MPI: A Message-Passing Interface Standard

Extension: Nonblocking Collective Operations

(draft)

Message Passing Interface Forum

December 31, 2008

Contents

Chapter 5

Collective Communication

5.1 Introduction and Overview

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

- MPI_BARRIER, MPI_IBARRIER: Barrier synchronization across all members of a group (Section 5.3).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 5.4). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHERMPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 5.5). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 5.6). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 5.8). This is shown as "alltoall" in Figure 5.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group and a variation where the result is returned to only one member (Section 5.9).
- MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10).
- MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11).

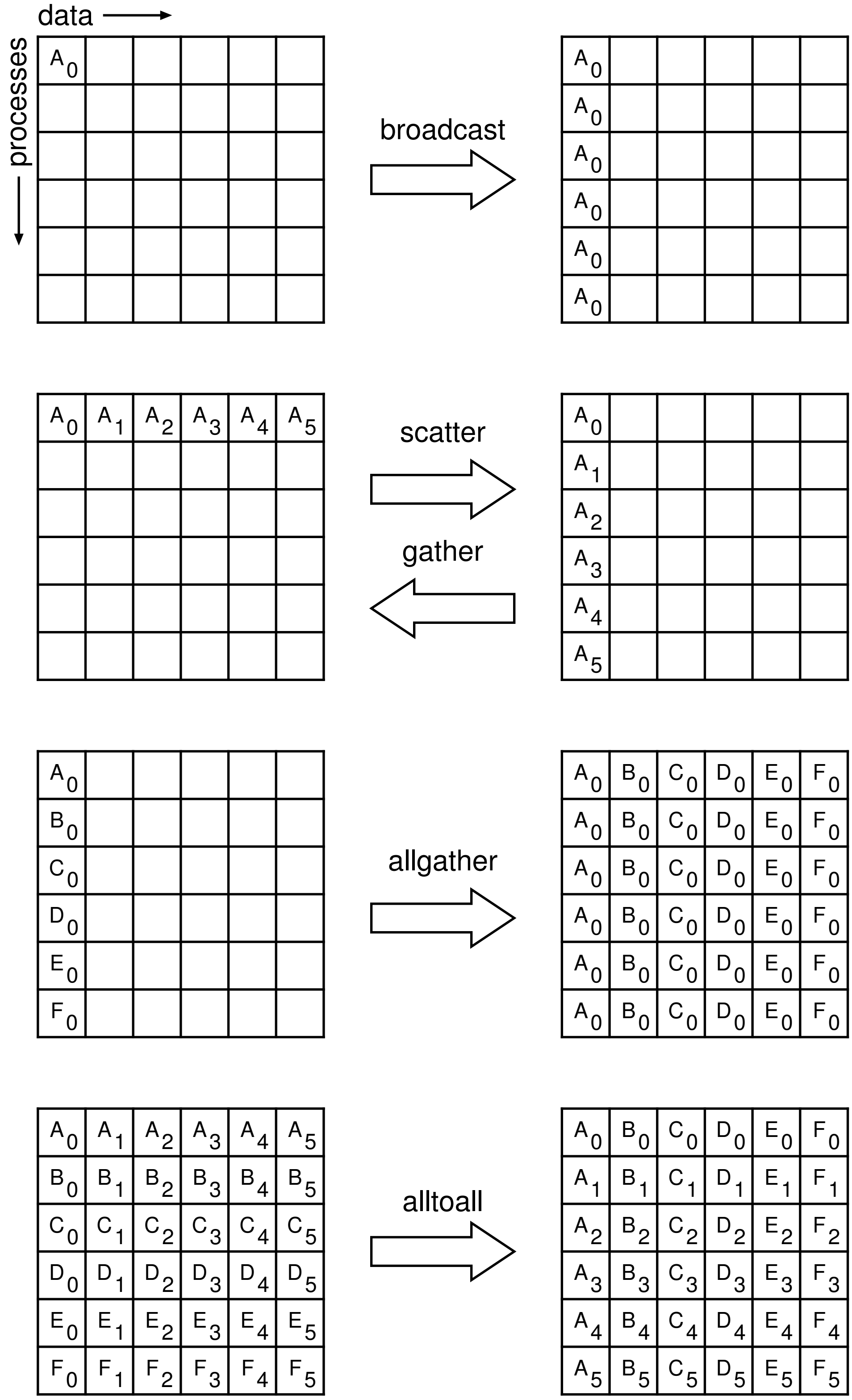

Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 5.2. 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 **??**. Several collective routines such as broadcast and gather have a single originating or receiving process. Such a process is called the *root*. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root. The reader is referred to Chapter **??** for information concerning communication buffers, general datatypes and type matching rules, and to Chapter **??** 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. Different type maps (the layout in memory, see Section **??**) between sender and receiver are still allowed. The collective operations do not have a message tag argument.

Collective ~~routine calls~~operations can (but are not required to) ~~return~~complete locally as soon as ~~their~~the caller's participation in the collective communication is ~~complete~~finished. The local completion of a ~~call~~collective operation 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 implied by in the description of the operation). A blocking operation is complete as soon as the call returns. A nonblocking (immediate) call requires a separate completion operation, cf. **??** (Section 3.7). Thus, a collective communication ~~call~~operation may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier ~~function~~operation.

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 5.13.

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 nonblocking collective functions, then tags (or a similar mechanism) might need to be added so as to allow the dis-ambiguation 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 5.13. (*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 might 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 5.13. (*End of advice to implementors.*)

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are *not* correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter **??**. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intracommunicator can be thought of as an indentifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

5.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine with matching arguments.

In many cases, collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

Rationale. The "in place" operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits aliasing of arguments; the approach of using a special value to denote "in place" operation eliminates that difficulty. (*End of rationale.*)

Advice to users. By allowing the "in place" option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes `INTENT` must mark these as `INOUT`, not `OUT`.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has.

