

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

February 9, 2015

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

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 and Section 5.12.1).
- `MPI_BCAST`, `MPI_IBCAST`: Broadcast from one member to all members of a group (Section 5.4 and Section 5.12.2). This is shown as “broadcast” in Figure 5.1.
- `MPI_GATHER`, `MPI_IGATHER`, `MPI_GATHERV`, `MPI_IGATHERV`: Gather data from all members of a group to one member (Section 5.5 and Section 5.12.3). 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 and Section 5.12.4). 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 and Section 5.12.5). 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) (Section 5.8 and Section 5.12.6). This is shown as “complete exchange” 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 (Section 5.9.6 and Section 5.12.8) and a variation where the result is returned to only one member (Section 5.9 and Section 5.12.7).
- `MPI_REDUCE_SCATTER_BLOCK`, `MPI_IREDUCE_SCATTER_BLOCK`, `MPI_REDUCE_SCATTER`, `MPI_IREDUCE_SCATTER`: A combined reduction and scatter operation (Section 5.10, Section 5.12.9, and Section 5.12.10).

- `MPI_SCAN`, `MPI_ISCAN`, `MPI_EXSCAN`, `MPI_IEXSCAN`: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

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 4. 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 4 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 6 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 4.1) between sender and receiver are still allowed.

Collective operations can (but are not required to) complete as soon as the caller’s participation in the collective communication is finished. A blocking operation is complete as soon as the call returns. A nonblocking (immediate) call requires a separate completion call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free to modify 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 the description of the operation). Thus, a collective communication operation may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier 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. The collective operations do not have a message tag argument. 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.

(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.

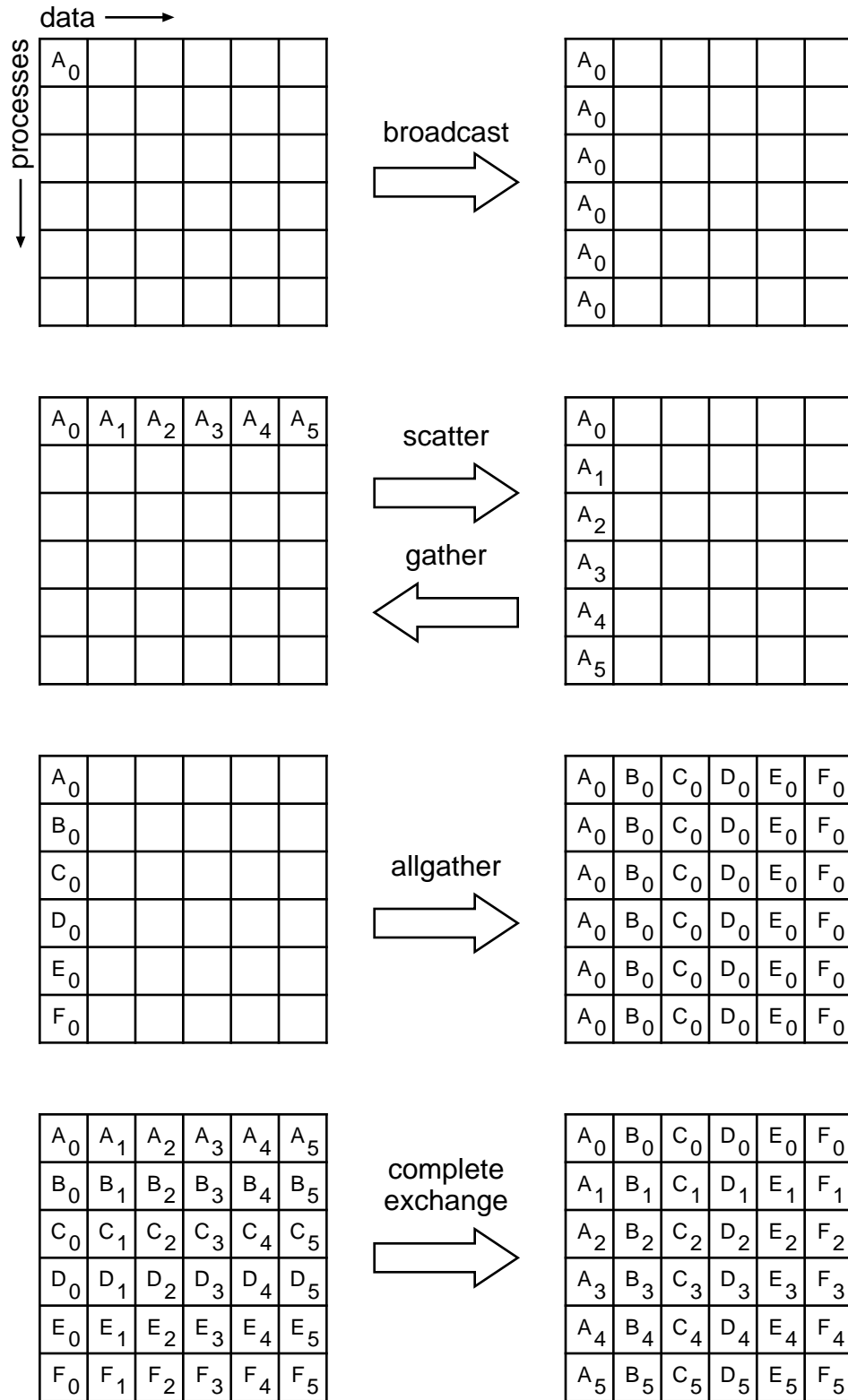


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

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 6. 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 identifier 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.

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. (*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_BLOCK`,
`MPI_IREDUCE_SCATTER_BLOCK`, `MPI_REDUCE_SCATTER`,
`MPI_IREDUCE_SCATTER`
- `MPI_BARRIER`, `MPI_IBARRIER`

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`

The data movement patterns of `MPI_SCAN`, `MPI_ISCAN`, `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_BLOCK, MPI_IREDUCE_SCATTER_BLOCK,
MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER.

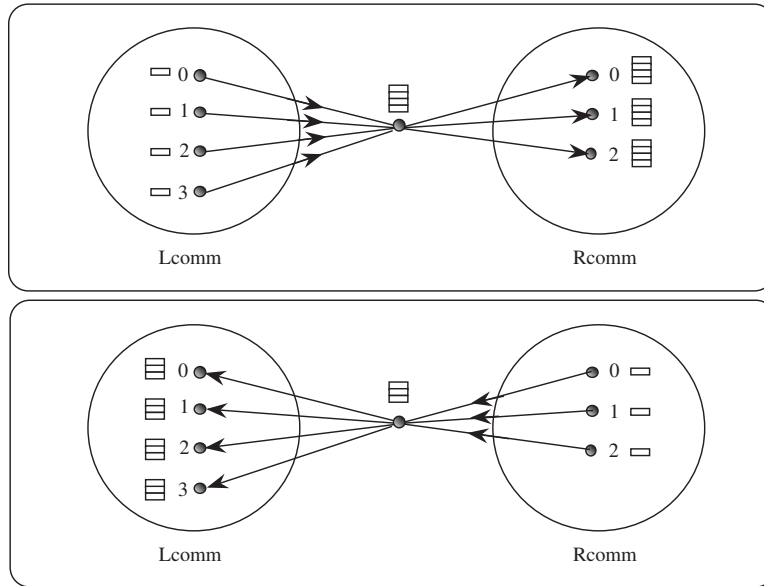


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine.

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.

For intercommunicator collective communication, if the operation is in the All-To-One or One-To-All categories, 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

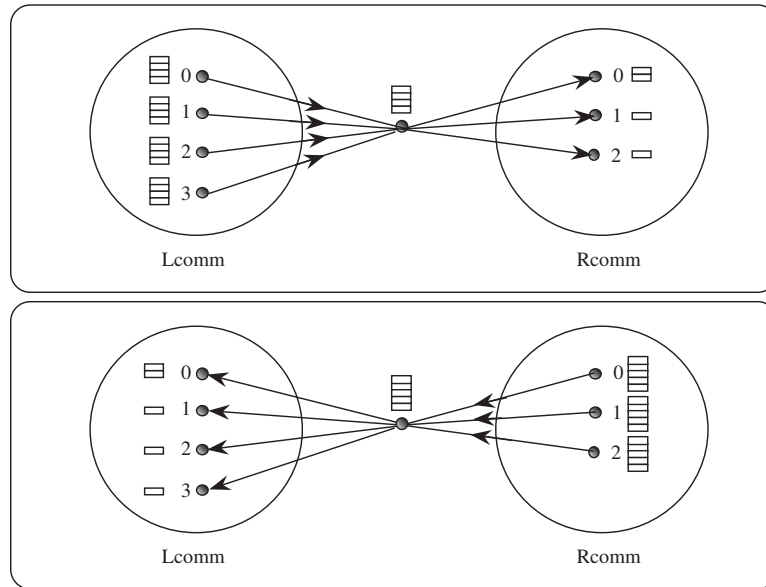


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

and provide the rank of the root. If the operation is in the All-To-All category, then the transfer is bidirectional.

Rationale. Operations in the All-To-One and One-To-All categories are unidirectional by nature, and there is a clear way of specifying direction. Operations in the All-To-All category 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)`

TYPE(MPI_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

`MPI_BARRIER(COMM, IERROR)`

INTEGER COMM, IERROR

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, `MPI_BARRIER` involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the other group (group B) have entered the call (and vice versa). A process may return from the call before all processes in its own group have entered the call.

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(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
```

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

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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount, root
```

```

1      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2      TYPE(MPI_Comm), INTENT(IN) :: comm
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
6                ROOT, COMM, IERROR)
7      <type> SENDBUF(*), RECVBUF(*)
8      INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, IERROR

```

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

`MPI_Send(sendbuf, sendcount, sendtype, root , ...)`, and the root had executed `n` calls to

`MPI_Recv(recvbuf+i·recvcount·extent(recvtype), recvcount, recvtype, i,...)`, where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_get_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)			1
			2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative integer)	4
			5
			6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8
			9
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)	10
			11
			12
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement relative to recvbuf at which to place the incoming data from process <i>i</i> (significant only at root)	13
			14
			15
			16
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	17
			18
			19
IN	root	rank of receiving process (integer)	20
			21
IN	comm	communicator (handle)	22
			23

```

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

```

```

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
            recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

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

```

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

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

MPI_Send(sendbuf, sendcount, sendtype, root, ...), and the root executes *n* receives,

`MPI_Recv(recvbuf+displs[j]·extent(recvtype), recvcnts[j], recvtype, 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 `recvcnts[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.

