COLLECTIVE COMMUNICATION

4.1 Introduction and Overview

Collective communication is defined as communication that involves a group of processes. The functions of this type provided by MPI are the following:

- Barrier synchronization across all group members (Section 4.3).
- Broadcast from one member to all members of a group (Section 4.4). This is shown in figure 4.1.
- Gather data from all group members to one member (Section 4.5). This is shown in figure 4.1.
- Scatter data from one member to all members of a group (Section 4.6). This is shown in figure 4.1.
- A variation on Gather where all members of the group receive the result (Section 4.7). This is shown as "allgather" in figure 4.1.
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 4.8). This is shown as "alltoall" in figure 4.1.
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member (Section 4.9).
- A combined reduction and scatter operation (Section 4.10).
- Scan across all members of a group (also called prefix) (Section 4.11).

A collective operation is executed by having all processes in the group call the communication routine, with matching arguments. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 3. One of the key arguments is a communicator that defines the group of participating processes and provides a context for the operation. Several collective routines such as broadcast and gather have a single originating or receiving process. Such processes are called the *root*. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all par-

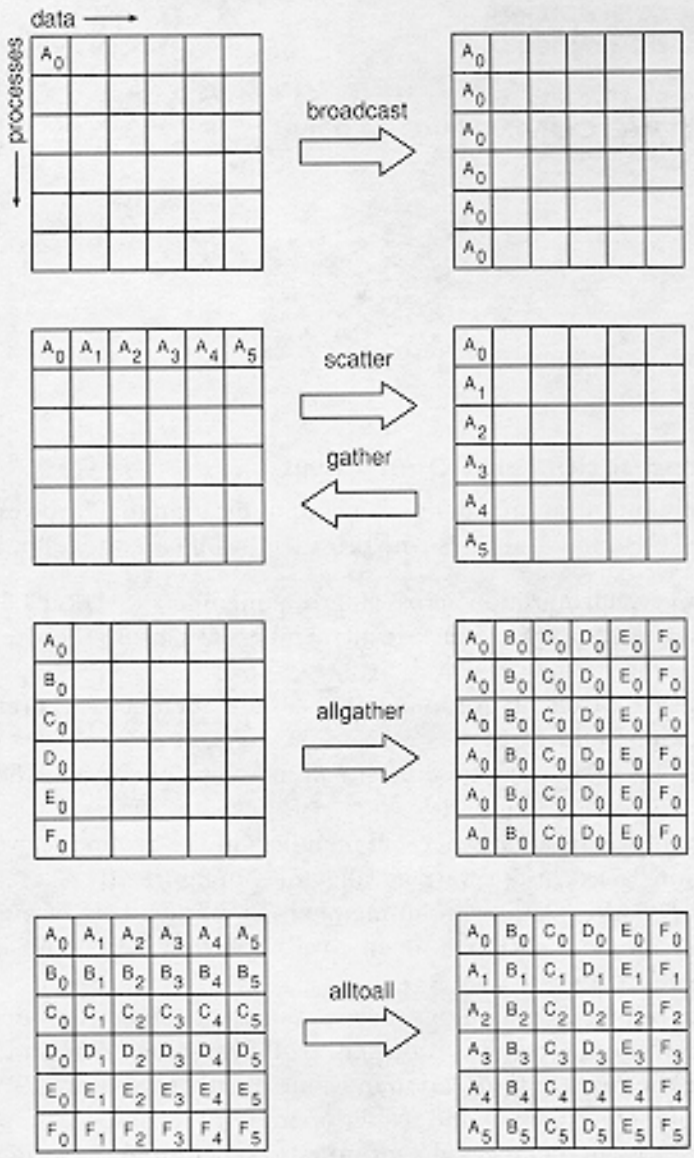

Fig. 4.1 Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

ticipants except the root. The reader is referred to Chapter 3 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 5 for information on how to define groups and create communicators.