Some intracommunicator collective operations do not support the "in place" option (e.g., MPI_ALLTOALLV). (*End of advice to users.*)

5.2.2 Applying Collective Operations to Intercommunicators

To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [5]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER

All-To-One All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV
- MPI_REDUCE, MPI_IREDUCE

One-To-All One process contributes to the result. All processes receive the result.

- MPI_BCAST, MPI_IBCAST
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV

Other Collective operations that do not fit into one of the above categories.

- MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN
- MPI_BARRIER, MPI_IBARRIER

The MPI_BARRIER and MPI_IBARRIER operation ~~doess~~ do not fit into this classification since no data is being moved (other than the implicit fact that a barrier has been called). The data movement patterns of MPI_SCAN, MPI_ISCAN ~~and~~, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:

- MPI_BARRIER, MPI_IBARRIER
- MPI_BCAST, MPI_IBCAST
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV,
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW,
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE,
- MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER.

In C++, the bindings for these functions are in the `MPI::Comm` class. However, since the collective operations do not make sense on a C++ `MPI::Comm` (as it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual.

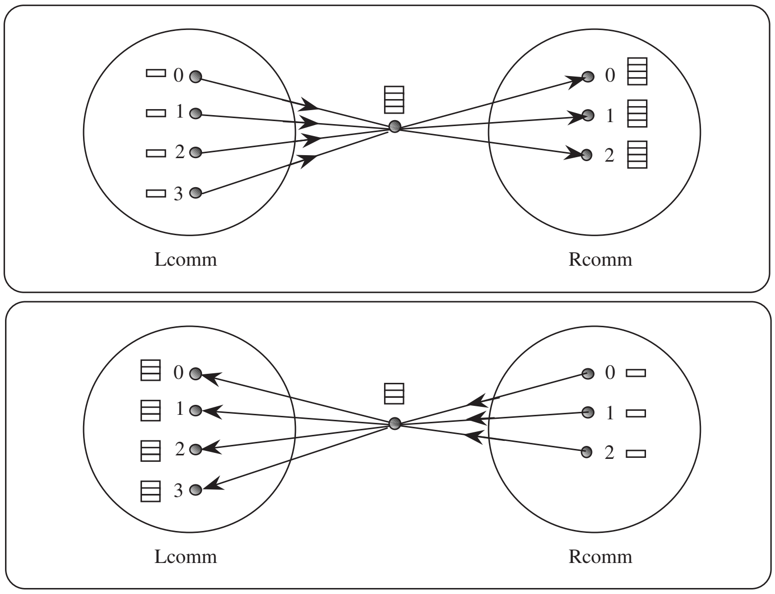

Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine. In addition, processes in the same group must call the routine with matching arguments.

Note that the “in place” option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process to itself.

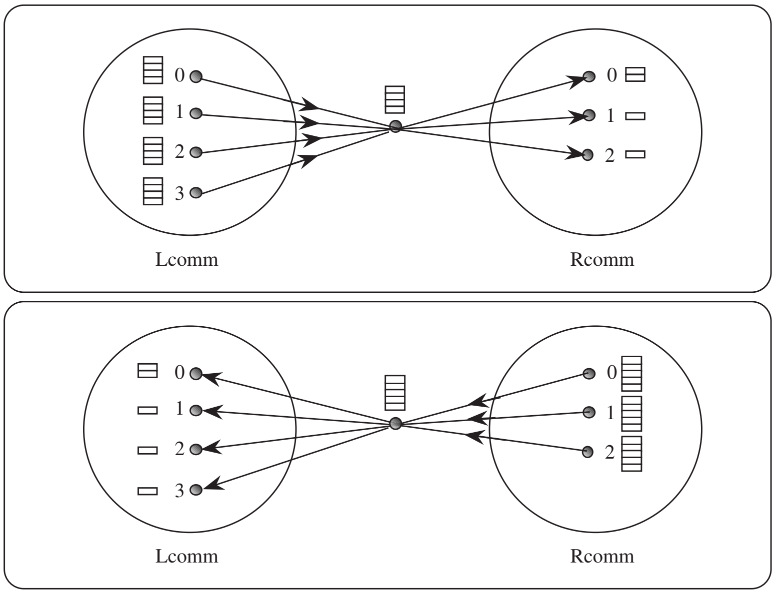

Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

For intercommunicator collective communication, if the operation is rooted (e.g., broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, for the group containing the root process, all processes in the group must call the routine using a special argument for the root. For this, the root process uses the special root value MPI_ROOT; all other processes in the same group as the root use MPI_PROC_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is unrooted (e.g., alltoall), then the transfer is bidirectional.

Rationale. Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

5.3 Barrier Synchronization

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier(MPI_Comm comm)
```

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

```
void MPI::Comm::Barrier() const = 0
```

If comm is an intracommunicator, 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.

If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in one group (group A) of the intercommunicator may exit the barrier when all of the processes in the other group (group B) have entered the barrier.

5.4 Broadcast

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

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative 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
```

```
void MPI::Comm::Bcast(void* buffer, int count,
              const MPI::Datatype& datatype, int root) const = 0
```

If comm is an intracommunicator, 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 the group using the same arguments for comm and root. On return, the content of root's buffer is copied to all other 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.

The "in place" option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1 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.

5.5 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 (non-negative 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 (non-negative 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
```

```
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

If comm is an intracommunicator, 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

$$\mathtt{MPI\_Send}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{root}, ...),$$

and the root had executed n calls to

$$\mathtt{MPI\_Recv}(\mathtt{recvbuf} + \mathtt{i} \cdot \mathtt{recvcount} \cdot \mathtt{extent}(\mathtt{recvtype}), \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{i}, ...),$$

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 each process 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, and 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.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

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 (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	non-negative 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
```

```
void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype, int root) const = 0
```

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.

If comm is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

$$\texttt{MPI\_Send}(\texttt{sendbuf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{root}, ...),$$

and the root executes n receives,

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[j] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[j], \texttt{recvtype}, \texttt{i}, ...).$$

The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype).

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 5.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and 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.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

Example 5.2 Gather 100 ints from every process in group to root. See figure 5.4.

