultanious gather. The MPI call is **MPI_Allgather**.

Operation	BEFORE						AFTER					
	$\mathbf{P}_0$	$\mathbf{P}_1$	$\cdots$	$\mathbf{P}_k$	$\cdots$	$\mathbf{P}_{p-1}$	$\mathbf{P}_0$	$\mathbf{P}_1$	$\cdots$	$\mathbf{P}_k$	$\cdots$	$\mathbf{P}_{p-1}$
Broadcast				x			x	x	x	x	x	x
Reduce-to-one	$x^{(0)}$	$x^{(1)}$	$\cdots$	$x^{(k)}$	$\cdots$	$x^{(p-1)}$				y		
Scatter				x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}			x_0	x_1	$\ddots$	x_k	$\ddots$	x_{p-1}
Gather	x_0	x_1	$\ddots$	x_k	$\ddots$	x_{p-1}				x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}		
Reduce-to-all	$x^{(0)}$	$x^{(1)}$	$\cdots$	$x^{(k)}$	$\cdots$	$x^{(p-1)}$	y	y	y	y	y	y
Collect	x_0	x_1	$\ddots$	x_k	$\ddots$	x_{p-1}	x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}	x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}	$\cdots$ $\cdots$ $\cdots$ $\cdots$	x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}	$\cdots$ $\cdots$ $\cdots$ $\cdots$	x_0 x_1 $\vdots$ x_k $\vdots$ x_{p-1}
Distr. Red.	$x_0^{(0)}$ $x_1^{(0)}$ $\vdots$ $x_k^{(0)}$ $\vdots$ $x_{(0)p-1}$	$x_0^{(1)}$ $x_1^{(1)}$ $\vdots$ $x_k^{(1)}$ $\vdots$ $x_{(1)p-1}$	$\cdots$ $\cdots$ $\cdots$ $\cdots$	$x_0^{(k)}$ $x_1^{(k)}$ $\vdots$ $x_k^{(k)}$ $\vdots$ $x_{(k)p-1}$	$\cdots$ $\cdots$ $\cdots$ $\cdots$	$x_0^{(p-1)}$ $x_1^{(p-1)}$ $\vdots$ $x_k^{(p-1)}$ $\vdots$ $x_{p-1}^{(p-1)}$	y_0	y_1	$\ddots$	y_k	$\ddots$	y_{p-1}

Reduce-to-one: Initially, all nodes, $\mathbf{P}_j$, own a vector on length n, $x^{(j)}$. After the reduce-to-one, the result $y = x^{(0)} \oplus \cdots \oplus x^{(p-1)}$ is owned by one node, $\mathbf{P}_k$. The MPI call is **MPI_Reduce**.

Reduce-to-all: Identical to reduce-to-one, except that the result is duplicated on all nodes. The MPI call is **MPI_Allreduce**.

Distributed reduce: Identical to reduce-to-one, except that each node, $\mathbf{P}_i$ owns a portion of the final result, y_i.

All-scatter: This operation is equivalent to performing a scatter from every node. It is often called a all-to-all personalized broadcast. More on this operation in Chapter ??. The MPI call is **MPI_Alltoall**.

1.3 Lower bounds

Before launching into the discussion of collective communication algorithms, it is very useful to present lower bounds on the time required in order to better gauge the algorithms. In this section, we give handwaving arguments to derive these lower bounds. We will distinguish between three components of communication cost: terms due to latency, bandwidth, and computation. A table of lower bounds is given in Table 1.1.

Communication	Latency	Bandwidth	Computation
Broadcast	$\lceil log_2(p) \rceil \alpha$	$n\beta$	–
Reduction-to-one	$\lceil log_2(p) \rceil \alpha$	$n\beta$	$\frac{p-1}{p} n\gamma$
Scatter	$\lceil log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	–
Gather	$\lceil log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	–
Collect	$\lceil log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	–
Distributed Reduction	$\lceil log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$
Reduction-to-all	$\lceil log_2(p) \rceil \alpha$	$2\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$
All-scatter	$\lceil log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	–

Table 1.1: Lower bounds for the different components of communication cost. Pay particular attention to the conditions for the lower bound, given in the text

Latency. The lower bound on latency is derived by the simple observation that for all collective communications, at least one nodes has data that must somehow arrive at all other nodes. Under our model, at each step, we can at most double the number of nodes that get that data.

Computation. Only the reduction communications require computation. The computation involved would require $(p-1)n$ computations if executed on a single node, or time $(p-1)n\gamma$. Distributing this computation perfectly among the nodes reduces the time to $(p-1)n\gamma/p$ under ideal circumstances. Hence the lower bound.

Bandwidth. For the broadcast and reduction-to-one, the root node must either send or receive all n item, repectively. The cost of this is bounded below by $n\beta$. For the scatter and gather, the root node must either send or receive $\frac{p-1}{p} n$ items, with a cost of at least $\frac{p-1}{p} n\beta$. The same is the case for all nodes during the collect and distributed reduction.

The reduction-to-all is somewhat more complicated: *If the lower bound on computation is to be achieved,* one can argue that $\frac{p-1}{p}n$ items must leave each node, and $\frac{p-1}{p}n$ items must be received by each node after the computation is completed, for a total cost of at least $2\frac{p-1}{p}n\beta$.

We will discuss lower bounds for the all-scatter in Chapter ??.

1.4 Communication Networks Considered

1.4.1 Hypercubes

Definition

In general, a hypercube has $p = 2^d$ nodes, where d is the dimension of the cube. We introduce hypercubes by showing how we can build large hypercubes from small ones, in Figure 1.1.

An immediate consequence of the indexing scheme shown in the figure is that in a hypercube two nodes are neighbors if and only if the binary representation of their indices differ in exactly one bit.

We now give a definition of a hypercube network:

Definition 1 *The communication network is a hypercube has the property that two nodes* $\mathbf{P}_i$ *and* $\mathbf{P}_j$ *are nearest neighbors if and only if the binary representations of* i *and* j *differ in at most one bit.*

Properties

Given two nodes $\mathbf{P}_i$ and $\mathbf{P}_j$, it is relatively easy to generate a path between them: Consider the binary representations of i and j:

$$\begin{array}{lcccc} i \rightarrow & i_{d-1} & \cdots & i_1 & i_0 \\ j \rightarrow & j_{d-1} & \cdots & j_1 & j_0 \end{array}$$

One algorithm for finding a path between the two nodes is

do $k = 0, \ldots, d-1$
  if $i_k \neq i_j$
    send message from
      $i_{d-1} \quad \cdots \quad i_k \quad j_{k-1} \quad \cdots \quad j_0$
    to
      $i_{d-1} \quad \cdots \quad i_{k+1} \quad j_k \quad \cdots \quad j_0$
  fi
od

In most hypercubes architectures this routing is done in hardware, using pipelining as discussed previously.

It is not hard to see that a shortest path will only consider each dimension of the hypercube once, i.e. each bit is toggled at most once in a routing algorithm. Notice that there are up to $d!$ different shortest paths between two nodes. If the order in which the bits are considered is always fixed, the resulting routing algorithm is often called dimensional routing.

The *network diameter*, defined as maximum over all node pairs of length of the shortest path between them is only the dimension of the hypercube.

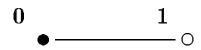
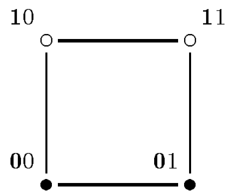
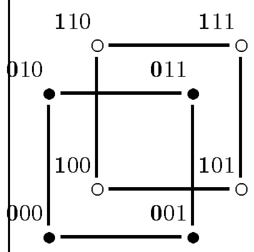
dim. d	# of nodes p	comment	
0	1	A single node.	
1	2	Duplicate the 0 dimensional cube, and connect corresponding nodes with a link.	
2	2^2	Duplicate the 1 dimensional cube, and connect corresponding nodes with a link.	
3	2^3	Duplicate the 2 dimensional cube, and connect corresponding nodes with a link.	
$\vdots$	$\vdots$	$\vdots$	$\vdots$
d	2^d	Duplicate the $d-1$ dimensional cube, and connect corresponding nodes with a link.	$\vdots$

Figure 1.1: Construction of hypercubes

1.4.2 Linear Arrays and Rings

Definition

A network forms a linear array if there exists a link between $\mathbf{P}_i$ and $\mathbf{P}_j$ if and only if $i = j+1$ or $j = i+1$. If in addition there is a link between $\mathbf{P}_{p-1}$ and $\mathbf{P}_0$, then the linear array becomes a ring. If each link in the ring is only unidirectional, e.g. there is a unidirectional link from $\mathbf{P}_i$ to $\mathbf{P}_{(i+1) \bmod p}$, then the ring is said to be a unidirectional ring.

Properties

Notice that under our assumptions that messages traveling in opposite directions on a link do not conflict, all linear arrays are also unidirectional rings.

The network diameter for a linear array is $p-1$, and for a ring is $\lfloor p/2 \rfloor$. **check this!**

1.4.3 Meshes and Tori

A natural extention to linear arrays and rings are higher dimensional meshes and tori.

Definition

A network is said to be a two dimensional mesh if there is an indexing of the nodes $\mathbf{P}_{i,j}$, $0 \leq i < r$ and $0 \leq j < c$ such that P_{i_1,j_1} is a neighbor of P_{i_2,j_2} if and only if one of the following is true

$$i_1 = i_2 \quad \text{and} \quad j_2 = j_1 + 1, \quad \text{or} \tag{1.3}$$
$$i_1 = i_2 \quad \text{and} \quad j_1 = j_2 + 1, \quad \text{or} \tag{1.4}$$
$$j_1 = j_2 \quad \text{and} \quad i_2 = i_1 + 1, \quad \text{or} \tag{1.5}$$
$$j_1 = j_2 \quad \text{and} \quad i_1 = i_2 + 1, \quad \text{or} \tag{1.6}$$

Parameters r and c will be referred to as the row and column dimensions of the mesh, with $p = r \times c$. i and j are the row and column indices, respectively. If in addition, there are "wrap-around" connections like for the ring, the mesh becomes a torus. A unidirectional torus is a torus with unidirectional links.