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Distinct type maps (the layout in memory, see Section 3.12) between sender and receiver are still allowed.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. A more detailed discussion of correct use of collective routines is found in Section 4.12.

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

The collective operations do not accept a message tag argument. If future revisions of MPI define non-blocking collective functions, then tags (or a similar mechanism) will need to be added so as to allow the disambiguation of multiple, pending, collective operations. (*End of rationale.*)

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 4.12. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator must be created for the collective operation so as to avoid interference with any

on-going point-to-point communication at the time of the collective call. This is discussed further in Section 4.12. (*End of advice to implementors.*)

4.2 Communicator Argument

The key concept of the collective functions is to have a "group" of participating processes. The routines do not have a group identifier as an explicit argument. Instead, there is a communicator argument. For the purposes of this chapter, a communicator can be thought of as a group identifier linked with a context. An inter-communicator, that is, a communicator that spans two groups, is *not* allowed as an argument to a collective function.

4.3 Barrier Synchronization

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier(MPI_Comm comm )

MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

4.4 Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm )

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of group

using the same arguments for comm, root. On return, the contents of root's communication buffer has been copied to all processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

4.4.1 EXAMPLE USING MPI_BCAST

Example 4.1 Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as comm in the above) have been assigned appropriate values.

4.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (integer, significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
           COMM, IERROR)
   <type> SENDBUF(*), RECVBUF(*)
   INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is *as if* each of the `n` processes in the group (including the root process) had executed a call to

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```

and the root had executed `n` calls to

```
MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount,
recvtype, i, ...),
```

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

An alternative description is that the `n` messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount·n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on process `i` must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the recvcount argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry `i` specifies the displacement relative to recvbuf at which to place the incoming data from process `i` (significant only at root)

IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
            RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs.

The outcome is *as if* each process, including the root process, sends a message to the root,

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```

and the root executes n receives,

```
MPI_Recv(recvbuf + disp[i] · extent(recvtype), recvcounts[i],
recvtype, i, ...).
```

Messages are placed in the receive buffer of the root process in rank order, that is, the data sent from process j is placed in the jth portion of the receive buffer recvbuf on process root. The jth portion of recvbuf begins at offset displs[j] elements (in terms of recvtype) into recvbuf.

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 4.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

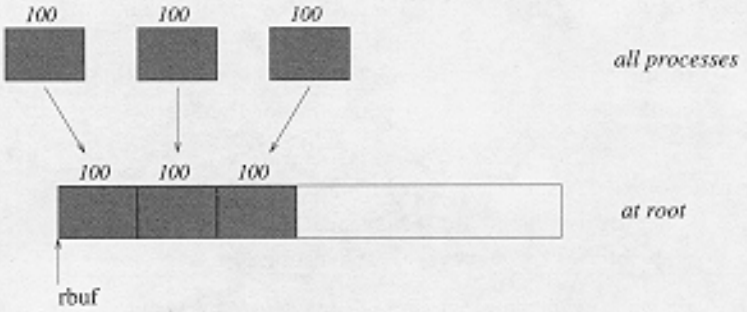

Fig. 4.2 The root process gathers 100 ints from each process in the group.

4.5.1 EXAMPLES USING MPI_GATHER, MPI_GATHERV

Example 4.2 Gather 100 ints from every process in group to root. See figure 4.2.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 4.3 Previous example modified—only the root allocates memory for the receive buffer.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
   MPI_Comm_size( comm, &gsize);
   rbuf = (int *)malloc(gsize*100*sizeof(int));
   }
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 4.4 Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of gsize*100 ints since type matching is defined pairwise between the root and each process in the gather.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
```

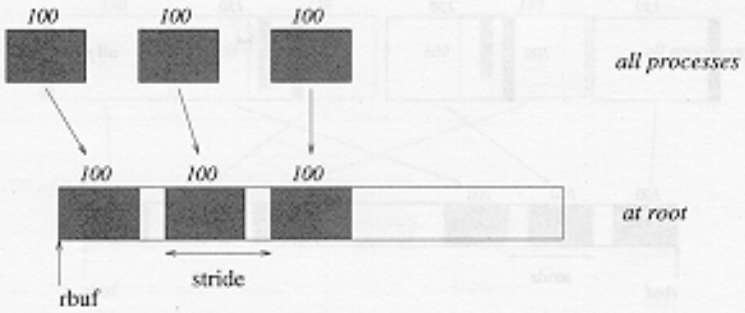

Fig. 4.3 The root process gathers 100 ints from each process in the group, and each set is placed stride ints apart.