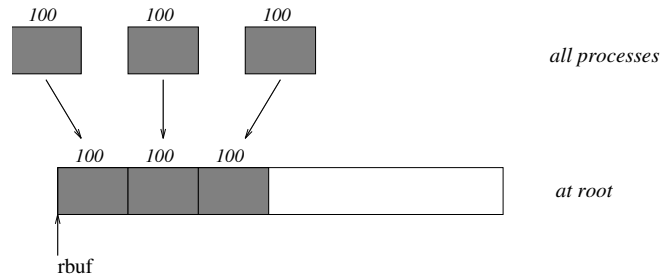


Figure 5.4: The root process gathers 100 ints from each process in the group.

```

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

```

Example 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 $stride \geq 100$. See Figure 5.5.

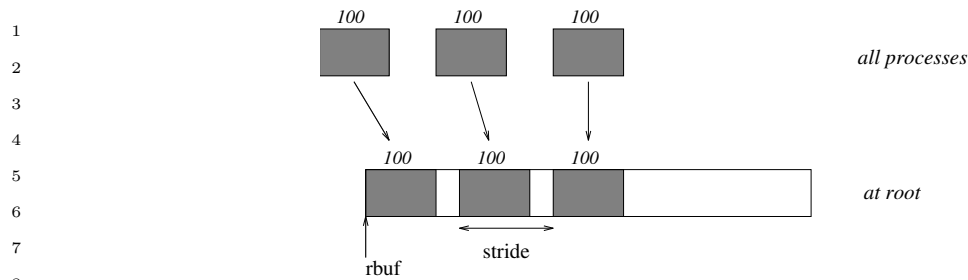


Figure 5.5: The root process gathers 100 ints from each process in the group, each set is placed `stride` ints apart.

```

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

...

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

```

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

Example 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) {

```

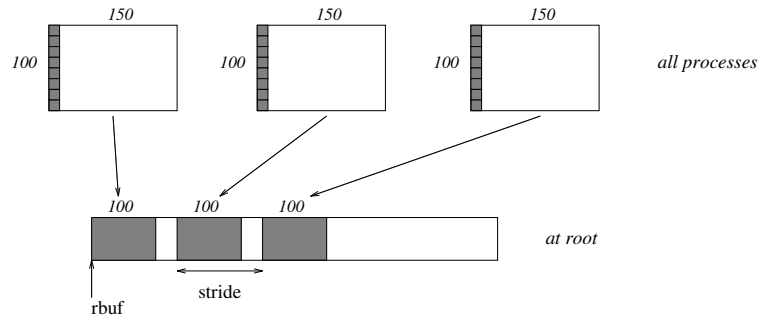



Figure 5.6: The root process gathers column 0 of a 100×150 C array, and each set is placed `stride` ints apart.

```

    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.

```

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);

```

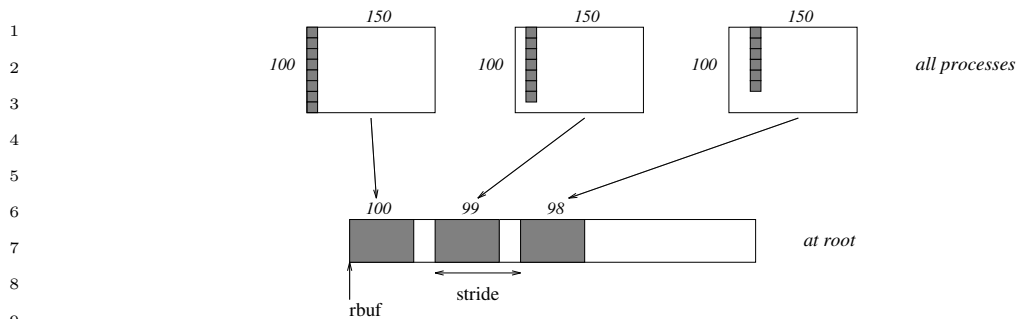


Figure 5.7: The root process gathers $100-i$ ints from column i of a 100×150 C array, and each set is placed `stride` ints apart.

```

/* 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 4.16, Section 4.1.14.

```

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;
}
/* Create datatype for one int, with extent of entire row
 */
MPI_Type_create_resized( MPI_INT, 0, 150*sizeof(int), &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;

...

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

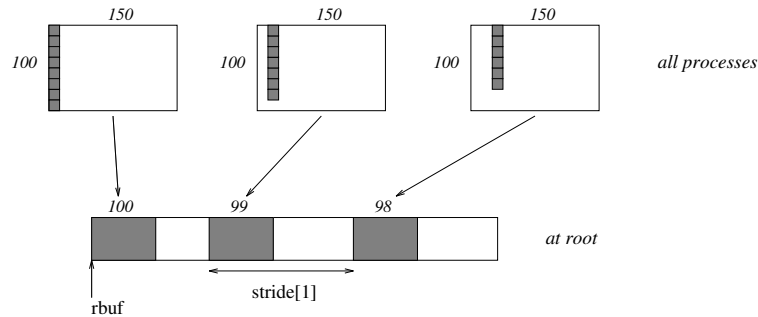


Figure 5.8: The root process gathers $100-i$ ints from column i of a 100×150 C array, and each set is placed $\text{stride}[i]$ ints apart (a varying stride).

Process i sends num ints 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, myrank;
MPI_Datatype stype;
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
 */
MPI_Type_create_resized( MPI_INT, 0, 150*sizeof(int), &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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

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

```

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,

MPI_Send(sendbuf+i·sendcount·extent(sendtype), sendcount, sendtype, i,...), and each process executed a receive,

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

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

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

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

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

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

16
17 *Rationale.* Though not needed, the last restriction is imposed so as to achieve
18 symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write
19 restriction) is necessary. (*End of rationale.*)

20
21 The “in place” option for intracommunicators is specified by passing MPI_IN_PLACE as
22 the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and
23 root “sends” no data to itself. The scattered vector is still assumed to contain n segments,
24 where n is the group size; the root-th segment, which root should “send to itself,” is not
25 moved.

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

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)			1
			2
IN	sendbuf	address of send buffer (choice, significant only at root)	3
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	4
			5
			6
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	7
			8
			9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	address of receive buffer (choice)	11
			12
IN	recvcount	number of elements in receive buffer (non-negative integer)	13
			14
IN	recvtype	data type of receive buffer elements (handle)	15
			16
IN	root	rank of sending process (integer)	17
IN	comm	communicator (handle)	18
			19

```

int MPI_Scatterv(const void* sendbuf, const int sendcounts[], const
                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(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

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

```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

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

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

MPI_Send(sendbuf+displs[i]·extent(sendtype), sendcounts[i], sendtype, i,...), and each process executed a receive,

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

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

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

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 a 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.

```
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* \geq 100. See Figure 5.10.

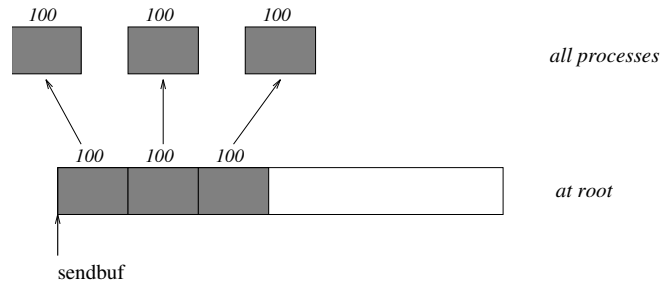


Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

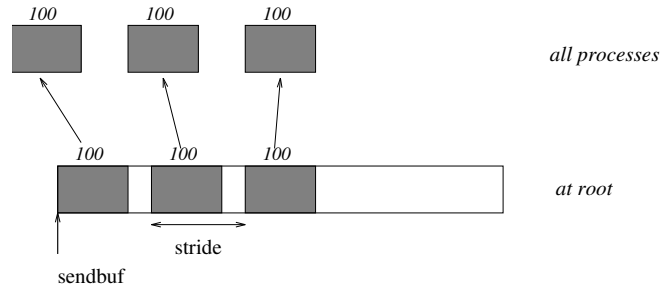


Figure 5.10: The root process scatters sets of 100 ints, moving by `stride` ints from send to send in the scatter.

```

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;
int gsize,recvarray[100][150],*rprr;

```

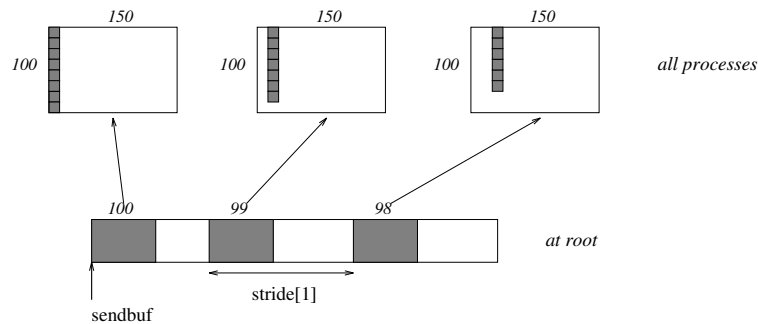


Figure 5.11: The root scatters blocks of $100-i$ ints into column i of a 100×150 C array. At the sending side, the blocks are $\text{stride}[i]$ ints apart.

```

14     int root, *sendbuf, myrank, *stride;
15     MPI_Datatype rtype;
16     int i, *displs, *counts, offset;
17     ...
18     MPI_Comm_size(comm, &gsize);
19     MPI_Comm_rank(comm, &myrank);
20
21     stride = (int *)malloc(gsize*sizeof(int));
22     ...
23     /* stride[i] for i = 0 to gsize-1 is set somehow
24      * sendbuf comes from elsewhere
25      */
26     ...
27     displs = (int *)malloc(gsize*sizeof(int));
28     counts = (int *)malloc(gsize*sizeof(int));
29     offset = 0;
30     for (i=0; i<gsize; ++i) {
31         displs[i] = offset;
32         offset += stride[i];
33         counts[i] = 100 - i;
34     }
35     /* Create datatype for the column we are receiving
36      */
37     MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
38     MPI_Type_commit(&rtype);
39     rptr = &recvarray[0][myrank];
40     MPI_Scatterv(sendbuf, counts, displs, MPI_INT, rptr, 1, rtype,
41                 root, comm);

```

5.7 Gather-to-all

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)			1
IN	sendbuf	starting address of send buffer (choice)	2
IN	sendcount	number of elements in send buffer (non-negative integer)	3
IN	sendtype	data type of send buffer elements (handle)	4
OUT	recvbuf	address of receive buffer (choice)	5
IN	recvcount	number of elements received from any process (non-negative integer)	6
IN	recvtype	data type of receive buffer elements (handle)	7
IN	comm	communicator (handle)	8

```

int MPI_Allgather(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
              COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, IERROR