```
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 5.3 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);
```

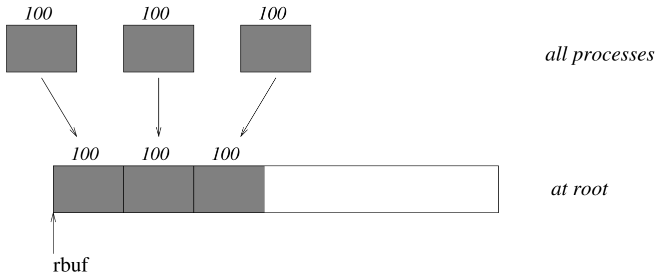

Figure 5.4: The root process gathers 100 ints from each process in the group.

```
   rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.4 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;
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 5.5 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* ≥ 100. See Figure 5.5.

```
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;
```

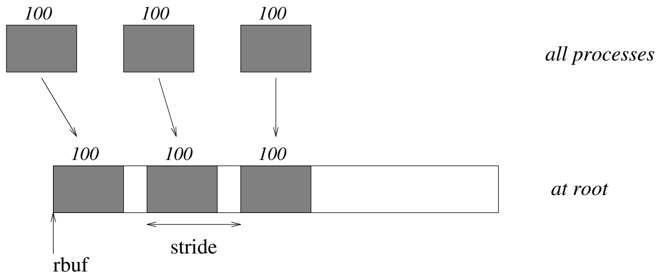

Figure 5.5: The root process gathers 100 `ints` from each process in the group, each set is placed `stride` `ints` apart.

```
    }
    MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                             root, comm);
```

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

Example 5.6 Same as Example 5.5 on the receiving side, but send the 100 `ints` from the 0th column of a 100×150 `int` array, in C. See Figure 5.6.

```
    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 5.7 Process `i` sends `(100-i)` `ints` from the `i`-th 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 5.7.

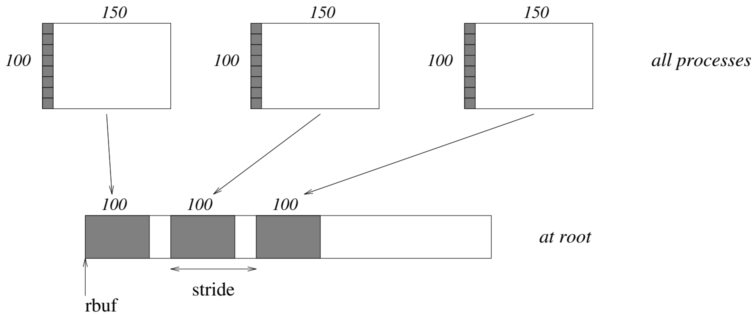

Figure 5.6: The root process gathers column `0` of a 100×150 C array, and each set is placed `stride` `int`s apart.

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

...

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 5.8 Same as Example 5.7, 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 we read a column of a C array. A similar thing was done in Example **??**, Section **??**.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
```

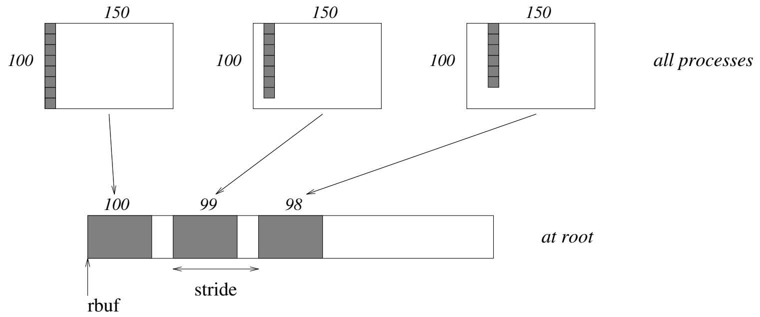

Figure 5.7: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride ints apart.

```
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,type[2];
int *displs,i,*rcounts;

...

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;
}
/* 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, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                           root, comm);
```

Example 5.9 Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.

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

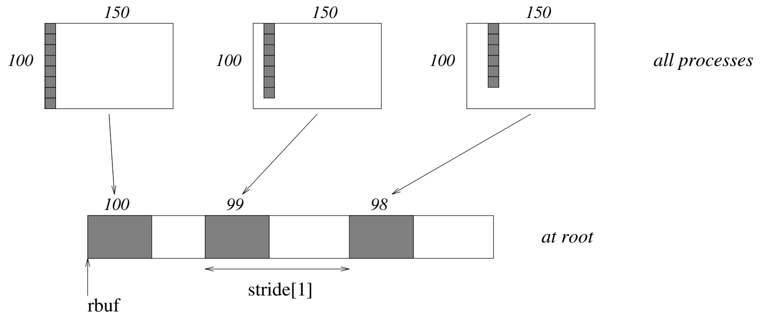

Figure 5.8: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride[i]` `int`s apart (a varying stride).

```
...

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 5.10 Process `i` sends `num` `int`s from the `i`-th 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_Comm_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);
```

5.6 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 (non-negative 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 (non-negative 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
```

```
void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

If comm is an intracommunicator, the outcome is *as if* the root executed n send operations,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

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, and 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.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

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

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative 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 (non-negative 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
```

```
void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], const MPI::Datatype& sendtype,
             void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
             int root) const = 0
```

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 an additional argument, displs.

If comm is an intracommunicator, the outcome is as if the root executed n send operations,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

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.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and 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.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

The examples in this section use intracommunicators.

Example 5.11 The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.

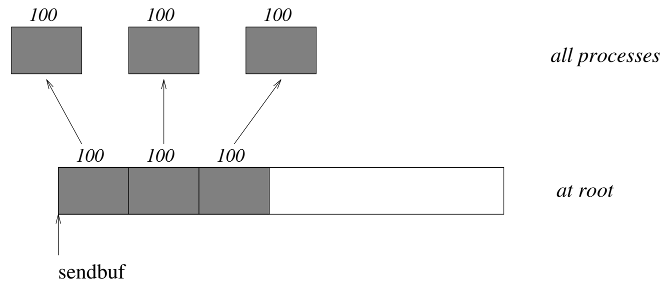

Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

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

Example 5.12 The reverse of Example 5.5. 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* ≥ 100. See Figure 5.10.