The order in which message are routed is often fixed. We will assume that all messages take the shortest path between two nodes, traversing the x-direction first, followed by the y-directions. In other words, a message between two nodes will travel from the source within the row in which the source lies to the column in which the destination lies, after which it is routed to the destination within that column.

This definition naturally generalizes to higher dimensions.

Properties

Notice that a $2 \times 2 \times \cdots \times 2$ mesh is a hypercube. Under our model, any mesh can be viewed as a unidirectional torus.

1.5 Viewing Topologies as Higher Logical Meshes

1.6 MPI

The paradigm we will use to express our algorithms is generally known of the Single Program-Multiple Data (SPMD) paradigm: An identical program drives all nodes, but the path through the program is determined by the index of the the node. In addition, we will use the *explicit message passing paradigm.* Nodes exchange data by the explicit passing of messages between them.

Whenever practical, we will give code that will execute on current generation distributed memory parallel architectures. We do so by expressing our algorithms in the C programming language and using the Message Passing Interface (MPI). This interface is a recently adopted industry standard for message passing routines.

A more complete discussion on using MPI is given in the book *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* by Gropp, Lusk, and Skjellum and the more recent book *MPI: the complete reference,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra. In this section, we give the minimal background required to express our algorithms. Only in the chapter on optimizing the presented implementations will we discuss more advanced MPI features.

1.6.1 Communicators

The MPI standard uses the concept of a communicator to describe the framework on which a message passing operation is to be executed. The communicator is an object that is opaque to the user, yet contains information describing the *group* and *context* of operation.

A communicator `comm` is declared by

MPI_Comm *comm*;

A *group* defines the list of nodes (or members) that ore the communication participants for the communicator. Any member in the group can communicate with any other member in the group. However, communication between any member in the group and a node in the outside world is not permitted.

A *context* defines the communication space for the communicator. A member of the group can receive a given message only if the message has the same context as the communicator context. The context can be as simple as an integer that is unique between communicators or it can be as complex as virtual communication channels that are unique between communicators.

Question: do we ever use contexts in our book? If not, let's not even mention it.

Two communicators may share the same group, but have two seperate contexts. This allows the same group to perform two seperate message passing operations concurrently, without interfering with each other. For instance, all members of a group may do a broadcast using communicator A, except that two members do nonblocking send and receive using communicator B, followed immediately by a broadcast using communicator A. The nonblocking point to point operation will not interfere with the broadcast operation, since seperate contexts are being used for messages in each operation. In the same manner, message passing operations that act concurrently on overlapping groups will not interfere with each other.

It is important for the reader to understand the purpose of the communicator, since it will show up in all of the MPI calls used in the code of this book. However, for further details on the MPI calls that create and free groups and communicators, we refer the reader to either of the above mentioned references.

1.6.2 Information Calls

The MPI communicator contains more useful information in its group structure than just the list of participating members. A member's rank in the group as well as the size of the group are also found in the group structure.

Many algorithms need to know the group size and member rank for message passing, and our algorithms are no exception. As a result, MPI provides functions that will return this information when executed. The calls are:

MPI_COMM_SIZE(*comm, size*)
MPI_COMM_RANK(*comm, rank*)

MPI_COMM_SIZE returns the *size* of the group associated with the communitor *comm*. **MPI_COMM_RANK** returns the *rank* of the group associated with the communicator *comm*. We will use these throughout the code in this book as necessary.

1.6.3 Point to Point Communication

In order to implement a point to point data communication, information about the source, destination, and data to be transfered are required. In MPI, the following parameters are frequently refered to in sending and receiving data.

buf	*send or receive buffer*
count	*number of elements in buf*
datatype	*datatype of the elements in buf*
dest	*destination rank*
source	*source rank*
tag	*message tag*
comm	*communicator*
status	*status structure for received message*
request	*nonblocking handle for test and wait routines*

MPI provides several ways to do point to point data communication between nodes. There is one type of receive operation and four types of send operations, shown below. The send operations differ only in the protocol semantics of when the operation is considered complete.

Blocking:

MPI_SEND(*buf, count, datatype, dest, tag, comm*)
MPI_BSEND(*buf, count, datatype, dest, tag, comm*)
MPI_SSEND(*buf, count, datatype, dest, tag, comm*)
MPI_RSEND(*buf, count, datatype, dest, tag, comm*)
MPI_RECV(*buf, count, datatype, source, tag, comm, status*)

MPI_BSEND is the buffered send operation. It is considered complete when the message envelope and data have been safely tucked away in a preallocated buffer so that the user can overwrite *buf*. It is considered a local operation, since there is no requirement of the destination node to commit to receive it.

MPI_SSEND is the synchronous send operation. It is considered complete only when a receive has been posted at the destination and the data is being read. At this point, the *buf* can be overwritten. Since the completion depends on the posting of a receive at the destination, this operation is considered non-local.

MPI_RSEND is the ready send operation. In this operation, it is **assumed** that a receive as been posted at the destination and the message is sent as soon and as quickly as possible. There is no buffering. Thus, the ready send operation completes as soon as *buf* can be reused. Since the send depends on a receive being posted at the destination, the operation is considered non-local. If a receive has not been posted at the destination, then the operation is erroneous and the outcome is undefined.

MPI_SEND is the standard send operation. It may or may not buffer the envelope and data depending on the availability of buffer space. If buffer space is available, then the operation can complete as soon as the envelope and data have been safely tucked away. Otherwise, the operation must wait for a posted receive at the destination. Since, there may be a dependency on having a receive posted at the destination, this operation is considered non-local.

MPI_RECV is the standard receive operation. It can receive messages from any of the above send operations and it completes as soon as all of the data has been deposited into *buf*.

The **Blocking** descriptor above refers to the fact that these calls do not return until the send or receive operation is completed. Their nonblocking counterparts are described below.

Nonblocking:

MPI_ISEND(*buf, count, datatype, dest, tag, comm, request*)
MPI_IBSEND(*buf, count, datatype, dest, tag, comm, request*)
MPI_ISSEND(*buf, count, datatype, dest, tag, comm, request*)
MPI_IRSEND(*buf, count, datatype, dest, tag, comm, request*)
MPI_IRECV(*buf, count, datatype, source, tag, comm, request*)

Test, Probe, and Wait:

MPI_WAIT(*request, status*)
MPI_TEST(*request, flag, status*)
MPI_REQUEST_FREE(*request*)
MPI_WAITANY(*count, array_of_requests, index, status*)
MPI_TESTANY(*count, array_of_requests, index, flag, status*)
MPI_WAITALL(*count, array_of_requests, array_of_statuses*)
MPI_TESTALL(*count, array_of_requests, flag, array_of_statuses*)
MPI_WAITSOME(*incount, array_of_requests, outcount, array_of_indices, array_of_statuses*)
MPI_TESTSOME(*incount, array_of_requests, outcount, array_of_indices, array_of_statuses*)

The nonblocking calls are included in order to provide the user a way to overlap communication and computation. The nonblocking receive may return before the data has arrived to allow the process to continue doing useful work. When the data is actually needed, **MPI_WAIT** can be used to block until the message arrives. The *request* parameter passed to **MPI_WAIT** originally came from one of the non-blocking calls and is used to specify the message on which to wait. **MSG_WAIT** will wait on send operations as well as receive operations.

The user may also want to simply test periodically to see if the operation has completed. If the test fails, then the user can go on doing other useful work. In this scenario, the user should consider using **MPI_TEST**. The *flag* parameter returns TRUE or FALSE to signal whether the operation has completed. **MSG_TEST** will test both send and receive operations.

The semantics of the nonblocking routines are the same as their blocking counterparts, including the operation completion semantics. For instance, although **MPI_SSEND** may return before a receive has been posted at the destination, the subsequent call to **MPI_WAIT** will not complete until such a receive has been posted.

Notice that MPI provides a number of test and wait routines. All of them are functional variations of **MPI_TEST** and **MPI_WAIT**. For a more detailed description of these other routines, refer to *MPI: A Message-Passing Interface Standard.*

1.6.4 Datatypes

The user must specify the *datatype* of the elements being passed into a send or receive. MPI provides a number of useful elementary datatypes that match FORTRAN and C elementary datatypes. MPI also provides a mechanism for forming new datatypes, but we will not be doing this in the examples of this book. The MPI elementary datatypes for the C language follow.

MPI_CHAR	`char`
MPI_SHORT	`short`
MPI_INT	`int`
MPI_LONG	`long`
MPI_UNSIGNED_CHAR	`unsigned char`
MPI_UNSIGNED_SHORT	`unsigned short`
MPI_UNSIGNED_INT	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long`
MPI_UNSIGNED_DOUBLE	`unsigned double`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_BYTE	`long byte`

1.7 Collecting and Presenting Timing Data

Part I

Collective Communication

Chapter 2

Short Vector Algorithms

We start by giving an overview of basic techniques that have been proposed for implementing the different collective communication routines. In this chapter, we concentrate on methods that are appropriate when little data is being communicated. The next chapter deals with cases where large volumes of data are being transmitted.

For each of the operations, we address implementation on hypercubes, linear arrays and meshes. We carefully analyse the cost, which will help us later to make an assessment of the relative strengths and weaknesses of particular implementations.

2.1 Broadcast

Recall that a broadcast begins with a single node holding a vector of data, which is left duplicated on all nodes at the completion of the broadcast. In this section, we discuss the most widely used techniques for implementing the broadcast on a network of nodes. This technique minimizes the number of steps, and hence latency, making it appropriate when the data consists of short vectors.