```
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 4.5 Now have each process send 100 ints to root, but place each set (of 100) *stride* ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume *stride* $\geq$ 100. See figure 4.3.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

Note that the program is erroneous if *stride* < 100.

Example 4.6 Same as Example 4.5 on the receiving side, but send the 100 ints from the 0th column of a 100×150 int array, in C. See figure 4.4.

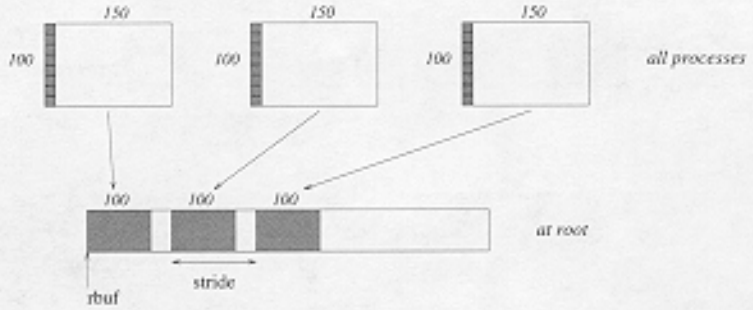

Fig. 4.4 The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

```
MPI_Comm comm;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array
 */
MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                        root, comm);
```

Example 4.7 Process i sends (100 − i) ints from the ith column of a 100 × 150 int array, in C. It is received into a buffer with stride, as in the previous two examples. See figure 4.5.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;
```

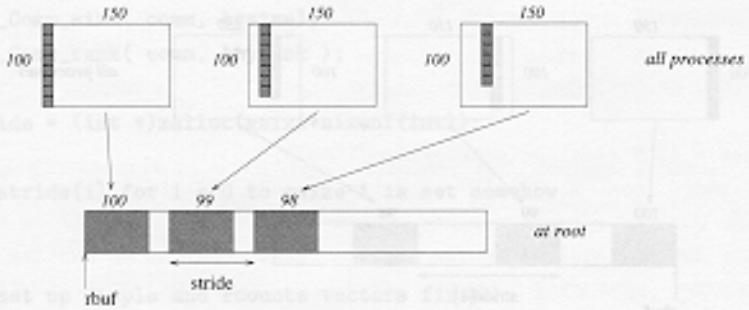

Fig. 4.5 The root process gathers 100−i ints from column i of a 100×150 C array, and each set is placed stride ints apart.

```
    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;     /* note change from previous example */
    }
    /* Create datatype for the column we are sending
     */
    MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    /* sptr is the address of start of "myrank" column
     */
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

Note that a different amount of data is received from each process.

Example 4.8 Same as Example 4.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that that we read a column of a C array. A similar thing was done in Example 3.32, Section 3.12.7.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank, disp[2], blocklen[2];
    MPI_Datatype stype,type[2];
    int *displs,i,*rcounts;
```

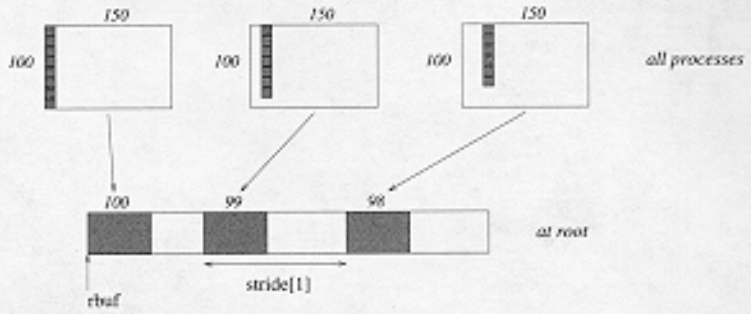

Fig. 4.6 The root process gathers 100−i ints from column i of a 100×150 C array, and each set is placed `stride[i]` ints apart (a varying stride).