```

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The 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 of one group (group A) contributes `sendcount` data items; these data are concatenated and the result is stored at each process in the other group (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	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements in send buffer (non-negative integer)
IN	<code>sendtype</code>	data type of send buffer elements (handle)
OUT	<code>recvbuf</code>	address of receive buffer (choice)
IN	<code>recvcounts</code>	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	<code>displs</code>	integer array (of length group size). Entry <code>i</code> specifies the displacement (relative to <code>recvbuf</code>) at which to place the incoming data from process <code>i</code>
IN	<code>recvtype</code>	data type of receive buffer elements (handle)
IN	<code>comm</code>	communicator (handle)

```

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

MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
               recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm

```

```

    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
               RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
IERROR

```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The 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. In such a case, `sendcount` and `sendtype` are ignored, and 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 of one group (group A) contributes `sendcount` data items; these data are concatenated and the result is stored at each process in the other group (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 Example using MPI_ALLGATHER

The example in this section uses 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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The 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,

```
MPI_Send(sendbuf+i·sendcount·extent(sendtype),sendcount,sendtype,i, ...), and a receive
from every other process with a call to,
```

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

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

The “in place” option for intracommunicators is specified by passing `MPI_IN_PLACE` to the argument `sendbuf` at *all* processes. In such a case, `sendcount` and `sendtype` are ignored. The data to be sent is taken from the `recvbuf` and replaced by the received data. Data sent and received must have the same type map as specified by `recvcount` and `recvtype`.

Rationale. For large `MPI_ALLTOALL` instances, allocating both send and receive buffers may consume too much memory. The “in place” option effectively halves the application memory consumption and is useful in situations where the data to be sent will not be used by the sending process after the `MPI_ALLTOALL` exchange (e.g., in parallel Fast Fourier Transforms). (*End of rationale.*)

Advice to implementors. Users may opt to use the “in place” option in order to conserve memory. Quality MPI implementations should thus strive to minimize system buffering. (*End of advice to implementors.*)

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 a complete exchange 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.*)

```

1  MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
2      recvtype, comm)
3      IN      sendbuf      starting address of send buffer (choice)
4      IN      sendcounts   non-negative integer array (of length group size) spec-
5                          ifying the number of elements to send to each rank
6
7      IN      sdispls      integer array (of length group size). Entry j specifies
8                          the displacement (relative to sendbuf) from which to
9                          take the outgoing data destined for process j
10
11     IN      sendtype      data type of send buffer elements (handle)
12     OUT     recvbuf       address of receive buffer (choice)
13     IN      recvcoun-    non-negative integer array (of length group size) spec-
14                     ifying the number of elements that can be received
15                     from each rank
16
17     IN      rdispls      integer array (of length group size). Entry i specifies
18                     the displacement (relative to recvbuf) at which to place
19                     the incoming data from process i
20
21     IN      recvtype     data type of receive buffer elements (handle)
22     IN      comm         communicator (handle)

```

```

23  int MPI_Alltoallv(const void* sendbuf, const int sendcounts[], const
24      int sdispls[], MPI_Datatype sendtype, void* recvbuf, const
25      int recvcoun-    int rdispls[], MPI_Datatype recvtype,
26      MPI_Comm comm)

```

```

27  MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
28      rdispls, recvtype, comm, ierror)
29      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
30      TYPE(*), DIMENSION(..) :: recvbuf
31      INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcoun-
32      rdispls(*)
33      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34      TYPE(MPI_Comm), INTENT(IN) :: comm
35      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

36  MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, REVCOUNTS,
37      RDISPLS, RECVTYPE, COMM, IERROR)
38      <type> SENDBUF(*), RECVBUF(*)
39      INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),
40      RECVTYPE, COMM, IERROR

```

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

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

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

`MPI_Send(sendbuf+sdispls[i]·extent(sendtype),sendcounts[i],sendtype,i,...)`, and received a message from every other process with a call to

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

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

The “in place” option for intracommunicators is specified by passing `MPI_IN_PLACE` to the argument `sendbuf` at *all* processes. In such a case, `sendcounts`, `sdispls` and `sendtype` are ignored. The data to be sent is taken from the `recvbuf` and replaced by the received data. Data sent and received must have the same type map as specified by the `recvcounts` array and the `recvtype`, and is taken from the locations of the receive buffer specified by `rdispls`.

Advice to users. Specifying the “in place” option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that `recvcounts[j]` and `recvtype` on process `i` match `recvcounts[i]` and `recvtype` on process `j`. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the `MPI_ALLTOALLV` exchange. (*End of advice to users.*)

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.*)

```

1  MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls,
2                recvtypes, comm)
3
4  IN      sendbuf      starting address of send buffer (choice)
5
6  IN      sendcounts    non-negative integer array (of length group size) spec-
7                      ifying the number of elements to send to each rank
8
9  IN      sdispls       integer array (of length group size). Entry j specifies
10                      the displacement in bytes (relative to sendbuf) from
11                      which to take the outgoing data destined for process j
12                      (array of integers)
13
14  IN      sendtypes     array of datatypes (of length group size). Entry j spec-
15                      ifies the type of data to send to process j (array of
16                      handles)
17
18  OUT     recvbuf       address of receive buffer (choice)
19
20  IN      recvcounts    non-negative integer array (of length group size) spec-
21                      ifying the number of elements that can be received
22                      from each rank
23
24  IN      rdispls       integer array (of length group size). Entry i specifies
25                      the displacement in bytes (relative to recvbuf) at which
26                      to place the incoming data from process i (array of
27                      integers)
28
29  IN      recvtypes     array of datatypes (of length group size). Entry i spec-
30                      ifies the type of data received from process i (array of
31                      handles)
32
33  IN      comm          communicator (handle)
34
35  int MPI_Alltoallw(const void* sendbuf, const int sendcounts[], const
36                    int sdispls[], const MPI_Datatype sendtypes[], void* recvbuf,
37                    const int recvcounts[], const int rdispls[], const
38                    MPI_Datatype recvtypes[], MPI_Comm comm)
39
40  MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
41                rdispls, recvtypes, comm, ierror)
42
43  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
44  TYPE(*), DIMENSION(..) :: recvbuf
45  INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
46  rdispls(*)
47  TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
48  TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
49  TYPE(MPI_Comm), INTENT(IN) :: comm
50  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
51
52  MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
53                RDISPLS, RECVTYPES, COMM, IERROR)
54
55  <type> SENDBUF(*), RECVBUF(*)
56  INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
57  RDISPLS(*), RECVTYPES(*), COMM, IERROR

```

MPI_ALLTOALLW is the most general form of complete exchange. 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

`MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i],i,...)`, and received a message from every other process with a call to

`MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i],i,...)`.

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

Like for MPI_ALLTOALLV, the “in place” option for intracommunicators is specified by passing MPI_IN_PLACE to the argument `sendbuf` at *all* processes. In such a case, `sendcounts`, `sdispls` and `sendtypes` are ignored. The data to be sent is taken from the `recvbuf` and replaced by the received data. Data sent and received must have the same type map as specified by the `recvcounts` and `recvtypes` arrays, and is taken from the locations of the receive buffer specified by `rdispls`.

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 (for example sum, maximum, and logical and) 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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
    TYPE(*), DIMENSION(..) :: recvbuf
```

```
    INTEGER, INTENT(IN) :: count, root
```

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
    TYPE(MPI_Op), INTENT(IN) :: op
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

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 of the same length, with elements of the same type as the output buffer at the root. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is `MPI_MAX` and the send buffer contains two elements that are floating point numbers (`count = 2` and `datatype = MPI_FLOAT`), then `recvbuf(1) = global max(sendbuf(1))` and `recvbuf(2) = global max(sendbuf(2))`.

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

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 ranks. (*End of advice to implementors.*)

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 5.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with `MPI_GATHER`), applying the reduction operation in the desired order (e.g., with `MPI_REDUCE_LOCAL`), and if needed, broadcast or scatter the result to the other processes (e.g., with `MPI_BCAST`). (*End of advice to users.*)

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. It is safest 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 a 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_BLOCK`, `MPI_REDUCE_SCATTER`, `MPI_SCAN`, `MPI_EXSCAN`, all nonblocking variants of those (see Section 5.12), and `MPI_REDUCE_LOCAL`. These operations are invoked by placing the following in `op`.

Name	Meaning
<code>MPI_MAX</code>	maximum
<code>MPI_MIN</code>	minimum
<code>MPI_SUM</code>	sum
<code>MPI_PROD</code>	product
<code>MPI_LAND</code>	logical and
<code>MPI_BAND</code>	bit-wise and
<code>MPI_LOR</code>	logical or
<code>MPI BOR</code>	bit-wise or
<code>MPI_LXOR</code>	logical exclusive or (xor)
<code>MPI_BXOR</code>	bit-wise exclusive or (xor)
<code>MPI_MAXLOC</code>	max value and location
<code>MPI_MINLOC</code>	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:	<code>MPI_INT</code> , <code>MPI_LONG</code> , <code>MPI_SHORT</code> , <code>MPI_UNSIGNED_SHORT</code> , <code>MPI_UNSIGNED</code> , <code>MPI_UNSIGNED_LONG</code> , <code>MPI_LONG_LONG_INT</code> , <code>MPI_LONG_LONG</code> (as synonym), <code>MPI_UNSIGNED_LONG_LONG</code> , <code>MPI_SIGNED_CHAR</code> , <code>MPI_UNSIGNED_CHAR</code> , <code>MPI_INT8_T</code> , <code>MPI_INT16_T</code> , <code>MPI_INT32_T</code> , <code>MPI_INT64_T</code> , <code>MPI_UINT8_T</code> , <code>MPI_UINT16_T</code> , <code>MPI_UINT32_T</code> , <code>MPI_UINT64_T</code>
Fortran integer:	<code>MPI_INTEGER</code> , and handles returned from <code>MPI_TYPE_CREATE_F90_INTEGER</code> , and if available: <code>MPI_INTEGER1</code> , <code>MPI_INTEGER2</code> , <code>MPI_INTEGER4</code> , <code>MPI_INTEGER8</code> , <code>MPI_INTEGER16</code>
Floating point:	<code>MPI_FLOAT</code> , <code>MPI_DOUBLE</code> , <code>MPI_REAL</code> , <code>MPI_DOUBLE_PRECISION</code> <code>MPI_LONG_DOUBLE</code> and handles returned from

	MPI_TYPE_CREATE_F90_REAL,	1
	and if available: MPI_REAL2,	2
	MPI_REAL4, MPI_REAL8, MPI_REAL16	3
Logical:	MPI_LOGICAL, MPI_C_BOOL,	4
	MPI_CXX_BOOL	5
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	6
	MPI_C_FLOAT_COMPLEX (as synonym),	7
	MPI_C_DOUBLE_COMPLEX,	8
	MPI_C_LONG_DOUBLE_COMPLEX,	9
	MPI_CXX_FLOAT_COMPLEX,	10
	MPI_CXX_DOUBLE_COMPLEX,	11
	MPI_CXX_LONG_DOUBLE_COMPLEX,	12
	and handles returned from	13
	MPI_TYPE_CREATE_F90_COMPLEX,	14
	and if available: MPI_DOUBLE_COMPLEX,	15
	MPI_COMPLEX4, MPI_COMPLEX8,	16
	MPI_COMPLEX16, MPI_COMPLEX32	17
Byte:	MPI_BYTE	18
Multi-language types:	MPI_AINT, MPI_OFFSET, MPI_COUNT	19

Now, the valid datatypes for each operation are specified below.

Op	Allowed Types	22
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point, Multi-language types	23
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex, Multi-language types	24
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	25
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte, Multi-language types	26

These operations together with all listed datatypes are valid in all supported programming languages, see also Reduce Operations on page 656 in Section 17.2.6.

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.