2.1.1 Hypercubes

The most popular broadcast algorithm is based on embedding a minimum spanning tree from the node originating the broadcast (the root) to all other nodes. Assuming that the number of nodes $p = 2^d$ for some integer d, the broadcast can be achieved by performing the following: For $k = 0, \ldots, \log(p) - 1$, all nodes $\mathbf{P}_i$ that currently have the data send the data to the node $\mathbf{P}_{\mathrm{nghbr(i,k)}}$. After each step, the maximal number of nodes have the data [**refer to lower bound section here**], which allows us to argue that this method implements the broadcast in the minimal number of steps, thereby minimizing the latency. It can be easily seen that this algorithm creates no network conflicts on hypercubes and hence its cost on hypercubes is given by

$$T_{\mathrm{MST-bcast}}^{\mathrm{cube}} = \log(p)\alpha + \log(p)n\beta.$$

We illustrate the MST broadcast on an eight node hypercube in Fig. 2.1.

2.1.2 Linear Arrays

An interesting way of viewing the nodes of the hypercube is to present them as a linear array as in Figure ??. From this figure, it is immediately apparent that on a linear array of nodes, the given minimum spanning tree broadcast will incur considerable network conflicts. Indeed, under our model the execution time is given by

$$T^{1D}_{\mathrm{MST-bcast-naive}} = \sum_{k=0}^{\log(p)-1} \left[\alpha + 2^k n\beta\right] = \log(p)\alpha + (p-1)n\beta$$

when there is no excess bandwidth in the network.

The network conflicts can be easily remedied by observing that, on a hypercube, the order in which the dimensions of the hypercubes are chosen for the minimum spanning tree is arbitrary. If the order is chosen as most significant to least significant bit, we see that on the linear array the MST broadcast can be viewed as recursively dividing the network in half, where at each stage a message is sent from the current root to the corresponding node in the other half of the array. Afterwards, both the original root and node to which it sent act as roots within the separate halves of the network. This process is illustrated in Figure ??. It can now be seen that no network conflicts occur during the broadcast, and the exection time is identical to that on a hypercube.

The MST broadcast on linear arrays can now be easily extended to the case where the number of nodes is not a power of two. At each recursive stage of the algorithm, the network can be divided in two nearly equal parts, with the root sending to one of the nodes in the part that does not contain the root. Again, each of these nodes then becomes a root for the subsequent recursive steps. This process is illustrated in Figure ?? and the algorithm is given in Figure ??. We will call this broadcast a recursive splitting broadcast. Since at each stage the receiving node is not specified, there is actually a family of recursive splitting broadcasts. The time for these algorithms now becomes

$$\lceil \log_2(p) \rceil \alpha + \lceil \log_2(p) \rceil n\beta$$

It can be easily argued that this algorithm is optimal in the sense that it incurs the minimum number of startups, making it particularly appropriate for the case where little data is being broadcast.

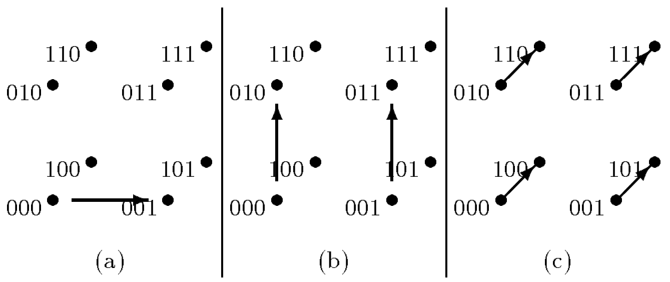


Figure 2.1: MST BCAST broadcast for 3D-cube.

2.1.3 Meshes

We first assume that we have a mesh with power-of-two number of nodes in each dimension. In this case, the minimum spanning tree that toggles bits from the most significant to least significant bit also does not incur network conflicts. Indeed, for a 2D mesh, with the nodes ordered row-major, this approach will perform a linear array broadcast within the column that holds the root, followed by a linear array broadcast within each row, with as root the node that is in the column of the original root. This is illustrated in Figure ??. For higher dimensioned meshes, one can easily show a similar effect.

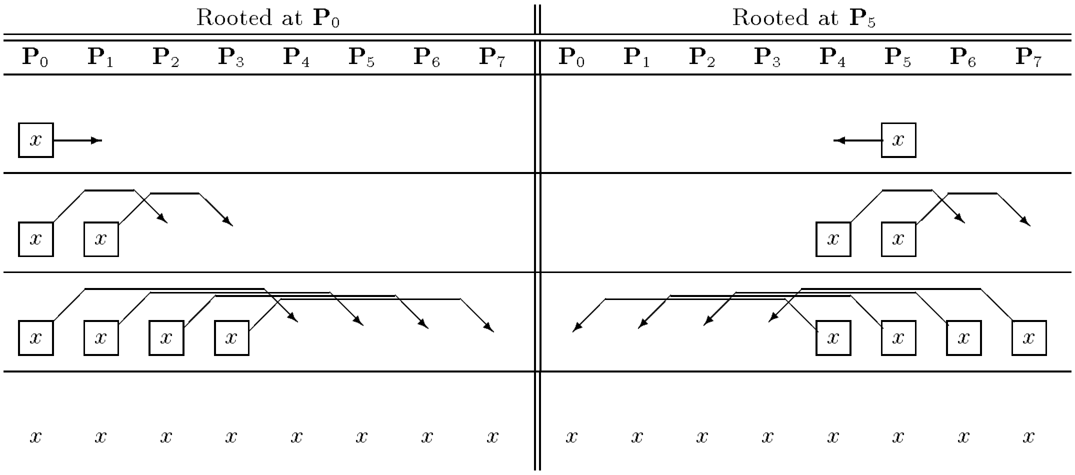


Figure 2.2: Broadcast on a hypercube and power-of-two linear array, toggling bits from least significant to most significant.

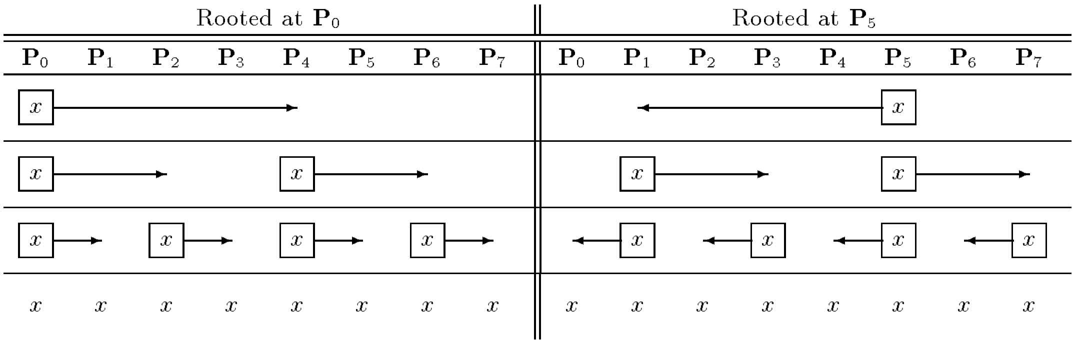


Figure 2.3: Broadcast on a hypercube and power-of-two linear array, toggling bits from most significant to least significant.

```
RSbcast_lin_rec( x, n, left, right, root, comm )
char     *x;             /* Address of data to be sent              */
int      n,              /* Length                                  */
         left, right,    /* current left and right index            */
         root;           /* current root                            */
MPI_COMM comm;           /* Communicator.                           */
{
  int  me, mid, dest,    /* holds me, mid, dest index               */
       stat(MPI_STATUS_SIZE), ierr;

  if ( left == right ) return;
  MPI_Comm_rank( comm, &me );
                /* determine midpoint.                              */
  half = ( right-left )/2;
  mid =  ( left+half < root ? left+half : right-half);
                /* determine dest                                   */
  if ( mid <= root ) dest = ( root == right ? mid-1 : left );
  else dest = ( root == left ? mid : right );
                /* send                                             */
  if ( me==root )
    MPI_Send( x, n, MPI_BYTE, dest, MPI_ANY_TAG, comm, stat, &ierr );
  if ( me == dest )
    MPI_Recv( x, n, MPI_BYTE, root, MPI_ANY_TAG, comm,  stat, &ierr );
                /* recursively call broadcast in each partition     */
  RSbcast_lin_rec( x, n, ( me<mid ? left : mid ),
                         ( me<mid ? mid-1 : right ),
                         ( ( me<mid ) == ( root<mid ) ? root : dest ),
                         comm );
}
```

Figure 2.4: Recursive splitting broadcast for linear arrays, formulated as a recursive algorithm.

Notice that for the power-of-two case, the higher dimensional mesh can simply be viewed as a linear array on which a MST broadcast is performed. For the non-power-of-two meshes, we can do the same: view the nodes as a linear array, with some reasonable ordering, and perform a recursive-splitting broadcast. It should be noted that we cannot in general guarantee that no network conflicts occur for the non-power-of-two case. For short vectors, we need not worry about this too much. However, we do address this in the section on long vectors, for the scatter and gather operations. **[Why don't we present a conflict-free algorithm here? Why open ourselves up to criticism? In all of the other short vector algorithms, we always give the full cost–including contention. We should do that here as well.]**

2.1.4 Code

In this section, we present code for the discussed broadcast algorithms. Interestingly enough, all the implementations that avoid network conflicts on linear arrays and meshes can be captured by a single recursive algorithm, with only a few minor modifications to get the different implementations.

We will go through this code line by line, to make sure it is completely understood.