```
MPI_Comm 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));
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 5.13 The reverse of Example 5.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the i-th column of a 100×150 C array. See Figure 5.11.

```
MPI_Comm comm;
```

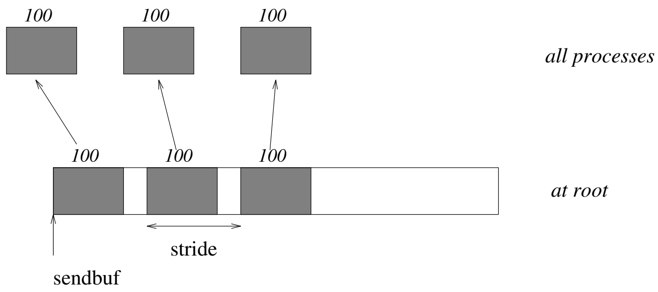

Figure 5.10: The root process scatters sets of 100 `int`s, moving by `stride` `int`s from send to send in the scatter.

```
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, 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
 * 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);
```

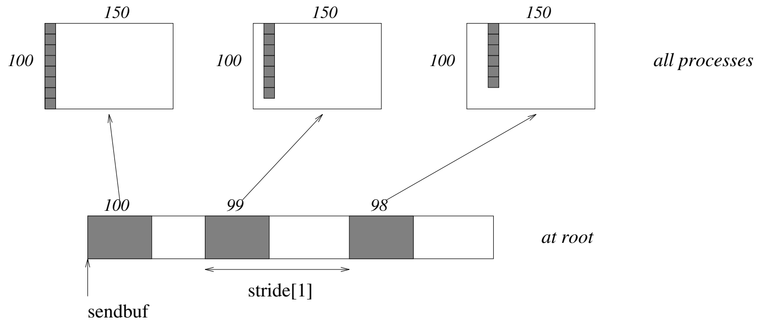

Figure 5.11: The root scatters blocks of `100-i` `int`s into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` `int`s apart.

5.7 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 (non-negative 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 (non-negative 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
```

```
void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

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

If comm is an intracommunicator, 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.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount $= 0$ for the communication in the reverse direction.

(*End of advice to users.*)

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 (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative 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
```

```
void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const
                MPI::Datatype& sendtype, void* recvbuf,
                const int recvcounts[], const int displs[],
                const MPI::Datatype& recvtype) const = 0
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th 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.

If comm is an intracommunicator, 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.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV

The examples in this section use intracommunicators.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &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.

5.8 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 (non-negative 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 (non-negative 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_Comm comm)
```

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

```
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

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

If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

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

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \cdot \texttt{recvcount} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

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

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

> *Advice to users.* When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction.
>
> (*End of advice to users.*)

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative 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	non-negative 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
```

```
void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype& sendtype,
             void* recvbuf, const int recvcounts[], const int rdispls[],
             const MPI::Datatype& recvtype) const = 0
```

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.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th 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,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\mathrm{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\mathrm{i}], \texttt{sendtype}, \texttt{i}, ...),$$

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

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[\mathrm{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\mathrm{i}], \texttt{recvtype}, \texttt{i}, ...).$$

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

No "in place" option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

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.*)

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, 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 (array of non-negative integers)
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
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 (array of non-negative integers)
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
IN	comm	communicator (handle)

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
             MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
             int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR
```

```
void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype sendtypes[], void*
             recvbuf, const int recvcounts[], const int rdispls[], const
             MPI::Datatype recvtypes[]) const = 0
```

MPI_ALLTOALLW is the most general form of All-to-all. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] 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

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{sdispls}[\texttt{i}], \texttt{sendcounts}[\texttt{i}], \texttt{sendtypes}[\texttt{i}], \texttt{i}, ...),$$

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

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{rdispls}[\texttt{i}], \texttt{recvcounts}[\texttt{i}], \texttt{recvtypes}[\texttt{i}], \texttt{i}, ...).$$

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

No "in place" option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have `sendcounts[i] = 0`, this achieves an MPI_SCATTERW function. (*End of rationale.*)

5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all 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 to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

5.9.1 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 (non-negative 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
```

```
void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op, int root)
             const = 0
```

If comm is an intracommunicator, 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 $\mathsf{recvbuf}(1) = \text{global}\max(\mathsf{sendbuf}(1))$ and $\mathsf{recvbuf}(2) = \text{global}\max(\mathsf{sendbuf}(2))$.

Section 5.9.2, 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 5.9.5.

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 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 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. 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 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. Safest is to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in “send” buffers. Overlapping datatypes in “receive” buffers are erroneous and may give unpredictable results.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and

MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR
Fortran integer:	MPI_INTEGER
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

Now, the valid datatypes for each option is specified below.

Op	Allowed Types
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

The following examples use intracommunicators.

Example 5.15 A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m)       ! local slice of array
REAL c                ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr

! local sum
sum = 0.0
```

```
DO i = 1, m
   sum = sum + a(i)*b(i)
END DO

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN
```

