DRAFT

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

May 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 4.7). 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 5.14.

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.

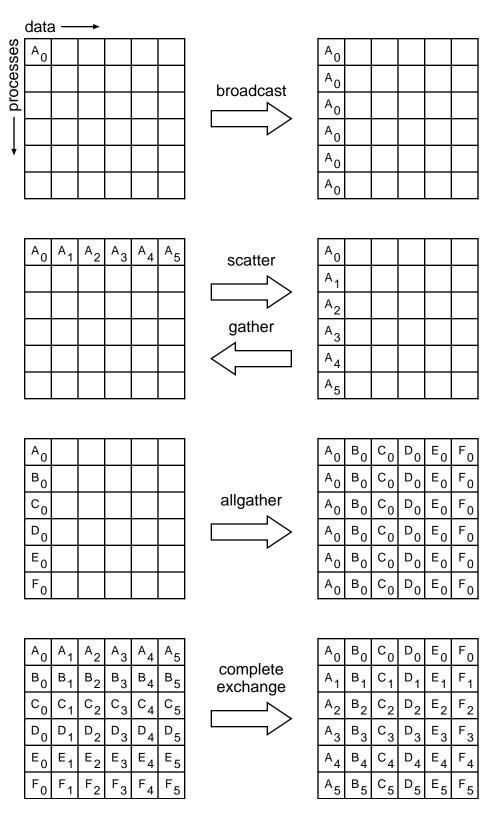


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

6 9

12 13 14

10

11

17 18

19

20 21

15

16

22 23 24

25

26

29

30

27 28

31 32 33

35 36 37

38

39

34

41 42 43

44 45

46

47

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 5.14. (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 5.14. (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.

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.

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.

11

12 13

14 15

16

17

18

19 20

21

22 23

24

25 26

27

28 29

30

31 32

34

35

36

37

38

39

42

43

44

45 46

47

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.

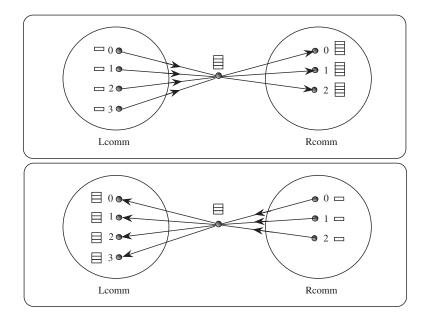


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

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

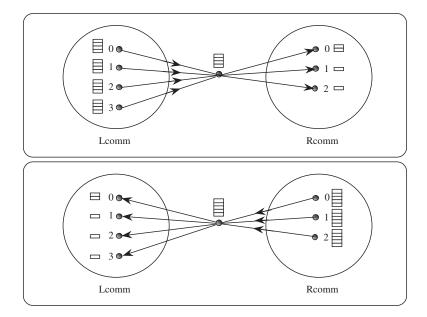


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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)
```

```
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

5.5. GATHER 9

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.

11

12

13 14

15 16

17

18 19

24

26

27 28

29

30

31

32

34

35

36

37

38 39

41

42 43

44

45

46

47

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 inte-
                                           ger)
IN
           sendtype
                                           data type of send buffer elements (handle)
OUT
            recvbuf
                                           address of receive buffer (choice, significant only at
                                           root)
IN
                                           number of elements for any single receive (non-negative
            recvcount
                                           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
                                           communicator (handle)
            comm
```

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

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.

5.5. *GATHER* 11

MPI_C	GATHERV(sendbuf, sendcount comm)	,,,,,,,,	1
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		7
OUT	recvbuf	root)	8
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 13
IN	displs	the displacement relative to recvbuf at which to place the incoming data from process i (significant only at	14 15 16 17
IN	recvtype	root) (handle)	18
IN	root	rank of receiving process (integer)	20 21
IN	comm	(2)	22
int M	void* recvbuf, c	endbuf, int sendcount, MPI_Datatype sendtype, onst int recvcounts[], const int displs[], vtype, int root, MPI_Comm comm) 2	23 24 25 26 27
MPI_G			28
т.	recvtype, root, YPE(*), DIMENSION(), I	-	29
	YPE(*), DIMENSION(), II	· recubuf	30
	•	endcount recucounts(*) displa(*) root	31 32
		T(IN) ·· sandtune recutune	33
	YPE(MPI_Comm), INTENT(IN)	0	34
I	NTEGER, OPTIONAL, INTENT	(OUT) :: ierror	35
MPI_G	ATHERV(SENDBUF, SENDCOUN	r, sendtype, recvbuf, recvcounts, displs,	36
	RECVTYPE, ROOT,	CUMM, IERRUR)	37
	type> SENDBUF(*), RECVBUI		38 39
	NTEGER SENDCOUNT, SENDTY DMM, IERROR	PE, RECYCUUNIS(*), DISPLS(*), RECVIIPE, RUUI,	10
			11
			12
		he reat by providing the new argument disple	13
	_	or the outcome is as if each process including the root	14
	s, sends a message to the roo	+	45 46
_	·	4	17
MPI_	Send(sendbut, sendcount, send	Itype, root,), and the root executes n receives,	18

MPI_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[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 recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

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

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

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

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

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

Example 5.2

Gather 100 ints from every process in group to root. See Figure 5.4.

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

Example 5.3

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

5.5. *GATHER* 13

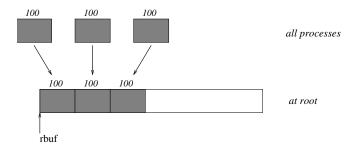


Figure 5.4: The root process gathers 100 ints from each process in the group.