[There are several major inconsistencies between the explanation of the code and the code itself. We should also consider making it closer to MPI–this won't add much complexity, but will make it's use in MPI much easier.]

```
RSbcast_lin_rec( x, n, left, right, root, comm )
char     *x;           /* Address of data to be sent             */
long     n,            /* Length                                 */
         left, right,  /* current left and right index           */
         root;         /* current root                           */
MPI_COMM comm;         /* Communicator.                          */
```

The naming convention indicates this is a recursive splitting broadcast, viewing the nodes logically as a linear array, and it is a recursively defined implementation. The arguments are:

- `x`, the address of the buffer where the data to be broadcast resides on the root, and where it is to be received on the other participating nodes,
- `n`. the number of bytes to be broadcast,
- `left, right`, the left and right indices of the partition of the linear array on which the broadcast is to proceed,
- `root`, the root of the broadcast,
- `comm`, the communicator, which holds information about the group and context, as discussed in Section ??.

```
  int  mid,            /* holds mid index                         */
       np, me,         /* nprocs. in group, my rank, hyp. dim.    */
       stat(MPI_STATUS_SIZE), ierr;
```

This code section declares the local variables to be used.

- `mid`, a variable to hold the midpoint of the partition,
- `np, me`, variables that will hold the number of processors in the group and the index of the processor executing the code,
- `stat, ierr`, will hold information returned by the MPI send and receive routines.

```
  if ( left == right ) return;
```

If the partition contains only one node, no action is required.

```
  size_rank( comm, &np, &me);
```

This is a utility routine that returns the number of nodes in the group and the relative index of this processor.

```
half = ( right-left )/2;
mid =  ( left+half < root ? left+half : right-half);
```

This piece of code computes the index of the node that is the midpoint of the current partition. If the number of nodes in the partition is odd, it guarantees that the root is in the larger subpartition. Next, we determine the destination node in the subpartition that does not contain the root. For different architectures, this choice will differ. For example, for a hypercube, the destination is the node that holds the same relative position in the subpartition as the root does in its subpartition:

```
if ( mid <= root ) dest = left + root - mid;
else dest = mid + root - left;
```

This choice will work on a power-of-two linear array as well. On a general linear array, the destination can be any of the nodes in the subpartition that does not contain the root. For meshes, if the vector length is short, this strategy will work as well. However, if network conflicts are to be avoided, we need to have all messages that don't originate from the original root of the broadcast flow towards the original root. This is accomplished by choosing the destination as

```
if ( mid <= root ) dest = ( root == right ? mid-1 : left );
else dest = ( root == left ? mid : right );
```

Once the destination has been chosen, a message is sent between current root and the destination:

```
if ( me==root )
  MPI_Send( &x, n, MPI_BYTE, dest, MPI_ANY_TAG, comm, stat, &ierr );
if ( me == dest )
  MPI_Recv( &x, n, MPI_BYTE, root, MPI_ANY_TAG, comm,  stat, &ierr );
```

and next, the broadcast is called recursively within each subpartition, with the current root and the current destination as subsequent roots:

```
if ( me==root )
  MPI_Send( &x, n, MPI_BYTE, dest, MPI_ANY_TAG, comm, stat, &ierr );
if ( me == dest )
  MPI_Recv( &x, n, MPI_BYTE, root, MPI_ANY_TAG, comm,  stat, &ierr );
```

To use this recursive subroutine, an application would call the subroutine with `left` and `right` equal to `0` and `np-1`, respectively, where `np` equals the number of nodes in the group in which the broadcast is to be accomplished. This group is assumed to physically form a hypercube, linear array, or mesh, with the required ordering of nodes, as specified in ?:

```
RSbcast( x, n, root, comm )
char     *x;            /* Address of data to be sent           */
long     n,             /* Length                               */
         root;          /* current root                         */
MPI_COMM comm;          /* Communicator.                        */
{
  int  np;              /* holds number of processors in group  */

  MPI_Comm_size( comm, &np);
  RSbcast_lin_rec( x, n, 0, np-1, root, comm );
}
```

Figure 2.5: Recursive splitting broadcast for linear arrays, which calls the recursive algorithm.

```
  RSbcast_lin_rec( x, n, ( me<mid ? left : mid ),
                         ( me<mid ? mid-1 : right ),
                         ( ( me<mid ) == ( root<mid ) ? root : dest ),
                         comm );
```

To make this simpler, we suggest the wrapper that does this automatically, given in Figure 2.5.

2.2 Reduction-to-One

A very similar operation to the broadcast is the reduction-to-one operation. In this routine, each node contributes a vector, all of which are combined with a user-supplied operation, and the result is deposited on a single, specified node. As one might suspect, efficient implementations of this routine mirror implemenations of the broadcast.

2.2.1 Hypercubes

Again, the hypercube implementation is based on embedding a spanning tree, this time rooted at the node that is to receive the final result. Assuming that the number of nodes is again $p = 2^d$, the global summation is achieved by performing the following: At the start, all nodes are active. For $k = log(p) - 1, \ldots, 0$, all nodes P_i which are still active, and differ from the root in the kth most significant digit send their current partial result to their neighbor in the kth bit: $P_{\mathrm{ngbr(i,k)}}$, and become inactive for the rest of the communication. All nodes $P_{\mathrm{ngbr(i,k)}}$ receive the partial result from P_i and combine it with the local partial result, to create the new partial result. As with the broadcast, this approach minimizes the number of startup latencies incurred.

The described minimum spanning tree reduction-to-one is given by Algorithm ??. It can be easily seen that this algorithm incurs no networks conflicts on hypercubes and hence its cost on hypercubes is given by

$$T^{\mathrm{cube}} = \log(p)\alpha + \log(p)n\beta + \log(p)n\gamma$$

where γ equals the time to combine two elements.

2.2.2 Linear Arrays

Just as for the broadcast, the above algorithm incurs network conflicts on a power-of-two linear array. The simple solution is to toggle the bits least significant to most significant. This has the effect of combining vectors on neighboring *active* processors at each step.

For non-power-of-two linear arrays, the tree can be embedded like the recursive splitting broadcast, performing the communication and associated computation in the opposite order of the broadcast. We shall call the the recursive splitting reduction-to-one.

2.2.3 Meshes

When vector lengths are short, the mesh can again be viewed as a linear array and a recursive splitting reduction-to-one performed. This automatically avoids network conflicts. We will address this issue in the long vector gather section. **[See note in broadcast section.]**

2.2.4 Code

In this section, we present code for the discussed reduction-to-one algorithms. We will give the code for a specific reduction, the double precision summation operator. Since we already observed that the reduction-to-one is incurs communication much like the broadcast, except in reverse, we will present the code by showing how it can be derived from the broadcast code. The final result is given in Figure 2.6.

The calling sequence is much like the broadcast, except that the root is now the destination of the result, rather than the source.

```
RSsum2one_lin_rec( x, n, left, right, root, comm, work )
double   *x,           /* Input: Address of data to be summed
                          Output: Address of result data on root,
                                  GARBAGE on other node.         */
         *work,        /* Work array of length n                 */
long     n,            /* Length in doubles                      */
         left, right,  /* current left and right index           */
         root;         /* current root                           */
MPI_COMM comm;         /* Communicator.                          */
```

Notice that we need a work array in which to store the received data before it is added to the local partial result. Also, the input vector is overwritten with the final result on the root of the reduction-to-one and with intermediate, unusable, results on the other nodes. The arguments are:

- `x`, upon entry, the address of the buffer where the data to be summed resides. Upon completion, this buffer is overwritten with the result on the root, and with intermediate results on the other participating nodes.
- `n`. the number of doubles to be summed.
- `left, right`, the left and right indices of the partition of the linear array on which the summation-to-one is to proceed,
- `root`, the root of the summation-to-one.

```
RSsum2one_lin_rec( x, n, left, right, root, comm, work )
double   *x,            /* Input: Address of data to be summed
                           Output: Address of result data on root,
                                   GARBAGE on other node.          */
         *work,         /* Work array of length n                  */
int      n,             /* Length in doubles                       */
         left, right,   /* current left and right index            */
         root;          /* current root                            */
MPI_COMM comm;          /* Communicator.                           */
{
  int  me, mid, srce,   /* holds me, mid, srce index               */
       stat(MPI_STATUS_SIZE), ierr;

  if ( left == right ) return;
  MPI_Comm_rank( comm, &me );
                /* determine midpoint.                             */
  half = ( right-left )/2;
  mid =  ( left+half < root ? left+half : right-half);
                /* determine source                                */
  if ( mid <= root ) srce = ( root == mid ? left : mid-1 );
  else srce = ( root == mid-1 ? right : mid );
                /* recursively call sum2one in each partition      */
  RSsum2one_lin_rec( x, n, ( me<mid ? left : mid ),
                         ( me<mid ? mid-1 : right ),
                         ( ( me<mid ) == ( root<mid ) ? root : srce ),
                         comm );
                /* send                                            */
  if ( me==root ) {
    MPI_Recv( work, n, MPI_DOUBLE, root, MPI_ANY_TAG, comm,  stat, &ierr );
    for (i-0; i<n; i++) x[i] += work[i];
  }
  if ( me == srce )
    MPI_Send( x, n, MPI_DOUBLE, dest, 0, comm, stat, &ierr );
}
```

Figure 2.6: Recursive splitting reduction to one for linear arrays, formulated as a recursive algorithm.

- `comm`, the communicator, which holds information about the group and context, as discussed in Section ??.
- `work`, a work array of length `n` doubles, into which messages are received.