Example 5.16 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! return result at node zero (and garbage at the other nodes)
RETURN
```

5.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR (which represents printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

> *Advice to users.* The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

5.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. Thus, if each process supplies a value and its rank within the group, then a reduce operation with op = MPI_MAXLOC will return the maximum value and the rank of the first process with that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered according to the first component of each pair, and ties are resolved according to the second component.

The reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, for Fortran, by having the MPI-provided type consist of a pair of the same type as value, and coercing the index to this type also. In C, the MPI-provided pair type has distinct types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide a datatype argument that represents a pair (value and index). MPI provides nine such predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with each of the following datatypes.

Fortran:

Name	Description
MPI_2REAL	pair of REALs
MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables
MPI_2INTEGER	pair of INTEGERs

C:

Name	Description
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

The datatype MPI_2REAL is *as if* defined by the following (see Section **??**).

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.

The datatype MPI_FLOAT_INT is *as if* defined by the following sequence of instructions.

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

Example 5.17 Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
...
/* each process has an array of 30 double: ain[30]
 */
double ain[30], aout[30];
int  ind[30];
struct {
    double val;
    int   rank;
} in[30], out[30];
int i, myrank, root;
```

```
    MPI_Comm_rank(comm, &myrank);
    for (i=0; i<30; ++i) {
        in[i].val = ain[i];
        in[i].rank = myrank;
    }
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
    /* At this point, the answer resides on process root
     */
    if (myrank == root) {
        /* read ranks out
         */
        for (i=0; i<30; ++i) {
            aout[i] = out[i].val;
            ind[i] = out[i].rank;
        }
    }
```

Example 5.18 Same example, in Fortran.

```
    ...
    ! each process has an array of 30 double: ain(30)

    DOUBLE PRECISION ain(30), aout(30)
    INTEGER ind(30)
    DOUBLE PRECISION in(2,30), out(2,30)
    INTEGER i, myrank, root, ierr

    CALL MPI_COMM_RANK(comm, myrank, ierr)
    DO I=1, 30
        in(1,i) = ain(i)
        in(2,i) = myrank    ! myrank is coerced to a double
    END DO

    CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                              comm, ierr)
    ! At this point, the answer resides on process root

    IF (myrank .EQ. root) THEN
        ! read ranks out
        DO I= 1, 30
            aout(i) = out(1,i)
            ind(i) = out(2,i)  ! rank is coerced back to an integer
        END DO
    END IF
```

Example 5.19 Each process has a non-empty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.

```
#define  LEN   1000

float val[LEN];        /* local array of values */
int count;             /* local number of values */
int myrank, minrank, minindex;
float minval;

struct {
    float value;
    int   index;
} in, out;

    /* local minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce(in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
    /* At this point, the answer resides on process root
     */
if (myrank == root) {
    /* read answer out
     */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(function, commute, op)

IN	function	user defined function (function)
IN	commute	`true` if commutative; `false` otherwise.
OUT	op	operation (handle)

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
```

```
MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

```
void MPI::Op::Init(MPI::User_function* function, bool commute)
```

MPI_OP_CREATE binds a user-defined global operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative. If commute = `true`, then the operation should be both commutative and associative. If commute = `false`, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = `true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

function is the user-defined function, which must have the following four arguments: invec, inoutvec, len and datatype.

The ISO C prototype for the function is the following.

```
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
             MPI_Datatype *datatype);
```

The Fortran declaration of the user-defined function appears below.

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, TYPE
```

The C++ declaration of the user-defined function appears below.

```
typedef void MPI::User_function(const void* invec, void *inoutvec, int len,
             const Datatype& datatype);
```

The datatype argument is a handle to the data type that was passed into the call to MPI_REDUCE. The user reduce function should be written such that the following holds: Let u[0], ... , u[len-1] be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let v[0], ... , v[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0], ... , w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then w[i] = u[i]∘v[i], for i=0 , ... , len-1, where ∘ is the reduce operation that the function computes.

Informally, we can think of invec and inoutvec as arrays of len elements that function is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: I.e, the function returns in inoutvec[i] the value invec[i] $\circ$ inoutvec[i], for $i = 0, \ldots, count - 1$, where $\circ$ is the combining operation computed by the function.

Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE not supporting the "in place" option.

```
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
```

```
        */
       if (rank == root) {
           MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
       }
       if (rank == groupsize-1) {
           MPI_Send(sendbuf, count, datatype, root, ...);
       }
       if (rank == root) {
           MPI_Wait(&req, &status);
       }
```

The reduction computation proceeds, sequentially, from process `0` to process `groupsize-1`. This order is chosen so as to respect the order of a possibly non-commutative operator defined by the function `User_reduce()`. A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size `len` <`count`.

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPI_REDUCE handles these functions as a special case. (*End of advice to implementors.*)

MPI_OP_FREE(op)

INOUT	op	operation (handle)

```
int MPI_op_free(MPI_Op *op)
```

```
MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR
```

```
void MPI::Op::Free()
```

Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.

Example of User-defined Reduce

It is time for an example of user-defined reduction. The example in this section uses an intracommunicator.

Example 5.20 Compute the product of an array of complex numbers, in C.

```
typedef struct {
    double real,imag;
} Complex;

/* the user-defined function
```

```
 */
void myProd(Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
                   inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                   inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* explain to MPI how type Complex is defined
     */
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
    MPI_Type_commit(&ctype);
    /* create the complex-product user-op
     */
    MPI_Op_create(myProd, True, &myOp);

    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);

    /* At this point, the answer, which consists of 100 Complexes,
     * resides on process root
     */
```

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

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

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

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
                  const MPI::Datatype& datatype, const MPI::Op& op) const = 0
```

If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as MPI_REDUCE except that the result appears in the receive buffer of all the group members.

Advice to implementors. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 5.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
```

```
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN
```

5.10 Reduce-Scatter

MPI includes a variant of the reduce operations where the result is scattered to all processes in a group on return.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

```
void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
             int recvcounts[], const MPI::Datatype& datatype,
             const MPI::Op& op) const = 0
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first does an element-wise reduction on vector of count $= \sum_i$ recvcounts[i] elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector of results is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcounts[i] elements. The i-th segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is scattered among processes in group B, and vice versa. Within each group, all processes provide the same recvcounts argument, and the sum of the recvcounts entries should be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

5.11 Scan

5.11.1 Inclusive Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

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

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