```
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.

2

6

9

10

11

30

31 32

33

34

35

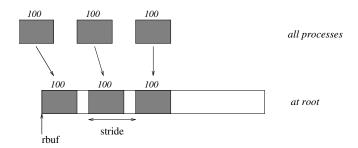


Figure 5.5: The root process gathers 100 ints from each process in the group, each set is placed stride ints apart.

```
12
         MPI_Comm comm;
13
         int gsize,sendarray[100];
14
         int root, *rbuf, stride;
15
         int *displs,i,*rcounts;
16
17
         . . .
19
         MPI_Comm_size(comm, &gsize);
20
         rbuf = (int *)malloc(gsize*stride*sizeof(int));
21
         displs = (int *)malloc(gsize*sizeof(int));
22
         rcounts = (int *)malloc(gsize*sizeof(int));
23
         for (i=0; i<gsize; ++i) {
24
             displs[i] = i*stride;
             rcounts[i] = 100;
         }
27
         MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
28
                                                                         root, comm);
29
```

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;
36
         int gsize, sendarray[100][150];
37
         int root, *rbuf, stride;
38
         MPI_Datatype stype;
39
         int *displs,i,*rcounts;
41
42
43
         MPI_Comm_size(comm, &gsize);
44
         rbuf = (int *)malloc(gsize*stride*sizeof(int));
45
         displs = (int *)malloc(gsize*sizeof(int));
46
         rcounts = (int *)malloc(gsize*sizeof(int));
47
         for (i=0; i<gsize; ++i) {</pre>
```

5.5. *GATHER* 15

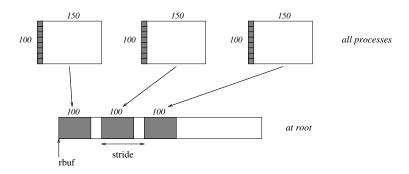


Figure 5.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

11

12 13

14

15

16

17

18

19

20

21

22 23 24

25

26

27

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;
                                                                                28
int gsize,sendarray[100][150],*sptr;
                                                                                29
int root, *rbuf, stride, myrank;
                                                                                30
MPI_Datatype stype;
                                                                                31
int *displs,i,*rcounts;
                                                                                32
                                                                                34
. . .
                                                                                35
MPI_Comm_size(comm, &gsize);
                                                                                36
MPI_Comm_rank(comm, &myrank);
                                                                                37
rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                38
displs = (int *)malloc(gsize*sizeof(int));
                                                                                39
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
                                                                                41
    displs[i] = i*stride;
                                                                                42
    rcounts[i] = 100-i;
                             /* note change from previous example */
                                                                                43
}
                                                                                44
/* Create datatype for the column we are sending
                                                                                45
                                                                                46
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                47
MPI_Type_commit(&stype);
```

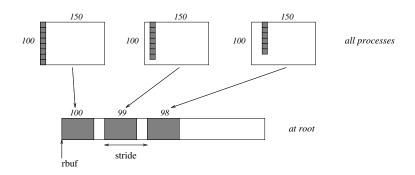


Figure 5.7: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride ints apart.

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;
26
27
         int gsize, sendarray[100][150], *sptr;
         int root, *rbuf, stride, myrank;
28
         MPI_Datatype stype;
29
         int *displs, i, *rcounts;
30
31
32
         . . .
         MPI_Comm_size(comm, &gsize);
34
         MPI_Comm_rank(comm, &myrank);
35
         rbuf = (int *)malloc(gsize*stride*sizeof(int));
36
         displs = (int *)malloc(gsize*sizeof(int));
37
         rcounts = (int *)malloc(gsize*sizeof(int));
38
         for (i=0; i<gsize; ++i) {
39
             displs[i] = i*stride;
             rcounts[i] = 100-i;
41
42
         }
         /* Create datatype for one int, with extent of entire row
43
44
         MPI_Type_create_resized( MPI_INT, 0, 150*sizeof(int), &stype);
45
         MPI_Type_commit(&stype);
46
         sptr = &sendarray[0][myrank];
47
         MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
```

5.5. GATHER 17