The only difference in the local variable declarations is that we now need a variable to indicate the source at each step, rather than the destination:

```
int  mid, srce,       /* holds mid index, srce                */
```

The structure of the rest of the code is quite similar to the broadcast. However, we now do all the recursive calls first and perform the communication and computation *after* returning from the recursive calls. Moreover, for the broadcast, we set up the minimum spanning tree so that all messages not originating at the original root flow towards that original root. Since the communication is now reversed, the minimum spanning tree is now created so that all messages not originating from the original root flow *away* from the root. Since the communication is then performed in reverse, we are again in the situation where they actually move towards the root:

```
              /* determine source                         */
if ( mid <= root ) srce = ( root == mid ? left : mid-1 );
else srce = ( root == mid-1 ? right : mid );
```

Next, we perform the recursive call

```
              /* recursively call sum2one in each partition      */
RSsum2one_lin_rec( x, n, ( me<mid ? left : mid ),
                         ( me<mid ? mid-1 : right ),
                         ( ( me<mid ) == ( root<mid ) ? root : srce ),
                         comm, work );
```

and communicate and sum the data

```
              /* send                                     */
if ( me==root ) {
  MPI_Recv( &work, n, MPI_DOUBLE, root, MPI_ANY_TAG, comm,  stat, &ierr );
  for (i-0; i<n; i++) x[i] += work[i];
}
if ( me == srce )
  MPI_Send( &x, n, MPI_DOUBLE, dest, 0, comm, stat, &ierr );
}
```

As for the destination node in the broadcast, source node in the subpartition that does not contain the root can be chosen differently on . different architectures. For example, for a hypercube, the source is the node that holds the same relative position in the subpartition as the root does in its subpartition:

```
RSsum2one_lin_rec( x, n, root, comm, work )
double   *x,           /* Input: Address of data to be summed
                          Output: Address of result data on root,
                                  GARBAGE on other node.          */
         *work,        /* Work array of length n                  */
int      n,            /* Length in doubles                       */
         root;         /* current root                            */
MPI_COMM comm;         /* Communicator.                           */
{
  int  np;             /* holds number of processors in group     */

  MPI_Comm_size( comm, &np);
  RSsum2one_lin_rec( x, n, 0, np-1, root, comm, work );
}
```

Figure 2.7: Recursive splitting sum2one for linear arrays, which calls the recursive algorithm.

```
  if ( mid <= root ) srce = left + root - mid;
  else srce = mid + root - left;
```

This choice will work on a power-of-two linear array as well. On a general linear array, the destination can be any of the nodes in the subpartition that does not contain the root. For meshes, if the vector length is short, this strategy will work as well. However, if network conflicts are to be avoided, we need to have all messages that don't originate from the original root of the broadcast flow towards the original root. This is accomplished by chosing the source as already indicated in the sample code.

To use this recursive subroutine, an application would call the subroutine with `left` and `right` equal to `0` and `np-1`, respectively, where `np` equals the number of nodes in the group in which the sum-to-one broadcast is to be accomplished. s for the broadcast, this group is assumed to physically form a hypercube, linear array, or mesh, with the required ordering of nodes, as specified in ?. A wrapper is provided in Figure 2.7.

2.3 Scatter

The discussion on the MST broadcast sets the stage for the discription of scatter operation. Recall that the scatter is defined by: Let x be a vector of length n bytes. Partition

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}$$

where x_j is a subvector of length n_j bytes.

- **Before:** For some $k \in 0, \ldots, p-1$, node $\mathbf{P}_k$ owns vector x.

- **After:** Each node $\mathbf{P}_j$, $0 \le j < p$ owns subvector x_j.

In this section, we will make the following simplification: $n_j \approx n/p$ *for all* j. Indeed, for the sake of the analysis, we will take $n_j = n/p$. We will make comments about the case where the subvectors vary drastically in length in Chapter ??.

2.3.1 Hypercubes

For the scatter, it is important to toggle bits from most significant to least significant, even on hypercubes. The reason for this is that it enforces the condition that at each stage only contiguous data needs to be sent. As a result, it is easiest to explain the scatter for both the linear array and hypercube simultaneously, as we will do in the next subsection.

2.3.2 Linear Arrays

The scatter proceeds like the minimum spanning tree broadcast or recursive splitting broadcast on linear arrays, except that at each stage, only the portion of the vector that eventually must reside on nodes within other partition is sent. On the hypercube, at each stage the new root in the other partition is chosen to have a binary representation equal to the current root, except in the appropriate bit. This is illustrated in Figure ??.

Under the assumption that $n_j = n/p$, this operation requires time

$$\sum_{k=0}^{\log_2(p)-1} \alpha + 2^k \frac{n}{p}\beta = \log_2(p)\alpha + \frac{p-1}{p} n\beta$$

Work out more exact formula for non power of two.

2.3.3 Code

The scatter operation proceeds much like the broadcast, except that at each stage the amount of data being sent is reduced.

The calling sequence difference from the broadcast by the fact that an array that indicates what data each node is to receive must be indicated.

```
RSscat_lin_rec( x, ix, left, right, root, comm )
char     *x;            /* Address of data to be sent              */
long     ix,            /* addresses of where subvectors start     */
         left, right,   /* current left and right index            */
         root;          /* current root                            */
MPI_COMM comm;          /* Communicator.                           */
```

Notice that `n` is no longer a parameter. It has been replaced by `ix`. On all nodes but the root, `x` becomes a work array, with the subvector of data intended for the node being returned in vector `x`, starting at address 0.

- `x`,

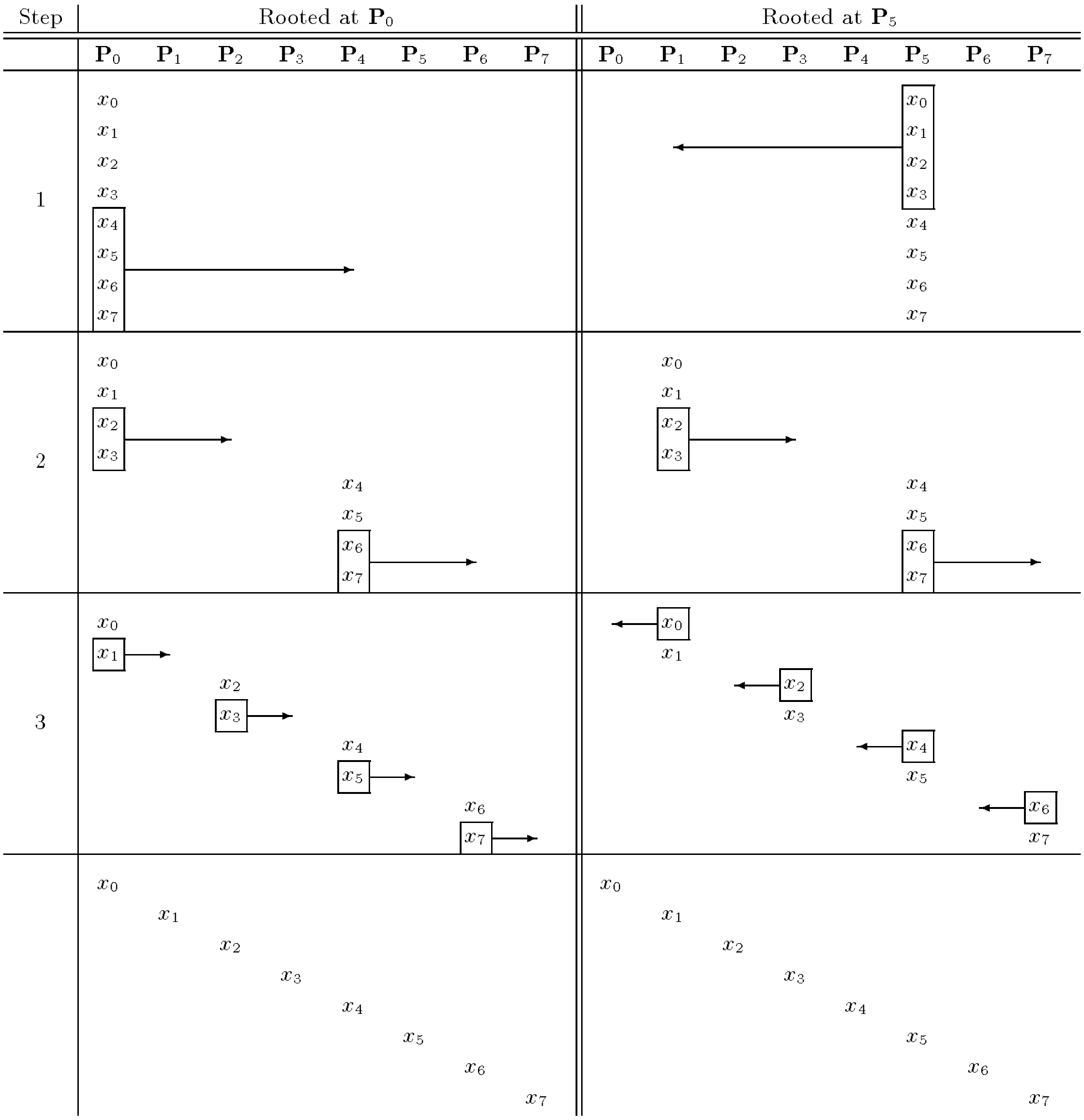


Figure 2.8: Scatter on a hypercube and power-of-two linear array

```
RSscat_lin_rec( x, ix, left, right, root, comm )
char     *x;            /* Address of data to be sent.  Upon return,
                           this buffer contains this node's subvector
                           of data, starting at x[0]              */
int      *ix,           /* addresses of where subvectors start     */
         left, right,   /* current left and right index            */
         root;          /* current root                            */
MPI_COMM comm;          /* Communicator.                           */
{
  int  me, mid, dest,   /* holds me, mid, dest index               */
       stat(MPI_STATUS_SIZE), ierr;

  if ( left == right )
                /* copy data to align with x[0]                    */
    for (i=0; i<(ix[me+1]-ix[me]); i++) x[i] = x[ix[me]+i];

  MPI_Comm_rank( comm, &me );
                /* determine midpoint.                             */
  half = ( right-left )/2;
  mid =  ( left+half < root ? left+half : right-half);
                /* determine dest                                  */
  if ( mid <= root ) dest = ( root == right ? mid-1 : left );
  else dest = ( root == left ? mid : right );
                /* send                                            */
  if ( me==root )
    MPI_Send( &x[ix[ mid<=root? left:mid ]],
              (mid<=root? ix[right+1]-ix[mid] : ix[mid]-ix[left]),
              MPI_BYTE, dest, 0, comm, stat, &ierr );
  if ( me == dest )
    MPI_Recv( &x[ix[ mid<=root? left:mid ]],
              (mid<=root? ix[right+1]-ix[mid] : ix[mid]-ix[left]),
              MPI_BYTE, root, MPI_ANY_TAG, comm,  stat, &ierr );
                /* recursively call scatter in each partition      */
  RSscat_lin_rec( x, ix, ( me<mid ? mid-1 : right ),
                         ( ( me<mid ) == ( root<mid ) ? root : dest ),
                         comm );
}
```

Figure 2.9: Recursive splitting scatter for linear arrays, formulated as a recursive algorithm.

```
RSscat( x, ix, root, comm )
char     *x;            /* Address of data to be sent.  Upon return,
                           this buffer contains this node's subvector
                           of data, starting at x[0]                */
int      *ix,           /* addresses of where subvectors start      */
         root;          /* current root                             */
MPI_COMM comm;          /* Communicator.                            */
{
  int  np;              /* holds number of processors in group      */

  MPI_Comm_size( comm, &np);
  RSscat_lin_rec( x, ix, 0, np-1, root, comm );
}
```

Figure 2.10: Recursive splitting scatter for linear arrays, which calls the recursive algorithm.