```
void MPI::Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const
```

If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,...,i (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.

This operation is invalid for intercommunicators.

5.11.2 Exclusive Scan

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	intracommunicator (handle)

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

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

```
void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const
```

If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not signficant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank $i > 1$, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

No "in place" option is supported.

This operation is invalid for intercommunicators.

Advice to users. As for MPI_SCAN, MPI does not specify which processes may call the operation, only that the result be correctly computed. In particular, note that the process with rank 1 need not call the MPI_Op, since all it needs to do is to receive the value from the process with rank 0. However, all processes, even the processes with ranks zero and one, must provide the same op. (*End of advice to users.*)

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan.

No in-place version is specified for MPI_EXSCAN because it is not clear what this means for the process with rank zero. (*End of rationale.*)

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.22 This example uses a user-defined operation to produce a *segmented scan*. A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

values	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
logicals	0	0	1	1	1	0	0	1
result	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

The operator that produces this effect is,

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ j \end{pmatrix},$$

where,

$$w = \begin{cases} u + v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases} .$$

Note that this is a non-commutative operator. C code that implements it is given below.

```
typedef struct {
    double val;
    int log;
} SegScanPair;

/* the user-defined function
 */
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
                                                MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if (in->log == inout->log)
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

Note that the inout argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
int i,base;
SeqScanPair  a, answer;
MPI_Op       myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint     disp[2];
int          blocklen[2] = { 1, 1};
MPI_Datatype sspair;

/* explain to MPI how type SegScanPair is defined
 */
MPI_Address(a, disp);
MPI_Address(a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;
MPI_Type_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
/* create the segmented-scan user-op
 */
MPI_Op_create(segScan, 0, &myOp);
...
MPI_Scan(a, answer, 1, sspair, myOp, comm);
```

5.12 Nonblocking Collective Operations

As described in Chapter **??** (Section 3.7), the performance of many systems can be improved by overlapping communication and computation. Nonblocking collective operations combine the potential to utilize overlap and avoid synchronization of nonblocking point-to-point operations with the optimized implementation and message scheduling of collective operations [1, 4]. One way of doing this would be to perform the collective operation in a separate thread. An alternative mechanism that often leads to better performance (i.e., avoids context switching, scheduler overheads, and thread management [2]) is the use of nonblocking collective communication. The model is similar to point-to-point communications **??** (Section 3). A nonblocking start call is used to initiate a collective communication, which is eventually completed by a separate call. As in the nonblocking point-to-point case, the communication may progress independently of the computations at all participating processes. Nonblocking collective communication can also be used to mitigate synchronizing effects of collective operations by running them in the "background." In addition to enabling communication-computation overlap, nonblocking collective operations can be used to perform collective operations on overlapping communicators which would lead to deadlocks with blocking operations. The semantic advantages can also be useful in combination with point-to-point communication.

As in the point-to-point case, all start calls are local and return immediately with a request handle **??** (Section 3.7.1), irrespective of the status of other processes. Multiple nonblocking collective communications can be outstanding on a single communicator. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in "pathological" cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

A nonblocking collective call indicates that the system may start copying data out of the send buffer and into the receive buffer. All associated send buffers should not be modified and all associated receive buffers should not be accessed between the initiation and the completion of a nonblocking collective operation. Collective operations complete locally when the local part of the operation has been performed (i.e., the semantics of the collective operation are guaranteed) and all buffers can be accessed. Similarly to the blocking case, completion does not imply that other processes have completed or even started the operation. However, calling blocking completion functions (e.g., MPI_WAIT) may synchronize the calling processes.

All request test and completion functions (e.g., MPI_WAIT) described in Section **??** (Section 3.7.3) are supported for nonblocking collective communications. Mixing of different request types (i.e., any combination of collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions **??** (Section 3.7.5) is allowed. Calling MPI_REQUEST_FREE or MPI_CANCEL for a request associated with a nonblocking collective is erroneous.

Collective operations do not have a tag argument.

> *Rationale.* Freeing a nonblocking collective request is erroneous because all collectives include a receive action for at least some of the processes. Thus, freeing these requests is erroneous since there is no portable way to verify that the operation is completed and the receive buffer can be used after an active receive request is freed (see Section **??** (3.7.3)).
>
> Cancelling a request is not supported because the semantics of this operation cannot be defined clearly.
>
> The use of tags for collective operations would prevent certain hardware-optimizations and is consistent with blocking collective operations. (*End of rationale.*)

> *Advice to implementors.* Nonblocking collective operations can be implemented with local execution schedules [3] using normal point-to-point communication and a reserved tag-space. (*End of advice to implementors.*)

All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, if some process calls a collective operation, all other processes in the communicator must eventually call the *same* collective operation, and no other collective operation in between. This is consistent with the ordering rules for blocking collective operations in threaded environments. Progression rules for nonblocking collectives are similar to progression of nonblocking point-to-point operations, refer to **??** (Section 3.7.4).

> *Rationale.* Matching blocking and nonblocking collectives is not allowed because the implementation might use different communication algorithms for the two cases. Blocking collectives would be optimized for their running time while nonblocking collectives could find an equilibrium between time to completion, CPU overhead and asynchronous progression. (*End of rationale.*)

> *Advice to users.* The matching semantics are different than for point-to-point operations. If matching of blocking and nonblocking collective operations is required,

then the user can use a nonblocking collective immediately followed by a call to wait in order to emulate blocking behavior. (*End of advice to users.*)

Status objects that are passed to request test and completion functions (e.g., MPI_WAIT) will be ignored if the request is associated with a nonblocking collective routine.

Advice to users. The user is encouraged to pass MPI_STATUS{ES}_IGNORE as status object to all requests of nonblocking collective routines. (*End of advice to users.*)

5.12.1 Nonblocking Barrier Synchronization

MPI_IBARRIER(comm , request)

IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

```
MPI_IBARRIER(COMM, REQUEST, IERROR)
    INTEGER COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Ibarrier() const = 0
```

This call starts a nonblocking barrier. The operation finishes locally after every process in the communicator called MPI_IBARRIER.

Advice to users. A nonblocking barrier can be used to hide latency. Moving independent computations between the MPI_IBARRIER and the subsequent completion call can overlap the barrier latency and therefore shorten possible waiting times. The semantic properties are also useful when mixing collectives and point-to-point messages. (*End of advice to users.*)

5.12.2 Nonblocking Broadcast

MPI_IBCAST(buffer, count, datatype, root, comm, request)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm, MPI_Request *request)
```

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

```
MPI::Request MPI::Comm::Ibcast(void* buffer, int count,
             const MPI::Datatype& datatype, int root) const = 0
```

This call starts a nonblocking broadcast. The data placements after the operation completes are identical as after a call to the blocking MPI_BCAST.