```

1  SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
2  REAL a(m), b(m)          ! local slice of array
3  REAL c                    ! result (at node zero)
4  REAL sum
5  INTEGER m, comm, i, ierr
6
7  ! local sum
8  sum = 0.0
9  DO i = 1, m
10     sum = sum + a(i)*b(i)
11 END DO
12
13 ! global sum
14 CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
15 RETURN
16 END

```

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.

```

22 SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
23 REAL a(m), b(m,n)      ! local slice of array
24 REAL c(n)              ! result
25 REAL sum(n)
26 INTEGER n, comm, i, j, ierr
27
28 ! local sum
29 DO j= 1, n
30     sum(j) = 0.0
31     DO i = 1, m
32         sum(j) = sum(j) + a(i)*b(i,j)
33     END DO
34 END DO
35
36 ! global sum
37 CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
38
39 ! return result at node zero (and garbage at the other nodes)
40 RETURN
41 END

```

5.9.3 Signed Characters and Reductions

The types `MPI_SIGNED_CHAR` and `MPI_UNSIGNED_CHAR` can be used in reduction operations. `MPI_CHAR`, `MPI_WCHAR`, and `MPI_CHARACTER` (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, `MPI_CHAR`, `MPI_WCHAR`, and `MPI_CHARACTER` 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`, `MPI_WCHAR`, 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 $(u_0, 0), (u_1, 1), \dots, (u_{n-1}, n-1)$, 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
<code>MPI_2REAL</code>	pair of <code>REAL</code> s
<code>MPI_2DOUBLE_PRECISION</code>	pair of <code>DOUBLE PRECISION</code> variables
<code>MPI_2INTEGER</code>	pair of <code>INTEGER</code> s

C:

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

The datatype `MPI_2REAL` is *as if* defined by the following (see Section 4.1).

```
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_CREATE_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)

```

```

1      ! At this point, the answer resides on process root
2
3      IF (myrank .EQ. root) THEN
4          ! read ranks out
5          DO I= 1, 30
6              aout(i) = out(1,i)
7              ind(i) = out(2,i) ! rank is coerced back to an integer
8          END DO
9      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.

```

15  #define  LEN    1000
16
17  float val[LEN];          /* local array of values */
18  int count;               /* local number of values */
19  int myrank, minrank, minindex;
20  float minval;
21
22  struct {
23      float value;
24      int    index;
25  } in, out;
26
27      /* local minloc */
28  in.value = val[0];
29  in.index = 0;
30  for (i=1; i < count; i++)
31      if (in.value > val[i]) {
32          in.value = val[i];
33          in.index = i;
34      }
35
36      /* global minloc */
37  MPI_Comm_rank(comm, &myrank);
38  in.index = myrank*LEN + in.index;
39  MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
40      /* At this point, the answer resides on process root
41      */
42  if (myrank == root) {
43      /* read answer out
44      */
45      minval = out.value;
46      minrank = out.index / LEN;
47      minindex = out.index % LEN;
48  }

```

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(user_fn, commute, op)

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

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

```
MPI_Op_create(user_fn, commute, op, ierror)
  PROCEDURE(MPI_User_function) :: user_fn
  LOGICAL, INTENT(IN) :: commute
  TYPE(MPI_Op), INTENT(OUT) :: op
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

MPI_OP_CREATE binds a user-defined reduction operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and MPI_REDUCE_LOCAL. 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, taking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument user_fn 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 declarations of the user-defined function user_fn appear below.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
```

```

1      TYPE(C_PTR), VALUE :: invec, inoutvec
2      INTEGER :: len
3      TYPE(MPI_Datatype) :: datatype
4
5      SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
6          <type> INVEC(LEN), INOUTVEC(LEN)
7          INTEGER LEN, 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 $w[i] = u[i] \circ v[i]$, for $i=0, \dots, len-1$, where \circ is the reduce operation that the function computes.

Informally, we can think of `invec` and `inoutvec` as arrays of `len` elements that `user_fn` 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 $i=0, \dots, count-1$, 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.


```
1     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
2 MPI_OP_FREE(OP, IERROR)
```

```
3     INTEGER OP, IERROR
```

```
4
5     Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
```

7 Example of User-defined Reduce

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

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

```
12
13 typedef struct {
14     double real,imag;
15 } Complex;
16
17 /* the user-defined function
18 */
19 void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
20 {
21     int i;
22     Complex c;
23     Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
24
25     for (i=0; i< *len; ++i) {
26         c.real = inout->real*in->real -
27             inout->imag*in->imag;
28         c.imag = inout->real*in->imag +
29             inout->imag*in->real;
30         *inout = c;
31         in++; inout++;
32     }
33 }
34
35 /* and, to call it...
36 */
37 ...
38
39 /* each process has an array of 100 Complexes
40 */
41 Complex a[100], answer[100];
42 MPI_Op myOp;
43 MPI_Datatype ctype;
44
45 /* explain to MPI how type Complex is defined
46 */
47 MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
48 MPI_Type_commit(&ctype);
```



```

/* create the complex-product user-op
*/
MPI_Op_create( myProd, 1, &myOp );

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

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

```

Example 5.21 How to use the `mpi_f08` interface of the Fortran `MPI_User_function`.

```

subroutine my_user_function( invec, inoutvec, len, type )
  use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
  use mpi_f08
  type(c_ptr), value :: invec, inoutvec
  integer :: len
  type(MPI_Datatype) :: type
  real, pointer :: invec_r(:), inoutvec_r(:)
  if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
    call c_f_pointer(invec, invec_r, (/ len /) )
    call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
    inoutvec_r = invec_r + inoutvec_r
  end if
end subroutine

```

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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf

```

```

1      TYPE(*), DIMENSION(..) :: recvbuf
2      INTEGER, INTENT(IN) :: count
3      TYPE(MPI_Datatype), INTENT(IN) :: datatype
4      TYPE(MPI_Op), INTENT(IN) :: op
5      TYPE(MPI_Comm), INTENT(IN) :: comm
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8      MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
9      <type> SENDBUF(*), RECVBUF(*)
10     INTEGER COUNT, DATATYPE, OP, COMM, IERROR

```

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 5.22

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).

```

29      SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
30      REAL a(m), b(m,n)      ! local slice of array
31      REAL c(n)              ! result
32      REAL sum(n)
33      INTEGER n, comm, i, j, ierr
34
35      ! local sum
36      DO j= 1, n
37         sum(j) = 0.0
38         DO i = 1, m
39            sum(j) = sum(j) + a(i)*b(i,j)
40         END DO
41      END DO
42
43      ! global sum
44      CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
45
46      ! return result at all nodes
47      RETURN
48      END

```

5.9.7 Process-Local Reduction

The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.

The following function applies a reduction operator to local arguments.

MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)

IN	inbuf	input buffer (choice)
INOUT	inoutbuf	combined input and output buffer (choice)
IN	count	number of elements in inbuf and inoutbuf buffers (non-negative integer)
IN	datatype	data type of elements of inbuf and inoutbuf buffers (handle)
IN	op	operation (handle)

```
int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
                    MPI_Datatype datatype, MPI_Op op)
```

```
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
TYPE(*), DIMENSION(..) :: inoutbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
```

```
<type> INBUF(*), INOUTBUF(*)
INTEGER COUNT, DATATYPE, OP, IERROR
```

The function applies the operation given by **op** element-wise to the elements of **inbuf** and **inoutbuf** with the result stored element-wise in **inoutbuf**, as explained for user-defined operations in Section 5.9.5. Both **inbuf** and **inoutbuf** (input as well as result) have the same number of elements given by **count** and the same datatype given by **datatype**. The **MPI_IN_PLACE** option is not allowed.

Reduction operations can be queried for their commutativity.

MPI_OP_COMMUTATIVE(op, commute)

IN	op	operation (handle)
OUT	commute	true if op is commutative, false otherwise (logical)

```
int MPI_Op_commutative(MPI_Op op, int *commute)
```

```
MPI_Op_commutative(op, commute, ierror)
```

```
TYPE(MPI_Op), INTENT(IN) :: op
```

```

1      LOGICAL, INTENT(OUT) :: commute
2      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4  MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
5      LOGICAL COMMUTE
6      INTEGER OP, IERROR

```

5.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

5.10.1 MPI_REDUCE_SCATTER_BLOCK

```

17 MPI_REDUCE_SCATTER_BLOCK( sendbuf, recvbuf, recvcount, datatype, op, comm)
18
19     IN      sendbuf      starting address of send buffer (choice)
20     OUT     recvbuf      starting address of receive buffer (choice)
21     IN      recvcount    element count per block (non-negative integer)
22     IN      datatype      data type of elements of send and receive buffers (handle)
23
24     IN      op            operation (handle)
25     IN      comm          communicator (handle)
26
27
28
29 int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
30                             int recvcount, MPI_Datatype datatype, MPI_Op op,
31                             MPI_Comm comm)
32
33 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
34                           ierror)
35     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
36     TYPE(*), DIMENSION(..) :: recvbuf
37     INTEGER, INTENT(IN) :: recvcount
38     TYPE(MPI_Datatype), INTENT(IN) :: datatype
39     TYPE(MPI_Op), INTENT(IN) :: op
40     TYPE(MPI_Comm), INTENT(IN) :: comm
41     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43 MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
44                           IERROR)
45     <type> SENDBUF(*), RECVBUF(*)
46     INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR

```

If `comm` is an intracommunicator, `MPI_REDUCE_SCATTER_BLOCK` first performs a global, element-wise reduction on vectors of count = `n*recvcount` elements in the send buffers

defined by `sendbuf`, `count` and `datatype`, using the operation `op`, where `n` is the number of processes in the group of `comm`. The routine is called by all group members using the same arguments for `recvcount`, `datatype`, `op` and `comm`. The resulting vector is treated as `n` consecutive blocks of `recvcount` elements that are scattered to the processes of the group. The `i`-th block is sent to process `i` and stored in the receive buffer defined by `recvbuf`, `recvcount`, and `datatype`.