```
...

MPI_Conn_size( comm, &gsize);
MPI_Conn_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;
}
/* Create datatype for one int, with extent of entire row
 */
disp[0] = 0;       disp[1] = 150*sizeof(int);
type[0] = MPI_INT; type[1] = MPI_UB;
blocklen[0] = 1;   blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_connit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                        root, comm);
```

Example 4.9 Same as Example 4.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See figure 4.6.

```
MPI_Conn comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;

...
```

```
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 */

/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

Example 4.10 Process i sends num ints from the ith column of a 100 × 150 int array, in C. The complicating factor is that the various values of num are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,i,*rcounts,num;
...

MPI_Comm_size( comm, &gsize);
```

```
MPI_Conn_rank( comm, &myrank );

/* First, gather nums to root
 */
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
/* root now has correct rcounts, using these we set displs[] so
 * that data is placed contiguously (or concatenated) at receive end
 */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i) {
    displs[i] = displs[i-1]+rcounts[i-1];
}
/* And, create receive buffer
 */
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
 *sizeof(int));
/* Create datatype for one int, with extent of entire row
 */
disp[0] = 0;       disp[1] = 150*sizeof(int);
type[0] = MPI_INT; type[1] = MPI_UB;
blocklen[0] = 1;   blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

4.6 Scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)

IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

The outcome is *as if* the root executed n send operations,

```
MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount,
sendtype, i, ...),
```

and each process executed a receive,

```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...).
```

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount·n, sendtype, ...). This message is split into n equal segments, the *i*th segment is sent to the *i*th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                 RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing the new argument, displs.

The outcome is as if the root executed n send operations,

```
MPI_Send(sendbuf + displs[i] · extent(sendtype), sendcounts[i],
sendtype, i, ...),
```

and each process executed a receive,

```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...).
```

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

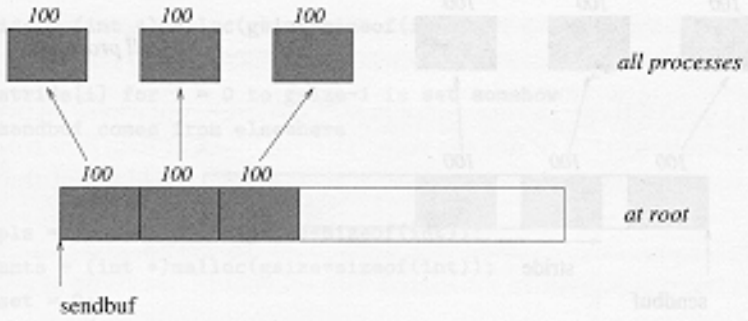

Fig. 4.7 The root process scatters sets of 100 ints to each process in the group.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

4.6.1 EXAMPLES USING MPI_SCATTER, MPI_SCATTERV

Example 4.11 The reverse of Example 4.2. Scatter sets of 100 ints from the root to each process in the group. See figure 4.7.

```
MPI_Conn comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Conn_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 4.12 The reverse of Example 4.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride* ints apart in the sending buffer. Requires use of MPI_SCATTERV. Assume *stride* $\geq$ 100. See figure 4.8.