```
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;
                                                                                     11
    int *displs,i,*rcounts,offset;
                                                                                     12
                                                                                     13
                                                                                     14
                                                                                     15
    MPI_Comm_size(comm, &gsize);
                                                                                     16
    MPI_Comm_rank(comm, &myrank);
                                                                                     17
                                                                                     18
    stride = (int *)malloc(gsize*sizeof(int));
                                                                                     19
                                                                                     20
    /* stride[i] for i = 0 to gsize-1 is set somehow
                                                                                     21
     */
                                                                                     22
                                                                                     23
    /* set up displs and rounts vectors first
                                                                                     24
     */
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                     27
    offset = 0;
                                                                                     28
    for (i=0; i<gsize; ++i) {
                                                                                     29
        displs[i] = offset;
                                                                                     30
        offset += stride[i];
                                                                                     31
        rcounts[i] = 100-i;
    }
    /* the required buffer size for rbuf is now easily obtained
                                                                                     34
                                                                                     35
    bufsize = displs[gsize-1]+rcounts[gsize-1];
                                                                                     36
    rbuf = (int *)malloc(bufsize*sizeof(int));
                                                                                     37
    /* Create datatype for the column we are sending
                                                                                     38
     */
                                                                                     39
    MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
    MPI_Type_commit(&stype);
                                                                                     41
    sptr = &sendarray[0][myrank];
                                                                                     42
    MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                                                     43
                                                            root, comm);
                                                                                     44
                                                                                     45
                                                                                     46
Example 5.10
                                                                                     47
```

2

6

9 10

11

12 13

14

15

16

17

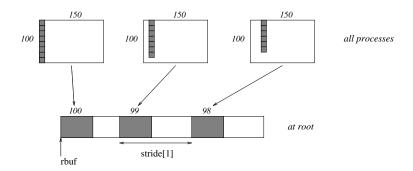


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

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;
19
         int gsize,sendarray[100][150],*sptr;
20
         int root, *rbuf, myrank;
21
         MPI_Datatype stype;
22
         int *displs,i,*rcounts,num;
23
24
         . . .
25
         MPI_Comm_size(comm, &gsize);
         MPI_Comm_rank(comm, &myrank);
28
29
         /* First, gather nums to root
30
          */
31
         rcounts = (int *)malloc(gsize*sizeof(int));
32
         MPI_Gather(&num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
         /* root now has correct rounts, using these we set displs[] so
34
          * that data is placed contiguously (or concatenated) at receive end
35
          */
36
         displs = (int *)malloc(gsize*sizeof(int));
37
         displs[0] = 0;
38
         for (i=1; i<gsize; ++i) {</pre>
39
             displs[i] = displs[i-1]+rcounts[i-1];
         }
41
         /* And, create receive buffer
42
43
         rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
44
                                                                      *sizeof(int));
45
         /* Create datatype for one int, with extent of entire row
46
          */
47
         MPI_Type_create_resized( MPI_INT, 0, 150*sizeof(int), &stype);
```

5.6. SCATTER

```
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)
                                                                                           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
  IN
            sendtype
                                        data type of send buffer elements (significant only at
                                                                                          15
                                        root) (handle)
                                                                                           16
                                                                                           17
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                           18
  IN
                                        number of elements in receive buffer (non-negative in-
            recvcount
                                                                                           19
                                        teger)
                                                                                          20
  IN
            recvtype
                                        data type of receive buffer elements (handle)
                                                                                          21
                                                                                          22
  IN
                                        rank of sending process (integer)
            root
                                                                                          23
  IN
                                        communicator (handle)
            comm
                                                                                          24
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                                                                                          27
               MPI_Comm comm)
                                                                                          28
                                                                                          29
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                          30
               root, comm, ierror)
                                                                                          31
    TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                                                           32
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                          34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          37
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                          38
               ROOT, COMM, IERROR)
                                                                                          39
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
                                                                                           41
                                                                                          42
    MPI_SCATTER is the inverse operation to MPI_GATHER.
                                                                                          43
    If comm is an intracommunicator, the outcome is as if the root executed n send oper-
                                                                                          44
ations,
                                                                                          45
                                                                                          46
 MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...), and each
                                                                                           47
process executed a receive,
```

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

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

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

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

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

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

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

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such 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.