Advice to implementors. The `MPI_REDUCE_SCATTER_BLOCK` routine is functionally equivalent to: an `MPI_REDUCE` collective operation with `count` equal to `recvcount*n`, followed by an `MPI_SCATTER` with `sendcount` equal to `recvcount`. 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 on *all* processes. In this case, the input data is taken from the receive buffer.

If `comm` is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the `recvcount` argument, and provide input vectors of `count = n*recvcount` elements stored in the send buffers, where `n` is the size of the group. The number of elements `count` must be the same for the two groups. The resulting vector from the other group is scattered in blocks of `recvcount` elements among the processes in the group.

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

5.10.2 MPI_REDUCE_SCATTER

`MPI_REDUCE_SCATTER` extends the functionality of `MPI_REDUCE_SCATTER_BLOCK` such that the scattered blocks can vary in size. Block sizes are determined by the `recvcounts` array, such that the `i`-th block contains `recvcounts[i]` elements.

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

IN	<code>sendbuf</code>	starting address of send buffer (choice)
OUT	<code>recvbuf</code>	starting address of receive buffer (choice)
IN	<code>recvcounts</code>	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.
IN	<code>datatype</code>	data type of elements of send and receive buffers (handle)
IN	<code>op</code>	operation (handle)
IN	<code>comm</code>	communicator (handle)

```

1  int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const
2      int recvcnts[], MPI_Datatype datatype, MPI_Op op,
3      MPI_Comm comm)
4
5  MPI_Reduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm,
6      ierror)
7      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
8      TYPE(*), DIMENSION(..) :: recvbuf
9      INTEGER, INTENT(IN) :: recvcnts(*)
10     TYPE(MPI_Datatype), INTENT(IN) :: datatype
11     TYPE(MPI_Op), INTENT(IN) :: op
12     TYPE(MPI_Comm), INTENT(IN) :: comm
13     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15 MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
16     IERROR)
17     <type> SENDBUF(*), RECVBUF(*)
18     INTEGER REVCOUNTS(*), DATATYPE, OP, COMM, IERROR

```

If `comm` is an intracommunicator, `MPI_REDUCE_SCATTER` first performs a global, element-wise reduction on vectors of $\text{count} = \sum_{i=0}^{n-1} \text{recvcnts}[i]$ elements in the send buffers defined by `sendbuf`, `count` and `datatype`, using the operation `op`, where `n` is the number of processes in the group of `comm`. The routine is called by all group members using the same arguments for `recvcnts`, `datatype`, `op` and `comm`. The resulting vector is treated as `n` consecutive blocks where the number of elements of the `i`-th block is `recvcnts[i]`. The blocks are scattered to the processes of the group. The `i`-th block is sent to process `i` and stored in the receive buffer defined by `recvbuf`, `recvcnts[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 `recvcnts[i]` followed by `MPI_SCATTERV` with `sendcounts` equal to `recvcnts`. 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 receive buffer. It is not required to specify the “in place” option on all processes, since the processes for which `recvcnts[i] == 0` may not have allocated a receive buffer.

If `comm` is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same `recvcnts` argument, and provide input vectors of $\text{count} = \sum_{i=0}^{n-1} \text{recvcnts}[i]$ elements stored in the send buffers, where `n` is the size of the group. The resulting vector from the other group is scattered in blocks of `recvcnts[i]` elements among the processes in the group. The number of elements `count` must 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 `recvcnts` 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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
    TYPE(*), DIMENSION(..) :: recvbuf
```

```
    INTEGER, INTENT(IN) :: count
```

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
    TYPE(MPI_Op), INTENT(IN) :: op
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

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 routine is called by all group members using the same arguments for `count`, `datatype`, `op` and `comm`, except that for user-defined operations, the same rules apply as for `MPI_REDUCE`. 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(const 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(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
    TYPE(*), DIMENSION(..) :: recvbuf
```

```
    INTEGER, INTENT(IN) :: count
```

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
    TYPE(MPI_Op), INTENT(IN) :: op
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

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 significant 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 $0, \dots, i-1$ (inclusive). The routine is called by all group members using the same arguments for `count`, `datatype`, `op` and `comm`, except that for user-defined operations, the same rules apply as for `MPI_REDUCE`. 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. The receive buffer on rank 0 is not changed by this operation.

This operation is invalid for intercommunicators.

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. (*End of rationale.*)

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.23

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:

<i>values</i>	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
<i>logicals</i>	0	0	1	1	1	0	0	1
<i>result</i>	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.

```

1      int i,base;
2      SegScanPair  a, answer;
3      MPI_Op      myOp;
4      MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
5      MPI_Aint     disp[2];
6      int         blocklen[2] = { 1, 1};
7      MPI_Datatype sspair;
8
9      /* explain to MPI how type SegScanPair is defined
10     */
11     MPI_Get_address( &a, disp);
12     MPI_Get_address( &a.log, disp+1);
13     base = disp[0];
14     for (i=0; i<2; ++i) disp[i] -= base;
15     MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
16     MPI_Type_commit( &sspair );
17     /* create the segmented-scan user-op
18     */
19     MPI_Op_create(segScan, 0, &myOp);
20     ...
21     MPI_Scan( &a, &answer, 1, sspair, myOp, comm );
22
23
24
25
26

```

5.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by overlapping communication and computation, and many systems enable this. Nonblocking collective operations combine the potential benefits of nonblocking point-to-point operations, to exploit overlap and to avoid synchronization, with the optimized implementation and message scheduling provided by collective operations [1, 4]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative mechanism that often leads to better performance (e.g., avoids context switching, scheduler overheads, and thread management) is to use nonblocking collective communication [2].

The nonblocking collective communication model is similar to the model used for nonblocking point-to-point communication. A nonblocking call initiates a collective operation, which must be completed in a separate completion call. Once initiated, the operation may progress independently of any computation or other communication at participating processes. In this manner, nonblocking collective operations can mitigate possible synchronizing effects of collective operations by running them in the “background.” In addition to enabling communication-computation overlap, nonblocking collective operations can perform collective operations on overlapping communicators, which would lead to deadlocks with blocking operations. Their semantic advantages can also be useful in combination with point-to-point communication.

As in the nonblocking point-to-point case, all calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation, which indicates

that the system may start to copy data out of the send buffer and into the receive buffer. Once initiated, all associated send buffers and buffers associated with input arguments (such as arrays of counts, displacements, or datatypes in the vector versions of the collectives) should not be modified, and all associated receive buffers should not be accessed, until the collective operation completes. The call returns a request handle, which must be passed to a completion call.

All completion calls (e.g., `MPI_WAIT`) described in Section 3.7.3 are supported for nonblocking collective operations. Similarly to the blocking case, nonblocking collective operations are considered to be complete when the local part of the operation is finished, i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be safely accessed and modified. Completion does not indicate that other processes have completed or even started the operation (unless otherwise implied by the description of the operation). Completion of a particular nonblocking collective operation also does not indicate completion of any other posted nonblocking collective (or send-receive) operations, whether they are posted before or after the completed operation.

Advice to users. Users should be aware that implementations are allowed, but not required (with exception of `MPI_IBARRIER`), to synchronize processes during the completion of a nonblocking collective operation. (*End of advice to users.*)

Upon returning from a completion call in which a nonblocking collective operation completes, the `MPI_ERROR` field in the associated status object is set appropriately, see Section 3.2.5 on page 30. The values of the `MPI_SOURCE` and `MPI_TAG` fields are undefined. It is valid to mix 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 (e.g., `MPI_WAITALL`). It is erroneous to call `MPI_REQUEST_FREE` or `MPI_CANCEL` for a request associated with a nonblocking collective operation. Nonblocking collective requests are not persistent.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (*End of rationale.*)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and generate an MPI exception. 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.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with the ordering rules for blocking collective operations in threaded environments.

Rationale. Matching blocking and nonblocking collective operations is not allowed because the implementation might use different communication algorithms for the two

cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (*End of rationale.*)

Advice to users. If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (*End of advice to users.*)

In terms of data movements, each nonblocking collective operation has the same effect as its blocking counterpart for intracommunicators and intercommunicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the “in place” option is allowed exactly as described for the corresponding blocking collective operations. When using the “in place” option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

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

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)`

TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

`MPI_IBARRIER(COMM, REQUEST, IERROR)`

INTEGER COMM, REQUEST, IERROR

`MPI_IBARRIER` is a nonblocking version of `MPI_BARRIER`. By calling `MPI_IBARRIER`, a process notifies that it has reached the barrier. The call returns immediately, independent of whether other processes have called `MPI_IBARRIER`. The usual barrier semantics

are enforced at the corresponding completion operation (test or wait), which in the intra-communicator case will complete only after all other processes in the communicator have called `MPI_IBARRIER`. In the intercommunicator case, it will complete when all processes in the remote group have 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 collective operations 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(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

This call starts a nonblocking variant of `MPI_BCAST` (see Section 5.4).

Example using `MPI_IBCAST`

The example in this section uses an intracommunicator.