Example using MPI_IBCAST

The example in this section uses intracommunicators.

Example 5.23 Start a broadcast of 100 `int`s from process `0` to every process in the group, perform some computation on independent data, and then complete the outstanding broadcast operation.

```
MPI_Comm comm;
int array1[100], array2[100];
int root=0;
MPI_Request req;
...
MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
compute(array2, 100);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

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.

5.12.3 Nonblocking Gather

MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative 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 (non-negative 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)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Igather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

This call starts a nonblocking gather. The data placements after the operation completes are identical as after a call to the blocking MPI_GATHER.

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)

IN	recvcounts	non-negative 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)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Igatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_IGATHERV extends the functionality of MPI_IGATHER by allowing a varying count of data from each process. The memory movement after completion is identical as for MPI_GATHERV.

5.12.4 Nonblocking Scatter

MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative 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 (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Iscatter(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_ISCATTER starts the reverse data movement as MPI_IGATHER. The data movement performed is equivalent to MPI_SCATTER.

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

IN	sendbuf	address of send buffer (choice, significant only at root)

IN	sendcounts	non-negative 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 (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Iscatterv(const void* sendbuf,
             const int sendcounts[], const int displs[],
             const MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_ISCATTERV starts the reverse data movement as MPI_IGATHERV. The data movement performed is equivalent to MPI_SCATTERV.

5.12.5 Nonblocking Gather-to-all

MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative 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 (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Iallgather(const void* sendbuf, int sendcount,
             const MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

The data movement after an MPI_IALLGATHER operation completes is identical to MPI_ALLGATHER.

MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)

IN	recvcounts	non-negative 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)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Iallgatherv(const void* sendbuf, int sendcount,
             const MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype) const = 0
```

The data movement after completion of MPI_IALLGATHERV is identical as if MPI_ALLGATHERV returned.

5.12.6 Nonblocking All-to-All Scatter/Gather

MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (non-negative 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 (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Ialltoall(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf, int recvcount,
              const MPI::Datatype& recvtype) const = 0
```

The data movement after an MPI_IALLTOALL operation completes is identical to MPI_ALLTOALL.

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length 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	non-negative integer array (of length 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)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Ialltoallv(const void* sendbuf,
             const int sendcounts[], const int sdispls[],
             const MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             const MPI::Datatype& recvtype) const = 0
```

MPI_IALLTOALLV adds flexibility to MPI_IALLTOALL 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.

MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor (array of non-negative integers)
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each processor (array of non-negative integers)
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ialltoallw(void *sendbuf, int sendcounts[], int sdispls[],
             MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
             int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm,
             MPI_Request *request)
```

```
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, REQUEST, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Ialltoallw(const void* sendbuf, const int
             sendcounts[], const int sdispls[], const MPI::Datatype
             sendtypes[], void* recvbuf, const int recvcounts[], const int
             rdispls[], const MPI::Datatype recvtypes[]) const = 0
```

MPI_IALLTOALLW is the nonblocking variant of MPI_ALLTOALLW. It starts a nonblocking all-to-all operation which delivers the same results as MPI_ALLTOALLW after completion.

5.12.7 Nonblocking Reduce

MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)

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 (non-negative 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)
OUT	request	communication request (handle)

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

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

```
MPI::Request MPI::Comm::Ireduce(const void* sendbuf, void* recvbuf,
             int count, const MPI::Datatype& datatype, const MPI::Op& op,
             int root) const = 0
```

MPI_IREDUCE is the nonblocking variant of MPI_REDUCE. It starts a nonblocking reduction operation which delivers the same results as MPI_REDUCE after completion.

Advice to implementors. It is strongly recommended that MPI_IREDUCE is implemented so that the same result is 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.*)

5.12.8 Nonblocking All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Iallreduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

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

```
MPI::Request MPI::Comm::Iallreduce(const void* sendbuf, void* recvbuf,
             int count, const MPI::Datatype& datatype, const MPI::Op& op)
             const = 0
```

MPI_IALLREDUCE is the nonblocking variant of MPI_ALLREDUCE. It starts a nonblocking reduction-to-all operation which delivers the same results as MPI_ALLREDUCE after completion.

5.12.9 Nonblocking Reduce-Scatter

MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ireduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

```
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Ireduce_scatter(const void* sendbuf, void* recvbuf,
             int recvcounts[], const MPI::Datatype& datatype,
             const MPI::Op& op) const = 0
```

MPI_IREDUCE_SCATTER is the nonblocking variant of MPI_REDUCE_SCATTER. It starts a nonblocking reduce-scatter operation which delivers the same results as MPI_REDUCE_SCATTER after completion.

5.12.10 Nonblocking Inclusive Scan

MPI_ISCAN(sendbuf, recvbuf, count, datatype, op, comm, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Iscan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

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

```
MPI::Request MPI::Intracomm::Iscan(const void* sendbuf, void* recvbuf,
             int count, const MPI::Datatype& datatype, const MPI::Op& op)
             const
```

MPI_ISCAN is the nonblocking variant of MPI_SCAN. It starts a nonblocking scan operation which delivers the same results as MPI_SCAN after completion.

5.12.11 Nonblocking Exclusive Scan

MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	intracommunicator (handle)
OUT	request	communication request (handle)

```
int MPI_Iexscan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

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

```
MPI::Request MPI::Intracomm::Iexscan(const void* sendbuf, void* recvbuf,
             int count, const MPI::Datatype& datatype, const MPI::Op& op)
             const
```

MPI_IEXSCAN is the nonblocking variant of MPI_EXSCAN. It starts a nonblocking exclusive scan operation which delivers the same results as MPI_EXSCAN after completion.