- `ix`, array of indices into array `x` that indicate what subarray each node is to receive. Specifically, the subvector consisting of elements `x[i]` through `x[i+1]` will be returned in vector `y` on the `i`th node of the group.

Notice that array `ix` has `np+1` entries, where `np` equals the number of nodes in the group. Entry `ix[np]` must equal `n`, the length of the vector being scattered.

A wrapper that calls the recursive routine is given in Figure 2.10.

2.4 Gather

The discussions on the broadcast, reduction-to-one, and scatter set the stage for the discription of scatter operation. Recall that the scatter is defined by: Let x be a vector of length n bytes. Partition

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}$$

where x_j is a subvector of length n_j bytes.

- **Before:** Each node $\mathbf{P}_j$, $0 \leq j < p$ owns subvector x_j.
- **After:** For some $k \in 0, \ldots, p-1$, node $\mathbf{P}_k$ owns vector x.

As for the gather, we will make the following simplification: $n_j \approx n/p$ for all j. Indeed, for the sake of the analysis, we will take $n_j = n/p$. We will make comments about the case where the subvectors vary drastically in length in Chapter ??.

2.4.1 Hypercubes

The gather is equivalent to the scatter, executed in reverse.

2.4.2 Linear Arrays

The gather is equivalent to the scatter, executed in reverse.

2.4.3 Meshes

The gather is equivalent to the scatter, executed in reverse.

2.4.4 Code

The code for the gather algorithm is given in Figure ??. This code is derived from the scatter code much like the summation-to-one code was derived from the broadcast code.

A wrapper that calls the recursive routine is given in Figure 2.12.

2.5 Collect

This sections presents a few basic implementations of the collect operation.

The collect operation can be viewed as a simultaneous broadcast by all of the nodes. (It should not, however, be implemented this way!) It is defined by: Let x be a vector of length n bytes. Partition

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}$$

where x_j is a subvector of length n_j bytes.

- Before: Each node $\mathbf{P}_j$, $0 \leq j < p$, owns x_j.
- After: Each node $\mathbf{P}_j$, $0 \leq j < p$, owns x.

In other words, each node contributes a subvector and is left with all of the subvectors, concatenated in node order.

As with the description of the scatter, we will make the assumption that $n_j = n/p$.

2.5.1 Hypercubes

The hypercube collect algorithm can be described as a recursive doubling algorithm. During the first step, pairs of nodes that differ in the least significant bit exchange subvectors, concatenating the received subvector to the original subvector. This process is repeated for each hypercube dimension, toggling bits from least significant to most significant. It can be easily seen that after $\log_2(p)$ steps, all nodes own the entire vector, as is illustrated for a 3-cube in Figure ??. Notice that the ordering of toggling is important: it preserves contiguity of the data to be communicated.

Under the assumption that $n_j = n/p$, this operation requires time

$$T_{\text{collect}}^{\text{cube}} = \sum_{k=0}^{\log_2(p)-1} \alpha + 2^k \frac{n}{p}\beta = \log_2(p)\alpha + \frac{p-1}{p}n\beta$$

```
RSgath_lin_rec( x, ix, left, right, root, comm )
char     *x;           /* Address of data to be sent             */
long     ix,           /* addresses of where subvectors start     */
         left, right,  /* current left and right index            */
         root;         /* current root                            */
MPI_COMM comm;         /* Communicator.                           */
{
  int  mid,            /* holds mid index                         */
       np, me, npdim,  /* nprocs. in group, my rank, hyp. dim.    */
       stat(MPI_STATUS_SIZE), ierr;

  if ( left == right ) return;
  size_rank_dim( comm, &np, &me);
              /* determine midpoint.                              */
  half = ( right-left )/2;
  mid =  ( left+half < root ? left+half : right-half);
              /* determine dest                                   */
  if ( mid <= root ) dest = ( root == mid ? left : mid-1 );
  else dest = ( root == mid-1 ? right : mid );
              /* recursively call gather in each partition        */
  RSgath_lin_rec( x, ix, ( me<mid ? mid-1 : right ),
                        ( ( me<mid ) == ( root<mid ) ? root : dest ),
                        comm );
              /* send                                             */
  if ( me==root )
    MPI_Recv( &x[ix[ mid<=root? left:mid ]],
              (mid<=root? ix[right+1]-ix[mid] : ix[mid]-ix[left]),
              MPI_BYTE, dest, MPI_ANY_TAG, comm,  stat, &ierr );
  if ( me == dest )
    MPI_Send( &x[ix[ mid<=root? left:mid ]],
              (mid<=root? ix[right+1]-ix[mid] : ix[mid]-ix[left]),
              MPI_BYTE, root, 0, comm, stat, &ierr );
}
```

Figure 2.11: Recursive splitting gather for linear arrays, formulated as a recursive algorithm.

```
RSgath( x, ix, root, comm )
char     *x;            /* Address of data to be sent              */
long     ix,            /* addresses of where subvectors start     */
         root;          /* current root                            */
MPI_COMM comm;          /* Communicator.                           */
{
  int  np;              /* holds number of processors in group     */

  MPI_Comm_size( comm, &np);
  RSgath_lin_rec( x, ix, 0, np-1, root, comm );
}
```

Figure 2.12: Recursive splitting gather for linear arrays, which calls the recursive algorithm.

2.5.2 Linear Arrays

On a linear array of 2^d processors, the same strategy can be used except that now contention will occur since multiple messages will traverse the same links. For short vectors, this doesn't pose a problem, since the goal is to minimize the number of messages. For longer vectors, the cost of link conflicts may be significant, however. Indeed, assuming no excess bandwidth in the network, the cost of the collect becomes

$$T^{\mathrm{1D}}_{\mathrm{RDcollect}} = \sum_{k=0}^{\log_2(p)-1} \alpha + 2^{2k}\frac{n}{p}\beta = \log_2(p)\alpha + \frac{(p-1)^2}{p}n\beta$$

We defer how to extend the hypercube collect to non-power-of-two linear arrays until after the chapter on long vector algorithms, for reasons that will become clear later. Instead, we propose a simpler scheme: Gather the vector to a single node, followed by a broadcast from that node. We have seen this can be accomplished without incurring network conflicts, at a cost of

$$2\lceil\log(p)\rceil\alpha + \frac{p-1}{p}n\beta + \lceil\log(p)\rceil n\beta$$

We shall call this the gather/broadcast collect.

2.5.3 Meshes

For power-of-two meshes, we can again employ the hypercube collect. This would incur some network conflicts, making an exact computation of the cost somewhat more complicated:

$$T^{\mathrm{2D}}_{\mathrm{RDcollect}} = \sum_{k=0}^{\log_2(c)-1} \alpha + 2^{2k}\frac{n}{rc}\beta + \sum_{k=0}^{\log_2(r)-1} \alpha + 2^{2k}\frac{n}{r}\beta = \log_2(p)\alpha + \frac{(c-1)^2 + c(r-1)^2}{p}n\beta$$

For non-power-of-two meshes, we can also employ the gather/broadcast collect, treating the nodes logically as a linear array. If recursive splitting gather and broadcast are used which avoid network conflicts, the cost becomes

$$2\lceil\log(p)\rceil\alpha + \frac{p-1}{p}n\beta + \lceil\log(p)\rceil n\beta$$