Example 5.24

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

```

1      MPI_Comm comm;
2      int array1[100], array2[100];
3      int root=0;
4      MPI_Request req;
5      ...
6      MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
7      compute(array2, 100);
8      MPI_Wait(&req, MPI_STATUS_IGNORE);

```

5.12.3 Nonblocking Gather

```

13 MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
14             request)
15
16     IN      sendbuf      starting address of send buffer (choice)
17     IN      sendcount    number of elements in send buffer (non-negative inte-
18                       ger)
19     IN      sendtype     data type of send buffer elements (handle)
20     OUT     recvbuf      address of receive buffer (choice, significant only at
21                       root)
22
23     IN      recvcount    number of elements for any single receive (non-negative
24                       integer, significant only at root)
25     IN      recvtype     data type of recv buffer elements (significant only at
26                       root) (handle)
27
28     IN      root         rank of receiving process (integer)
29     IN      comm         communicator (handle)
30     OUT     request      communication request (handle)
31
32 int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
33               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
34               MPI_Comm comm, MPI_Request *request)
35
36 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
37             root, comm, request, ierror)
38     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
39     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
40     INTEGER, INTENT(IN) :: sendcount, recvcount, root
41     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
42     TYPE(MPI_Comm), INTENT(IN) :: comm
43     TYPE(MPI_Request), INTENT(OUT) :: request
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46 MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
47             ROOT, COMM, REQUEST, IERROR)
48     <type> SENDBUF(*), RECVBUF(*)

```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, ROOT, COMM, REQUEST,
IERROR

This call starts a nonblocking variant of MPI_GATHER (see Section 5.5).

MPI_IGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs, recvmype, 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	recvcunts	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 <code>recvbuf</code> at which to place the incoming data from process i (significant only at root)
IN	recvmype	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(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, const int recvcunts[], const int displs[],
                MPI_Datatype recvmype, int root, MPI_Comm comm,
                MPI_Request *request)
```

```
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs,
              recvmype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcunts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvmype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

```

1      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVMODE, ROOT,
2      COMM, REQUEST, IERROR

```

This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).

5.12.4 Nonblocking Scatter

```

9      MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
10                  request)
11      IN          sendbuf          address of send buffer (choice, significant only at root)
12      IN          sendcount        number of elements sent to each process (non-negative
13                                  integer, significant only at root)
14
15      IN          sendtype         data type of send buffer elements (significant only at
16                                  root) (handle)
17
18      OUT         recvbuf          address of receive buffer (choice)
19      IN          recvcount        number of elements in receive buffer (non-negative in-
20                                  teger)
21      IN          recvtype         data type of receive buffer elements (handle)
22      IN          root             rank of sending process (integer)
23      IN          comm            communicator (handle)
24      OUT         request          communication request (handle)

```

```

27      int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
28                      void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
29                      MPI_Comm comm, MPI_Request *request)

```

```

31      MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
32                  root, comm, request, ierror)
33      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
35      INTEGER, INTENT(IN) :: sendcount, recvcount, root
36      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37      TYPE(MPI_Comm), INTENT(IN) :: comm
38      TYPE(MPI_Request), INTENT(OUT) :: request
39      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

40      MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVMODE,
41                  ROOT, COMM, REQUEST, IERROR)
42      <type> SENDBUF(*), RECVBUF(*)
43      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMODE, ROOT, COMM, REQUEST,
44      IERROR

```

This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).

MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, request)			1
			2
IN	sendbuf	address of send buffer (choice, significant only at root)	3
			4
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	5
			6
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to <code>sendbuf</code>) from which to take the outgoing data to process i	7
			8
			9
IN	sendtype	data type of send buffer elements (handle)	10
			11
OUT	recvbuf	address of receive buffer (choice)	12
			13
IN	recvcount	number of elements in receive buffer (non-negative integer)	14
			15
IN	recvtype	data type of receive buffer elements (handle)	16
			17
IN	root	rank of sending process (integer)	18
			19
IN	comm	communicator (handle)	20
OUT	request	communication request (handle)	21

```

int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const
    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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

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

```

This call starts a nonblocking variant of MPI_SCATTERV (see Section 5.6).

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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).

```

MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcoun- ts, displs, recvtype, comm, request)			1
			2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	4
			5
			6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice)	8
IN	recvcoun- ts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process	9
			10
			11
			12
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to <code>recvbuf</code>) at which to place the incoming data from process i	13
			14
			15
IN	recvtype	data type of receive buffer elements (handle)	16
IN	comm	communicator (handle)	17
OUT	request	communication request (handle)	18
			19
			20

```

int MPI_Iallgatherv(const void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf, const int recvcoun-
    ts[], MPI_Datatype recvtype, MPI_Comm comm,
    MPI_Request* request)

```

```

MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcoun-
    ts, displs, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcoun-
    ts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECV-
    COUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    REQUEST, IERROR

```

This call starts a nonblocking variant of `MPI_ALLGATHERV` (see Section 5.7).

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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

```

This call starts a nonblocking variant of `MPI_ALLTOALL` (see Section 5.8).

```

MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun- 1
                ts, rdispls, recvtype, comm, request)                                2

IN      sendbuf      starting address of send buffer (choice)                    3
IN      sendcounts    non-negative integer array (of length group size) spec-    4
                        ifying the number of elements to send to each rank        5
IN      sdispls       integer array (of length group size). Entry j specifies    6
                        the displacement (relative to sendbuf) from which to      7
                        take the outgoing data destined for process j             8
IN      sendtype      data type of send buffer elements (handle)                9
OUT     recvbuf       address of receive buffer (choice)                       10
IN      recvcoun-     non-negative integer array (of length group size) spec-   11
                ts     ifying the number of elements that can be received       12
                        from each rank                                           13
IN      rdispls       integer array (of length group size). Entry i specifies   14
                        the displacement (relative to recvbuf) at which to place  15
                        the incoming data from process i                        16
IN      recvtype      data type of receive buffer elements (handle)            17
IN      comm          communicator (handle)                                    18
OUT     request       communication request (handle)                          19
                                                    20
int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const          21
                  int sdispls[], MPI_Datatype sendtype, void* recvbuf, const    22
                  int recvcoun-                                                23
                ts[], const int rdispls[], MPI_Datatype recvtype,              24
                  MPI_Comm comm, MPI_Request *request)                        25
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-     26
                ts, rdispls, recvtype, comm, request, ierror)                 27
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf                  28
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf                             29
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),              30
recvcoun-                                                    31
ts(*), rdispls(*)                                           32
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype                    33
TYPE(MPI_Comm), INTENT(IN) :: comm                                       34
TYPE(MPI_Request), INTENT(OUT) :: request                                35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror                                36
                                                    37
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 38
               RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)                  39
<type> SENDBUF(*), RECVBUF(*)                                           40
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),    41
RECVTYPE, COMM, REQUEST, IERROR                                         42
This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 5.8).  43

```

```

1  MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcoun-
2                    recvtypes, comm, request)
3
4      IN      sendbuf      starting address of send buffer (choice)
5
6      IN      sendcounts   integer array (of length group size) specifying the num-
7                          ber of elements to send to each rank (array of non-
8                          negative integers)
9
10     IN      sdispls      integer array (of length group size). Entry j specifies
11                          the displacement in bytes (relative to sendbuf) from
12                          which to take the outgoing data destined for process j
13                          (array of integers)
14
15     IN      sendtypes    array of datatypes (of length group size). Entry j spec-
16                          ifies the type of data to send to process j (array of
17                          handles)
18
19     OUT     recvbuf      address of receive buffer (choice)
20
21     IN      recvcoun-    integer array (of length group size) specifying the num-
22                          ber of elements that can be received from each rank
23                          (array of non-negative integers)
24
25     IN      rdispls      integer array (of length group size). Entry i specifies
26                          the displacement in bytes (relative to recvbuf) at which
27                          to place the incoming data from process i (array of
28                          integers)
29
30     IN      recvtypes    array of datatypes (of length group size). Entry i spec-
31                          ifies the type of data received from process i (array of
32                          handles)
33
34     IN      comm         communicator (handle)
35
36     OUT     request      communication request (handle)
37
38
39  int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], const
40                    int sdispls[], const MPI_Datatype sendtypes[], void* recvbuf,
41                    const int recvcoun-
42                    MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)
43
44  MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
45                recvcoun-
46                recvtypes, comm, request, ierror)
47
48  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
49  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
50  INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
51  recvcoun-
52  recvtypes(*)
53  TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
54  recvtypes(*)
55  TYPE(MPI_Comm), INTENT(IN) :: comm
56  TYPE(MPI_Request), INTENT(OUT) :: request
57  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
58
59  MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
60                RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)

```

```

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

```

This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 5.8).

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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
            IERROR)

```

```

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

```

This call starts a nonblocking variant of MPI_REDUCE (see Section 5.9.1).

Advice to implementors. The implementation is explicitly allowed to use different algorithms for blocking and nonblocking reduction operations that might change the order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly recommended that MPI_IREDUCE 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 processes. (*End of advice to implementors.*)

Advice to users. For operations which are not truly associative, the result delivered upon completion of the nonblocking reduction may not exactly equal the result delivered by the blocking reduction, even when specifying the same arguments in the same order. (*End of advice to users.*)

5.12.8 Nonblocking All-Reduce

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

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

```
int MPI_Iallreduce(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
               IERROR)
```

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

This call starts a nonblocking variant of `MPI_ALLREDUCE` (see Section 5.9.6).

5.12.9 Nonblocking Reduce-Scatter with Equal Blocks

`MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcnt, datatype, op, comm, request)`

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnt	element count per block (non-negative integer)
IN	datatype	data type of elements of send and receive buffers (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,
                             int recvcnt, MPI_Datatype datatype, MPI_Op op,
                             MPI_Comm comm, MPI_Request *request)
```

```
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcnt, datatype, op, comm,
                           request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: recvcnt
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, REVCOUNT, DATATYPE, OP, COMM,
                           REQUEST, IERROR)
```

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

This call starts a nonblocking variant of `MPI_REDUCE_SCATTER_BLOCK` (see Section 5.10.1).

5.12.10 Nonblocking Reduce-Scatter

`MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcnts, datatype, op, comm, request)`

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnts	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(const void* sendbuf, void* recvbuf, const
                        int recvcnts[], MPI_Datatype datatype, MPI_Op op,
                        MPI_Comm comm, MPI_Request *request)
```

```
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm,
                    request, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
                    REQUEST, IERROR)
```

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

This call starts a nonblocking variant of `MPI_REDUCE_SCATTER` (see Section 5.10.2).

5.12.11 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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

This call starts a nonblocking variant of `MPI_SCAN` (see Section 5.11).

5.12.12 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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

This call starts a nonblocking variant of **MPI_EXSCAN** (see Section 5.11.2).

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 5.25

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 5.26

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 dependencies occur. Nonblocking collective operations can alleviate this issue.

Example 5.27

The following is erroneous.

```

1  switch(rank) {
2      case 0:
3          MPI_Bcast(buf1, count, type, 0, comm);
4          MPI_Send(buf2, count, type, 1, tag, comm);
5          break;
6      case 1:
7          MPI_Recv(buf2, count, type, 0, tag, comm, status);
8          MPI_Bcast(buf1, count, type, 0, comm);
9          break;
10 }

```

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 5.28

An unsafe, non-deterministic program.