5.13 Correctness

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators.

Example 5.24 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

We assume that the group of comm is {0,1}. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.25 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

Assume that the group of comm0 is {0,1}, of comm1 is {1, 2} and of comm2 is {2,0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 5.26 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 5.27 An unsafe, non-deterministic program.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then, messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved.

It is the implementor's responsibility to ensure that point-to-point messages are not confused with collective messages. One way to accomplish this is, whenever a communicator is created, to also create a "hidden communicator" for collective communication. One could achieve a similar effect more cheaply, for example, by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication. (*End of advice to implementors.*)

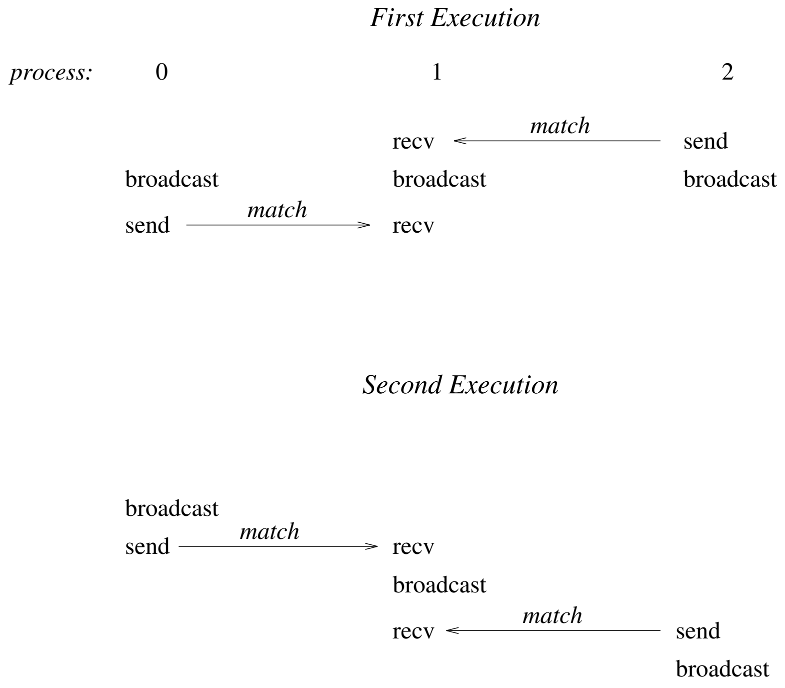

Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

Example 5.28 Blocking and nonblocking collective operations can be interleaved, i.e., a blocking collective operation can be posted even if there is a nonblocking collective operation outstanding.

```
MPI_Request req;

MPI_Ibarrier(comm, &req);
MPI_Bcast(buf1, count, type, 0, comm);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Each process starts a nonblocking barrier operation, participates in a blocking broadcast and then waits until every other process started the barrier operation. This effectively turns the broadcast into a synchronizing broadcast with possible communication/communication overlap (MPI_Bcast is allowed, but not required to synchronize).

Example 5.29 The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous matching */
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
```

```
        break;
    case 1:
        /* erroneous matching */
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be legal:

```
MPI_Request req;
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

Example 5.30 Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;

switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &req);
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      break;
    case 1:
      MPI_Ibarrier(comm, &req);
```

```
        MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

The MPI library must progress and finish the barrier in the MPI_Recv call which eventually completes the barrier operation on both processes and enables the matching MPI_Send.

Example 5.31 Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

```
MPI_Request reqs[2];

switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &reqs[0]);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
      break;
    case 1:
      MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
      MPI_Ibarrier(comm, &reqs[1]);
      MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
      break;
}
```

The Waitall call returns only after the barrier and the receive completed.

Example 5.32 Multiple nonblocking collective operations can be outstanding on a single communicator and match in order.

```
MPI_Request reqs[3];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

Advice to users. Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. (*End of advice to users.*)

Advice to implementors. The use of pipelining may generate many outstanding requests. Thus, the number of outstanding requests could only be limited by physical memory. A hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. (*End of advice to implementors.*)

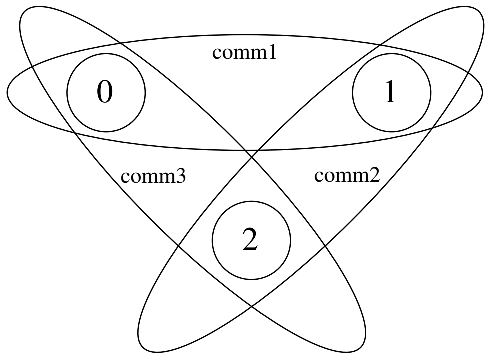

Figure 5.13: Example with overlapping communicators.

Example 5.33 Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 5.13). The following example is started with three processes and three communicators. The first communicator `comm1` includes ranks 0 and 1, `comm2` includes ranks 1 and 2 and `comm3` spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there exists no deadlock-free order to invoke them. However, nonblocking collective operations can easily be used to achieve this task.

```
MPI_Request reqs[2];

switch(rank) {
    case 0:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
    case 1:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break;
    case 2:
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
}
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

Advice to users. This method can be useful if overlapping neighboring regions (halo or ghost zones) are used in collective operations. (*End of advice to users.*)

Bibliography

[1] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12

[2] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. 5.12

[3] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007. 5.12

[4] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication Optimization for Medical Image Reconstruction Algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12

[5] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. `http://www.erc.msstate.edu/mpi/mpix.html`. 5.2.2