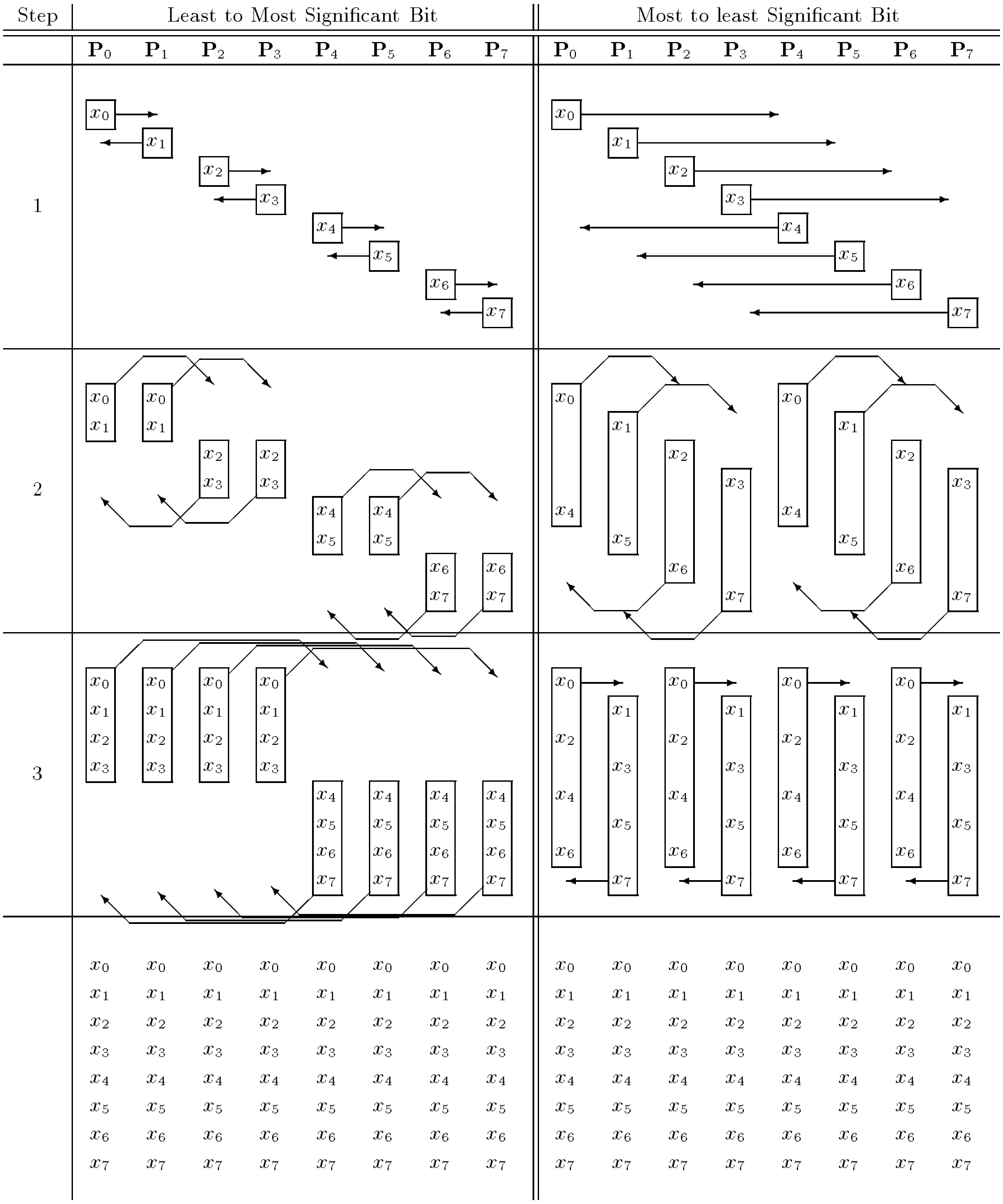


Figure 2.13: Collect on a hypercube and power-of-two linear array. *left:* Notice that contiguity of data is maintained, and on the linear array, network conflicts do occur, and become progressively worse as the vector length inceases. *right:* The worst network conflicts occur when the vectors are shortest, at the expense of having to send noncontiguous data.

```
RScoll( x, ix, comm )
char     *x;            /* Address of data to be collectd          */
long     *ix;           /* addresses of where subvectors start     */
MPI_COMM comm;          /* Communicator.                           */
{
  int  np;              /* holds number of processors in group     */

  MPI_Comm_size( comm, &np);
  RSgath_lin_rec( x, ix, 0, np-1, root, comm );
  RSbcast_lin_rec( x, 0, np-1, root, comm );
}
```

Figure 2.14: Recursive splitting gather/broadcast collect for linear arrays.

As with the non-power-of-two linear array collect, we defer a generalization of the hypercube collect for non-power-of-two meshes until later.

2.5.4 Code

We will restrict ourselves to giving the code for an implementation that has the following properties:

- it works on hypercubes, non-power-of-two linear arrays and meshes,
- it incurs reasonable latency (within a factor of two)
- it is guaranteed not to incur network conflicts

A simple algorithm that meets these criteria is give by a recursive splitting gather, followed by a broadcast. An implementation is given in Figure 2.14

2.6 Distributed Reduction

Recall that given an associative and commutative binary operator $\oplus$, and let x^j, $j = 0, \ldots, p-1$, be vectors with elements of length n so that and let

$$\oplus_{j=0}^{p-1} x^j = x^0 \oplus x^1 \oplus \cdots \oplus x^{p-1}$$

represent an *element-wise* application of this operation to the vectors. Partition each vector

$$x^j = \begin{pmatrix} x_0^j \\ x_1^j \\ \vdots \\ x_{p-1}^j \end{pmatrix}$$

where x_k^j are of length n_k. Then the distributed reduction operation is defined by

- **Before:** Each node $\mathbf{P}_j$, $0 \leq j < p$ owns vector x^j.

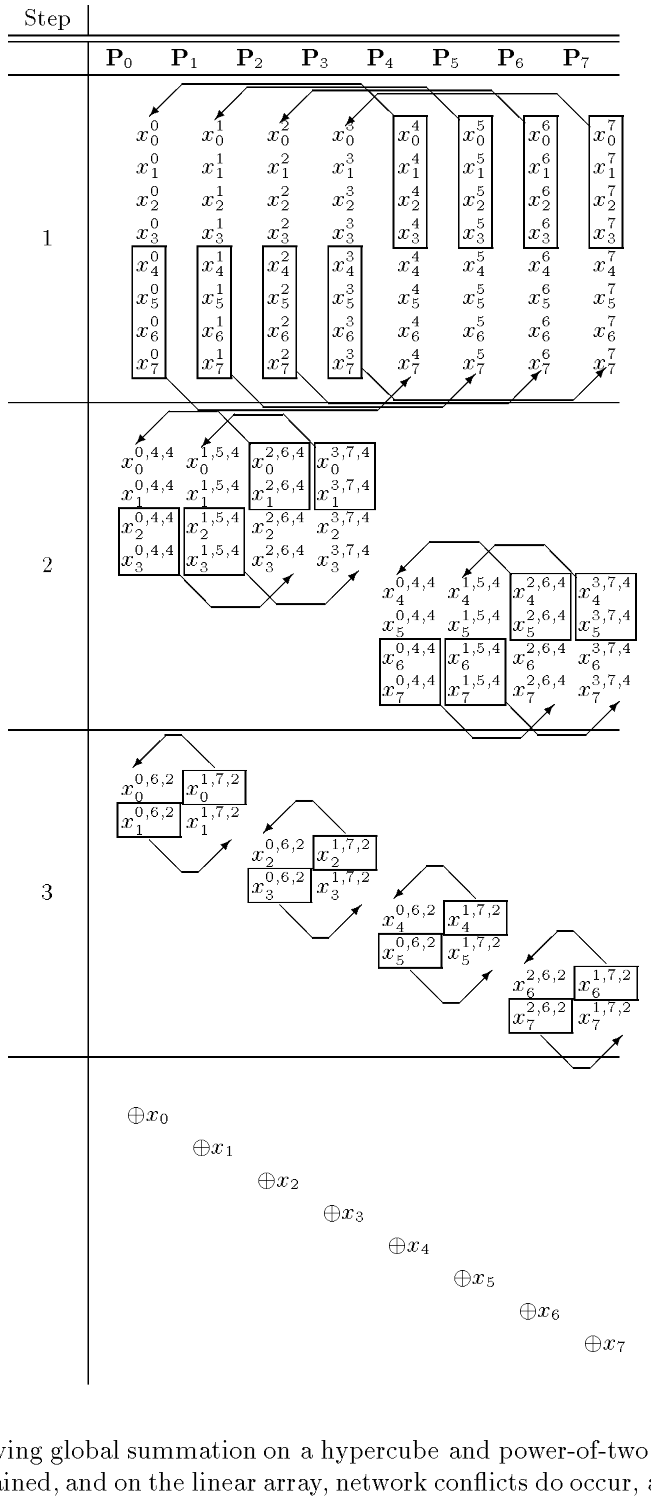


Figure 2.15: Recursive halving global summation on a hypercube and power-of-two linear array. Notice that contiguity of data is maintained, and on the linear array, network conflicts do occur, and become progressively worse as the vector length inceases.

- **After:** Each node $\mathbf{P}_j$, $0 \leq j < p$ owns subvector x_j, where

$$x_j = \oplus_{i=0}^{p-1} x_j^i$$

Similar to the fact that the broadcast and reduction-to-one are dual operation, so are the collect and distributed reduction: The communication required for the distributed reduction is essentially the collect in reverse, with a reduction operation performed between corresponding elements of local and arriving data.