```

24 switch(rank) {
25     case 0:
26         MPI_Bcast(buf1, count, type, 0, comm);
27         MPI_Send(buf2, count, type, 1, tag, comm);
28         break;
29     case 1:
30         MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
31         MPI_Bcast(buf1, count, type, 0, comm);
32         MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
33         break;
34     case 2:
35         MPI_Send(buf2, count, type, 1, tag, comm);
36         MPI_Bcast(buf1, count, type, 0, comm);
37         break;
38 }

```

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.

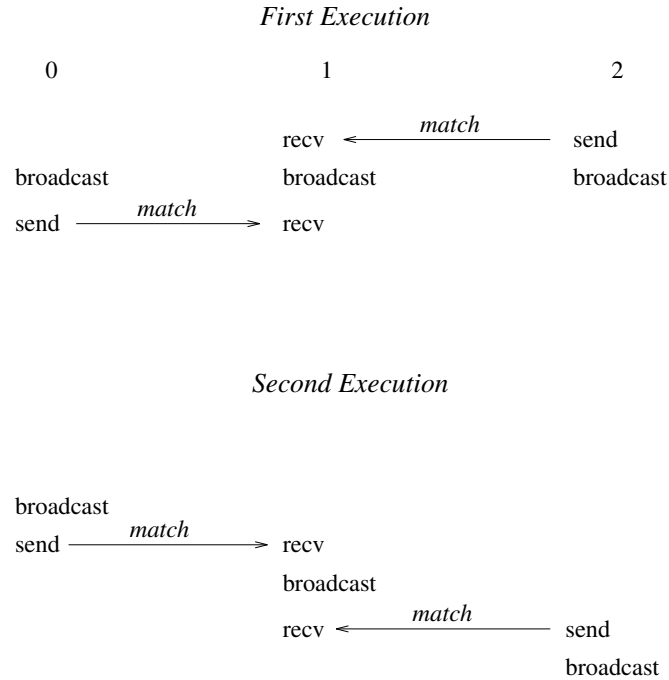


Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

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.*)

Example 5.29

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.

```

1  MPI_Request req;
2
3  MPI_Ibarrier(comm, &req);
4  MPI_Bcast(buf1, count, type, 0, comm);
5  MPI_Wait(&req, MPI_STATUS_IGNORE);
6

```

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 5.30

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.

```

16 MPI_Request req;
17 switch(rank) {
18     case 0:
19         /* erroneous matching */
20         MPI_Ibarrier(comm, &req);
21         MPI_Bcast(buf1, count, type, 0, comm);
22         MPI_Wait(&req, MPI_STATUS_IGNORE);
23         break;
24     case 1:
25         /* erroneous matching */
26         MPI_Bcast(buf1, count, type, 0, comm);
27         MPI_Ibarrier(comm, &req);
28         MPI_Wait(&req, MPI_STATUS_IGNORE);
29         break;
30 }
31

```

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 correct:

```

37 MPI_Request req;
38 MPI_Comm dupcomm;
39 MPI_Comm_dup(comm, &dupcomm);
40 switch(rank) {
41     case 0:
42         MPI_Ibarrier(comm, &req);
43         MPI_Bcast(buf1, count, type, 0, dupcomm);
44         MPI_Wait(&req, MPI_STATUS_IGNORE);
45         break;
46     case 1:
47         MPI_Bcast(buf1, count, type, 0, dupcomm);
48         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 5.31

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 the barrier in the MPI_Recv call. Thus, the MPI_Wait call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually return.

Example 5.32

Blocking and nonblocking collective operations do not match. The following example is erroneous.

```

MPI_Request req;

switch(rank) {
    case 0:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
        break;
}

```

Example 5.33

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 MPI_Waitall call returns only after the barrier and the receive completed.

Example 5.34

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. However, having too many outstanding requests might have a negative impact on performance. (*End of advice to users.*)

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

Example 5.35

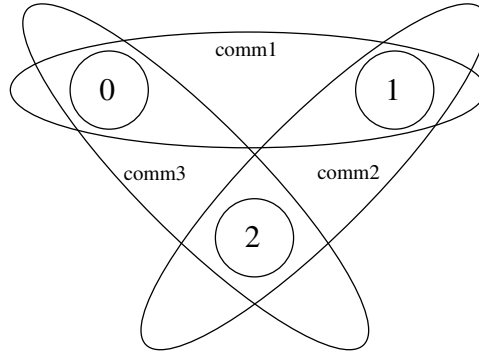


Figure 5.13: Example with overlapping communicators.

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. The sequence of the two calls in each process is irrelevant because the two nonblocking operations are performed on different communicators. (*End of advice to users.*)

Example 5.36

The progress of multiple outstanding nonblocking collective operations is completely independent.

```
1  MPI_Request reqs[2];
2
3  compute(buf1);
4  MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
5  compute(buf2);
6  MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
7  MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
8  /* nothing is known about the status of the first bcast here */
9  MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
```

Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via `reqs[1]`.

Bibliography

- [1] T. Hoefer, 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. Hoefer 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. Hoefer, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. 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. Hoefer, 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](#)

Index

CONST:, 36, 37
 CONST: , 36, 37
 CONST:Byte:, 37
 CONST:C integer, Fortran integer, Byte, Multi-
 language types, 37
 CONST:C integer, Fortran integer, Floating
 point,, 37
 CONST:C integer, Fortran integer, Floating
 point, Complex,, 37
 CONST:C integer, Logical, 37
 CONST:C integer:, 36
 CONST:C:, 40
 CONST:Complex:, 37
 CONST:Floating point:, 36
 CONST:Fortran integer:, 36
 CONST:Fortran:, 40
 CONST:Logical:, 37
 CONST:MPI_2DOUBLE_PRECISION, 40
 CONST:MPI_2INT, 40
 CONST:MPI_2INTEGER, 40
 CONST:MPI_2REAL, 40
 CONST:MPI_AINT, 37
 CONST:MPI_BAND, 36, 37
 CONST:MPI_BOR, 36, 37
 CONST:MPI_BOTTOM, 5
 CONST:MPI_BXOR, 36, 37
 CONST:MPI_BYTE, 37
 CONST:MPI_C_BOOL, 37
 CONST:MPI_C_COMPLEX, 37
 CONST:MPI_C_DOUBLE_COMPLEX, 37
 CONST:MPI_C_FLOAT_COMPLEX, 37
 CONST:MPI_C_LONG_DOUBLE_COMPLEX, 37
 CONST:MPI_CHAR, 38, 39
 CONST:MPI_CHARACTER, 38, 39
 CONST:MPI_COMPLEX, 37
 CONST:MPI_COMPLEX16, 37
 CONST:MPI_COMPLEX32, 37
 CONST:MPI_COMPLEX4, 37
 CONST:MPI_COMPLEX8, 37
 CONST:MPI_COUNT, 37
 CONST:MPI_CXX_BOOL, 37
 CONST:MPI_CXX_DOUBLE_COMPLEX, 37
 CONST:MPI_CXX_FLOAT_COMPLEX, 37
 CONST:MPI_CXX_LONG_DOUBLE_COMPLEX, 37
 CONST:MPI_DOUBLE, 36
 CONST:MPI_DOUBLE_COMPLEX, 37
 CONST:MPI_DOUBLE_INT, 40
 CONST:MPI_DOUBLE_PRECISION, 36
 CONST:MPI_ERROR, 57
 CONST:MPI_FLOAT, 34, 36
 CONST:MPI_FLOAT_INT, 40
 CONST:MPI_IN_PLACE, 4, 31
 CONST:MPI_INT, 36
 CONST:MPI_INT16_T, 36
 CONST:MPI_INT32_T, 36
 CONST:MPI_INT64_T, 36
 CONST:MPI_INT8_T, 36
 CONST:MPI_INTEGER, 36
 CONST:MPI_INTEGER1, 36
 CONST:MPI_INTEGER16, 36
 CONST:MPI_INTEGER2, 36
 CONST:MPI_INTEGER4, 36
 CONST:MPI_INTEGER8, 36
 CONST:MPI_LAND, 36, 37
 CONST:MPI_LOGICAL, 37
 CONST:MPI_LONG, 36
 CONST:MPI_LONG_DOUBLE, 36
 CONST:MPI_LONG_DOUBLE_INT, 40
 CONST:MPI_LONG_INT, 40
 CONST:MPI_LONG_LONG, 36
 CONST:MPI_LONG_LONG_INT, 36
 CONST:MPI_LOR, 36, 37
 CONST:MPI_LXOR, 36, 37
 CONST:MPI_MAX, 34, 36, 37, 54
 CONST:MPI_MAXLOC, 36, 39, 40, 43
 CONST:MPI_MIN, 36, 37
 CONST:MPI_MINLOC, 36, 39, 40, 43
 CONST:MPI_OFFSET, 37