```
MPI_Conn comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
```

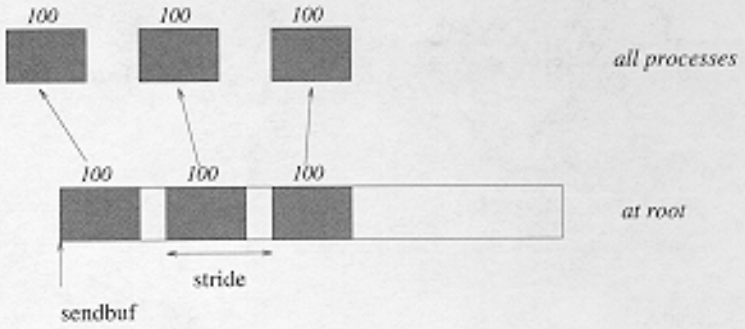

Fig. 4.8 The root process scatters sets of 100 ints, moving by `stride` ints from send to send in the scatter.

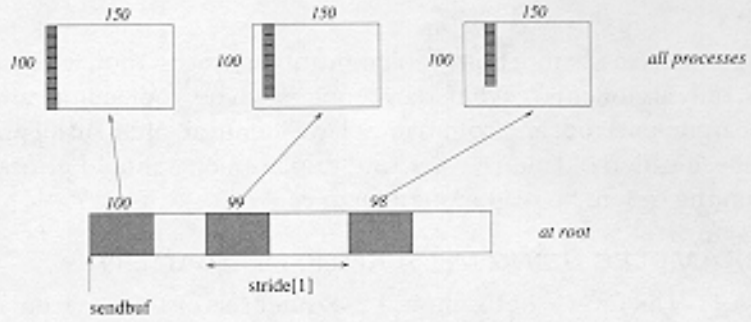

Fig. 4.9 The root scatters blocks of 100-i ints into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` ints apart.

```
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                                                        root, comm);
```

Example 4.13 The reverse of Example 4.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the ith column of a 100×150 C array. See figure 4.9.

```
MPI_Conn conn;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Conn_size( conn, &gsize);
MPI_Conn_rank( conn, &myrank );
```

```
stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                    root, comm);
```

4.7 Gather-to-All

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.

The outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to

```
MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                                              recvtype,root,comm),
```

for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                   void* recvbuf, int *recvcounts, int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
               RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The jth block of data sent

from each process is received by every process and placed in the jth block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

The outcome is as if all processes executed calls to

```
MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                                recvtype,root,comm),
```

for `root = 0 , ..., n-1`. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

4.7.1 EXAMPLES USING MPI_ALLGATHER, MPI_ALLGATHERV

Example 4.14 The all-gather version of Example 4.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```
MPI_Comm conn;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size( conn, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.

4.8 All-to-All Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Conn conn)
```

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

The outcome is as if each process executed a send to each process (itself included) with a call to,

```
MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount,
sendtype, i, ...),
```

and a receive from every other process with a call to,

```
MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, i, ...).
```

All arguments on all processes are significant. The argument comm must have identical values on all processes.

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
              MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
              int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

The jth block sent from process i is received by process j and is placed in the ith block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcount[j], sendtype at process i must be equal to the type signature associated with recvcount[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

```
MPI_Send(sendbuf + displs[i] · extent(sendtype), sendcounts[i],
sendtype, i, ...),
```

and received a message from every other process with a call to

```
MPI_Recv(recvbuf + displs[i] · extent(recvtype), recvcounts[i],
recvtype, i, ...).
```

All arguments on all processes are significant. The argument comm must have identical values on all processes.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

4.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all the members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction at one node, an all-reduce that returns this result at all nodes, and a scan (parallel prefix) operation. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

4.9.1 REDUCE

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)).

Section 4.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes each operation can be applied to. In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 4.9.4.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

> *Advice to implementors.* It is strongly recommended that MPI_REDUCE be implemented so that the same result will be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (*End of advice to implementors.*)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 4.9.2 and Section 4.9.3. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 4.9.4.