2.6.1 Hypercube

While the hypercube collect operation uses a recursive doubling algorithm, the distributed reduction uses a recursive halving algorithm. During $\log(p)$ stages, nodes are paired with neighbors that differ in the kth bit, where the bits are toggled from most significant to least significant. At each stage, the node sends either the first half or second half of the remaining vector, and performs computation according to the given combine operator between the remaining half and the data sent by its neighbor. Whether to send the first or second half depends on whether the node has a binary 0 or 1 for the current bit being toggled. This method is illustrated in Figure ??. Its cost is

$$T^{\text{cube}}_{\text{red-to-one}} = \sum_{k=\log(p)-1}^{0} \alpha + \frac{n}{2^{\log(p)-k}}(\beta+\gamma) = \log(p)\alpha + \frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma$$

2.6.2 Linear Arrays

The above method will suffer from contention if directly implemented on a power-of-two linear array, as with the collect. Indeed, on a linear array with no excess bandwidth, the cost becomes

$$T^{\text{1D}}_{\text{RHred-to-one}} = \sum_{k=\log(p)-1}^{0} \alpha + \frac{n}{2^{\log(p)-k}}(2^k\beta+\gamma) = \log(p)\alpha + \frac{(p-1)^2}{p}n\beta + \frac{p-1}{p}n\gamma$$

For non-power-of-two linear arrays, we can adopt a method similar to the gather/broadcast collect: First perform a recursive splitting reduction-to-one, followed by a scatter. Neither of these incurs network conflicts, yielding a cost of

$$T^{\text{1D}}_{???} = 2\log(p)\alpha + (\log(p) + \frac{p-1}{p})n\beta + \log(p)n\gamma$$

2.6.3 Meshes

For power-of-two size meshes, the hypercube algorithm can be used, resulting in a bit less link contention than on a linear array:

$$T^{\text{2D}}_{\text{RHred-to-one}} = \sum_{k=0}^{\log_2(c)-1} \alpha + 2^k\frac{n}{rc}(2^k\beta+\gamma) + \sum_{k=0}^{\log_2(r)-1} \alpha + 2^k\frac{n}{r}(2^k\beta+\gamma) = \log_2(p)\alpha + \frac{(c-1)^2 + c(r-1)^2}{p}n\beta + \frac{p-1}{p}n\gamma$$

For general, non-power-of-two meshes, we can use the same recursive splitting reduction-to-one/scatter technique used on linear arrays. This results in a time of:

$$2\lceil\log(p)\rceil\alpha + (\frac{p-1}{p} + \lceil\log(p)\rceil)n\beta + (\frac{p-1}{p} + \lceil\log(p)\rceil)n\gamma$$

```
RSdistsum( x, ix, comm, work )
double   *x,           /* Address of data to be summed           */
         *work,        /* Work array of length n                 */
long     *ix;          /* addresses of where subvectors start    */
MPI_COMM comm;         /* Communicator.                          */
{
  int  np;             /* holds number of processors in group    */

  MPI_Comm_size( comm, &np);
  RSsum2one_lin_rec( x, 0, np-1, root, comm, work );
  for (i=0; i<=np; i++) ix[i] *= sizeof(double);
  RSscat_lin_rec( x, ix, 0, np-1, root, comm );
  for (i=0; i<=np; i++) ix[i] /= sizeof(double);
}
```

Figure 2.16: Recursive splitting sum-to-one/scatter distributed summation for linear arrays.

2.6.4 Code

We will restrict ourselves to giving the code for an implementation that has the following properties:

- it works on hypercubes, non-power-of-two linear arrays, and meshes,
- incurs reasonable latency (within a factor of two), and
- is guaranteed not to incur network conflicts.

A simple algorithm that meets these criteria is give by a recursive splitting reduction-to-one, followed by a scatter. An implementation for summation is given in Figure ??

2.7 Reduction-to-All

Recall that the reduction-to-all is similar to the distributed reduction, except that the result is left duplicated on all nodes:

- **Before:** Each node $\mathbf{P}_j$, $0 \leq j < p$ owns vector x^j.
- **After:** All nodes node $\mathbf{P}_j$, $0 \leq j < p$ own vector x, where

$$x = \oplus_{j=0}^{p-1} x^j$$

2.7.1 Hypercubes

We will present three solutions to performing the reduction-to-all operation on hypercubes.

Fan-in, Fan-out

The fan-in, fan-out reduction-to-all is perhaps better described as a reduction-to-one followed by a broadcast. Using the hypercube implementations give before, its cost becomes

$$2\log(p)\alpha + 2\log(p)n\beta + \log(p)n\gamma$$

Notice that the fan-in, fan-out reduction-to-all underutilizes nodes: During the fan-in, at each stage the number of active nodes is halved, while during the fan-out at each stage the number of active nodes doubles.

Bidirectional Exchange

The bidirectional exchange reduction uses all nodes as follows: During $\log(p)$ stages, nodes pair up by toggling the bits is some order, without repeat. During each stage, the full vector that holds the current partial result is exchanged with the neighbor, and the arriving vector is combined with the current partial result to form the new partial result. It is clear that after the $\log(p)$ stages, all processors posses the result. The cost of this approach is

$$\log(p)\alpha + \log(p)n\beta + \log(p)n\gamma$$

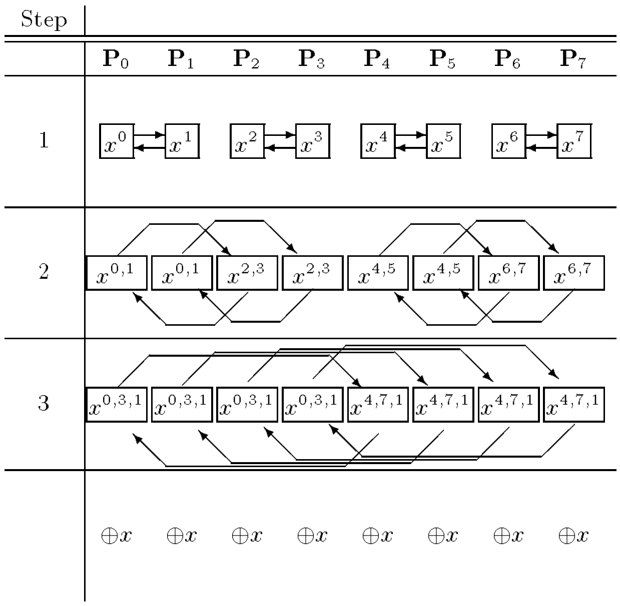


Figure 2.17: Bidirectional exchange reduction-to-all on a hypercube or power-of-two linear array of nodes. Notice that on a linear array, network conflicts get progressively worse. Changing the order in which the bits are toggled does not help, since at each stage full vectors are communicated.

2.7.2 Linear Arrays

For power-of-two linear arrays, both of the hypercube reduction-to-all schemes apply. However, network conflicts affect the bidirectional exchange increasing its cost to

$$T^{\rm 1D}_{\rm red-to-all} = \sum_{k=0}^{\log(p)-1} \alpha + 2^k n\beta + n\gamma = \log(p)\alpha + (p-1)n\beta + \log(p)n\gamma$$

In addition, on architecutures we have encountered, the startup cost α when messages are exchanged between two nodes is typically twice the startup cost when only one message is passed between two nodes. In addition, the fan-in, fan-out algorithm generalizes to non-power-of-two linear arrays. As a result, we advocate fan-in, fan-out on linear arrays. This has a cost of

$$\lceil \log(p) \rceil \alpha + \lceil \log(p) \rceil n\beta + \lceil \log(p) \rceil n\gamma$$

2.7.3 Meshes

For the same reasons as given for reduction-to-all on linear arrays, we advocate viewing the mesh as a logical linear array, and performing a fan-in, fan-out algorithm. The cost is thus the same as for linear arrays.

2.7.4 Code

As for the collect and the distributed reduction, we will restrict ourselves to giving the code for an implementation that has the following properties:

- it works on hypercubes, non-power-of-two linear arrays, and meshes,
- incurs reasonable latency (within a factor of two), and
- is guaranteed not to incur network conflicts.

A simple algorithm that meets these criteria is give by a recursive splitting reduction-to-one, followed by a broadcast. An implementation for summation is given in Figure 2.18

2.8 A Building Block Approach

Ignoring the special algorithms required for the hypercube algorithms, we can summarize what we learned in this chapter by proposing the following approach to short vector algorithms:

For the short vector cast, the communication operations can be split into two categories: building blocks and derived algorithms. In particular, the building blocks for short vectors are the broadcast, reduction-to-one, scatter, and gather operations. The derived algorithms are the collect, distributed reduction, and reduction-to-all operations. How to derive algorithms from building blocks is indicated in Figure ??

We should note that the implementations of the building blocks can be arranges so that no network conflicts are incurred on hypercubes, linear arrays, and higher dimensional meshes. As a result, the derived algorithms do not incur network conflicts either.

2.9 Summary

In this chapter, we have introduced the basic concepts which will be crucial throughout the remainder of the book: The recursive-splitting algorithm provides an effective method for getting data into or out of a single node in as few steps as possible.

```
RSsum2all( x, comm, work )
double   *x,            /* Address of data to be summed            */
         *work,         /* Work array of length n                  */
MPI_COMM comm;          /* Communicator.                           */
{
  int  np;              /* holds number of processors in group     */

  MPI_Comm_size( comm, &np);
  RSsum2one_lin_rec( x, 0, np-1, root, comm, work );
  for (i=0; i<=np; i++) ix[i] *= sizeof(double);
  RSbcast_lin_rec( x, ix, 0, np-1, root, comm );
  for (i=0; i<=np; i++) ix[i] /= sizeof(double);
}
```

Figure 2.18: Recursive splitting sum-to-one/broadcast sum-to-all for linear arrays.

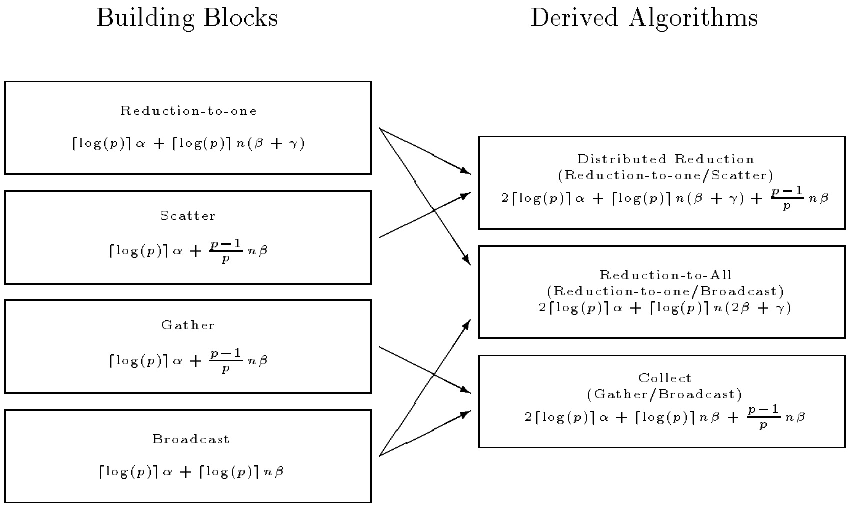


Figure 2.19: A building block approach to short vector algorithms