CONST:MPI_Op, 34 , 43 , 45 , 47 , 49–51 , 53 , 54 , 69–74	1
CONST:MPI_OP_NULL, 46	2
CONST:MPI_PROC_NULL, 6 , 8 , 10 , 12 , 20 , 22 , 35	3
CONST:MPI_PROD, 36 , 37	4
CONST:MPI_REAL, 36	5
CONST:MPI_REAL16, 37	6
CONST:MPI_REAL2, 37	7
CONST:MPI_REAL4, 37	8
CONST:MPI_REAL8, 37	9
CONST:MPI_ROOT, 6	10
CONST:MPI_SHORT, 36	11
CONST:MPI_SHORT_INT, 40	12
CONST:MPI_SIGNED_CHAR, 36 , 38 , 39	13
CONST:MPI_SOURCE, 57	14
CONST:MPI_SUM, 36 , 37	15
CONST:MPI_TAG, 57	16
CONST:MPI_UINT16_T, 36	17
CONST:MPI_UINT32_T, 36	18
CONST:MPI_UINT64_T, 36	19
CONST:MPI_UINT8_T, 36	20
CONST:MPI_UNSIGNED, 36	21
CONST:MPI_UNSIGNED_CHAR, 36 , 38 , 39	22
CONST:MPI_UNSIGNED_LONG, 36	23
CONST:MPI_UNSIGNED_LONG_LONG, 36	24
CONST:MPI_UNSIGNED_SHORT, 36	25
CONST:MPI_WCHAR, 38 , 39	26
CONST:Multi-language types:, 37	27
CONST:Name, 36 , 40	28
CONST:Op, 37	29
EXAMPLES:Deadlock	30
with MPI_Bcast, 74 , 75	31
EXAMPLES:False matching of collective operations, 78	32
EXAMPLES:Independence of nonblocking operations, 81	33
EXAMPLES:Mixing blocking and nonblocking collective operations, 77	34
EXAMPLES:Mixing collective and point-to-point requests, 80	35
EXAMPLES:MPI_Allgather, 27	36
EXAMPLES:MPI_ALLREDUCE, 48	37
EXAMPLES:MPI_Alltoall, 79	38
EXAMPLES:MPI_Bcast, 9 , 74–78	39
EXAMPLES:MPI_Gather, 12 , 13 , 17	40
EXAMPLES:MPI_Gatherv, 14–17	41
EXAMPLES:MPI_Iallreduce, 80	42
EXAMPLES:MPI_Ialltoall, 79	43
EXAMPLES:MPI_Ibarrier, 77–80	44
EXAMPLES:MPI_Ibcast, 59 , 80 , 81	45
EXAMPLES:MPI_Irecv, 80	46
EXAMPLES:MPI_Op_create, 46 , 47 , 55	47
EXAMPLES:MPI_Recv, 79	48
EXAMPLES:MPI_REDUCE, 37 , 38 , 41	
EXAMPLES:MPI_Reduce, 40 , 42 , 46 , 47	
EXAMPLES:MPI_Scan, 55	
EXAMPLES:MPI_Scatter, 22	
EXAMPLES:MPI_Scatterv, 22 , 23	
EXAMPLES:MPI_Send, 79 , 80	
EXAMPLES:MPI_Type_commit, 13–17 , 23 , 55	
EXAMPLES:MPI_Type_contiguous, 13	
EXAMPLES:MPI_Type_create_struct, 16 , 17 , 55	
EXAMPLES:MPI_Type_vector, 14 , 15 , 17 , 23	
EXAMPLES:MPI_User_function, 47	
EXAMPLES:MPI_Wait, 77–80	
EXAMPLES:MPI_Waitall, 80	
EXAMPLES:No Matching of Blocking and Nonblocking collective operations, 79	
EXAMPLES:Non-deterministic program with MPI_Bcast, 76	
EXAMPLES:Overlapping Communicators, 80	
EXAMPLES:Pipelining nonblocking collective operations, 80	
EXAMPLES:Progression of nonblocking collective operations, 79	
MPI_ABORT, 44	
MPI_ALLGATHER, 1 , 5 , 6 , 25–28 , 64	
MPI_ALLGATHER(...), 25	
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm), 25	
MPI_ALLGATHERV, 1 , 5 , 6 , 27 , 65	
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm), 26	
MPI_ALLREDUCE, 1 , 4–6 , 36 , 43 , 48 , 70	
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm), 47	
MPI_ALLTOALL, 1 , 5 , 6 , 28–31 , 66	

- MPI_ALLTOALL(sendbuf, sendcount, send-
 type, recvbuf, recvcount, recvtype,
 comm), [28](#)
 MPI_ALLTOALLV, [1](#), [5](#), [6](#), [30](#), [31](#), [33](#), [67](#)
 MPI_ALLTOALLV(sendbuf, sendcounts, sdis-
 pls, sendtype, recvbuf, recvcounts,
 rdispls, recvtype, comm), [30](#)
 MPI_ALLTOALLW, [1](#), [5](#), [6](#), [33](#), [69](#)
 MPI_ALLTOALLW(sendbuf, sendcounts, sdis-
 pls, sendtypes, recvbuf, recvcounts,
 rdispls, recvtypes, comm), [32](#)
 MPI_BARRIER, [1](#), [5](#), [7](#), [8](#), [58](#)
 MPI_BARRIER(comm), [7](#)
 MPI_BCAST, [1](#), [5](#), [8](#), [9](#), [35](#), [59](#), [78](#)
 MPI_BCAST(buffer, count, datatype, root,
 comm), [8](#)
 MPI_CANCEL, [57](#)
 MPI_EXSCAN, [2](#), [5](#), [36](#), [43](#), [54](#), [74](#)
 MPI_EXSCAN(sendbuf, recvbuf, count, datatype,
 op, comm), [54](#)
 MPI_GATHER, [1](#), [4–6](#), [11](#), [12](#), [19](#), [20](#), [25](#),
[35](#), [61](#)
 MPI_GATHER(sendbuf, sendcount, sendtype,
 recvbuf, recvcount, recvtype, root,
 comm), [9](#)
 MPI_GATHERV, [1](#), [5](#), [6](#), [11–13](#), [21](#), [27](#), [62](#)
 MPI_GATHERV(sendbuf, sendcount, send-
 type, recvbuf, recvcounts, displs, recv-
 type, root, comm), [11](#)
 MPI_IALLGATHER, [1](#), [5](#), [6](#)
 MPI_IALLGATHER(sendbuf, sendcount, send-
 type, recvbuf, recvcount, recvtype,
 comm, request), [64](#)
 MPI_IALLGATHERV, [1](#), [5](#), [6](#)
 MPI_IALLGATHERV(sendbuf, sendcount, send-
 type, recvbuf, recvcounts, displs, recv-
 type, comm, request), [65](#)
 MPI_IALLREDUCE, [1](#), [5](#), [6](#)
 MPI_IALLREDUCE(sendbuf, recvbuf, count,
 datatype, op, comm, request), [70](#)
 MPI_IALLTOALL, [1](#), [5](#), [6](#)
 MPI_IALLTOALL(sendbuf, sendcount, send-
 type, recvbuf, recvcount, recvtype,
 comm, request), [66](#)
 MPI_IALLTOALLV, [1](#), [5](#), [6](#)
 MPI_IALLTOALLV(sendbuf, sendcounts, sdis-
 pls, sendtype, recvbuf, recvcounts,
 rdispls, recvtype, comm, request), [67](#)
 MPI_IALLTOALLW, [1](#), [5](#), [6](#)
 MPI_IALLTOALLW(sendbuf, sendcounts, sdis-
 pls, sendtypes, recvbuf, recvcounts,
 rdispls, recvtypes, comm, request),
[68](#)
 MPI_IBARRIER, [1](#), [5](#), [57–59](#), [78](#)
 MPI_IBARRIER(comm, request), [58](#)
 MPI_IBCAST, [1](#), [5](#), [59](#), [82](#)
 MPI_IBCAST(buffer, count, datatype, root,
 comm, request), [59](#)
 MPI_IEXSCAN, [2](#), [5](#)
 MPI_IEXSCAN(sendbuf, recvbuf, count, datatype,
 op, comm, request), [74](#)
 MPI_IGATHER, [1](#), [5](#), [6](#)
 MPI_IGATHER(sendbuf, sendcount, sendtype,
 recvbuf, recvcount, recvtype, root,
 comm, request), [60](#)
 MPI_IGATHERV, [1](#), [5](#), [6](#)
 MPI_IGATHERV(sendbuf, sendcount, send-
 type, recvbuf, recvcounts, displs, recv-
 type, root, comm, request), [61](#)
 MPI_IREDUCE, [1](#), [5](#), [6](#), [69](#)
 MPI_IREDUCE(sendbuf, recvbuf, count, datatype,
 op, root, comm, request), [69](#)
 MPI_IREDUCE_SCATTER, [1](#), [5](#), [6](#)
 MPI_IREDUCE_SCATTER(sendbuf, recvbuf,
 recvcounts, datatype, op, comm, re-
 quest), [72](#)
 MPI_IREDUCE_SCATTER_BLOCK, [1](#), [5](#),
[6](#)
 MPI_IREDUCE_SCATTER_BLOCK(sendbuf,
 recvbuf, recvcount, datatype, op, comm,
 request), [71](#)
 MPI_ISCAN, [2](#), [5](#)
 MPI_ISCAN(sendbuf, recvbuf, count, datatype,
 op, comm, request), [73](#)
 MPI_ISCATTER, [1](#), [5](#), [6](#)
 MPI_ISCATTER(sendbuf, sendcount, send-
 type, recvbuf, recvcount, recvtype,
 root, comm, request), [62](#)
 MPI_ISCATTERV, [1](#), [5](#), [6](#)
 MPI_ISCATTERV(sendbuf, sendcounts, dis-
 pls, sendtype, recvbuf, recvcount, recv-
 type, root, comm, request), [63](#)
 MPI_OP_COMMUTATIVE(op, commute),
[49](#)
 MPI_OP_CREATE, [43](#), [45](#)
 MPI_OP_CREATE(user_fn, commute, op),

43	1
MPI_OP_FREE(op), 45	2
MPI_RECV, 2 , 79	3
MPI_RECV(recvbuf, recvcount·n, recvtype, ...), 10	4
MPI_REDUCE, 1 , 5 , 6 , 34–36 , 43–45 , 48 , 51–54 , 69	5
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm), 34	6
MPI_REDUCE_LOCAL, 35 , 36 , 43	7
MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op), 49	8
MPI_REDUCE_SCATTER, 1 , 5 , 6 , 36 , 43 , 51 , 52 , 72	9
MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm), 51	10
MPI_REDUCE_SCATTER_BLOCK, 1 , 5 , 6 , 36 , 43 , 50 , 51 , 71	11
MPI_REDUCE_SCATTER_BLOCK(send- buf, recvbuf, recvcount, datatype, op, comm), 50	12
MPI_REQUEST_FREE, 57	13
MPI_SCAN, 2 , 5 , 36 , 43 , 53 , 55 , 73	14
MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm), 53	15
MPI_SCATTER, 1 , 5 , 6 , 19 , 21 , 22 , 51 , 62	16
MPI_SCATTER(sendbuf, sendcount, send- type, recvbuf, recvcount, recvtype, root, comm), 19	17
MPI_SCATTERV, 1 , 5 , 6 , 21 , 22 , 52 , 63	18
MPI_SCATTERV(sendbuf, sendcounts, dis- pls, sendtype, recvbuf, recvcount, recv- type, root, comm), 21	19
MPI_SEND, 79	20
MPI_TYPE_CREATE_F90_COMPLEX, 37	21
MPI_TYPE_CREATE_F90_INTEGER, 36	22
MPI_TYPE_CREATE_F90_REAL, 37	23
MPI_TYPE_CREATE_STRUCT, 33	24
MPI_WAIT, 57 , 79	25
MPI_WAITALL, 57 , 80	26
TYPEDEF:MPI_User_function, 47	27
TYPEDEF:MPI_User_function(void* invec, void* inoutvec, int *len, MPI_Datatype *datatype), 43	28