Unofficial Draft for Comment Only

5.6. SCATTER 21

	comm)	
IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
		, ,
nt MPI_	Scatterv(const void	* sendbuf, const int sendcounts[], const
	<pre>int displs[],</pre>	<pre>MPI_Datatype sendtype, void* recvbuf,</pre>
	int recvcount,	, MPI_Datatype recvtype, int root, MPI_Comm comm)
'I_Scat		counts, displs, sendtype, recvbuf, recvcount,
		t, comm, ierror)
	E(*), DIMENSION(),	
1 1/1 1 F		:: recvbuf
	EGER, INTENT(IN) ::	sendcounts(*), displs(*), recvcount, root
TYPE	EGER, INTENT(IN) :: E(MPI_Datatype), INT	<pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype</pre>
TYPE TYPE	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(<pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm</pre>
TYPE TYPE	EGER, INTENT(IN) :: E(MPI_Datatype), INT	<pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm</pre>
TYPE TYPE INTE	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE	<pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm</pre>
TYPE TYPE INTE	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDC	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm ENT(OUT) :: ierror
TYPE TYPE INTE	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDC	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (I, COMM, IERROR)
TYPE TYPE INTE PI_SCAT <typ< td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT De> SENDBUF(*), RECV</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (I, COMM, IERROR)</td></typ<>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT De> SENDBUF(*), RECV	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (I, COMM, IERROR)
TYPE TYPE INTE PI_SCAT <typ inte<="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT De> SENDBUF(*), RECV</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*)</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT De> SENDBUF(*), RECV	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*)
TYPE TYPE INTE PI_SCAT <typ comm<="" inte="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR</td><td><pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre></td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR	<pre>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre>
TYPE TYPE INTE PI_SCAT <typ comm<="" inte="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR ESCATTERV is the invent</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV.</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR ESCATTERV is the invent	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV.
TYPE TYPE INTE PI_SCAT <typ comm="" inte="" mpi="" mpi<="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV is the invent</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV is the invent	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying
TYPE TYPE INTE PI_SCAT <typ comm="" inte="" mpi="" of<="" ount="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR _SCATTERV is the invelocation of the control of the control</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR _SCATTERV is the invelocation of the control	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows
TYPE TYPE INTE PI_SCAT <typ comm="" flex<="" inte="" mpi="" of="" ore="" ount="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), M, IERROR ESCATTERV is the inventional service of the sent to each ibility as to where the service of the service of</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), M, IERROR ESCATTERV is the inventional service of the sent to each ibility as to where the service of	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying
TYPE TYPE INTE PI_SCAT <typ comm="" flex="" gument<="" inte="" mpi="" of="" ore="" ount="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV extends to data to be sent to each ibility as to where the of the disple.</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows data is taken from on the root, by providing an additional</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV extends to data to be sent to each ibility as to where the of the disple.	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows data is taken from on the root, by providing an additional
TYPE TYPE INTE PI_SCAT <typ co<="" comm="" flex="" if="" inte="" mpi="" of="" ore="" ount="" rgument="" td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV extends to data to be sent to each ibility as to where the of the disple.</td><td>sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying a process, since sendcounts is now an array. It also allows</td></typ>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV extends to data to be sent to each ibility as to where the of the disple.	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (F, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying a process, since sendcounts is now an array. It also allows
TYPE TYPE INTE INTE COMM MPI MPI Dunt of nore flex rgument	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), I, IERROR SCATTERV extends to data to be sent to each ibility as to where the of the disple.	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows data is taken from on the root, by providing an additional
TYPE TYPE INTE INTE COMM MPI MPI ount of core flex regument If co	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), M, IERROR ESCATTERV extends the data to be sent to each ibility as to where the design of the send is an intracommunication.	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows data is taken from on the root, by providing an additional icator, the outcome is as if the root executed n send oper-
TYPE TYPE INTE INTE CLASSING STATE COMM MPI MPI Ount of core flex regument If co- ctions, MPI_Ser	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(EGER, OPTIONAL, INTE ETERV(SENDBUF, SENDO RECVTYPE, ROOT DE> SENDBUF(*), RECV EGER SENDCOUNTS(*), M, IERROR ESCATTERV extends the data to be sent to each ibility as to where the design of the send is an intracommunication.	sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: ierror COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, (T, COMM, IERROR) (BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, erse operation to MPI_GATHERV. the functionality of MPI_SCATTER by allowing a varying in process, since sendcounts is now an array. It also allows data is taken from on the root, by providing an additional

 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 $stride \ge 100$. See Figure 5.10.

5.6. SCATTER 23

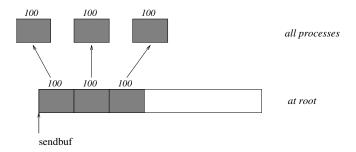


Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

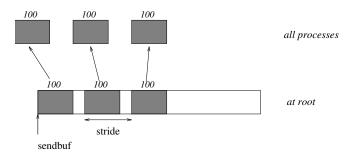


Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

```
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);</pre>
```

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],*rptr;

48
```

2 3

6

9 10

11

12 13

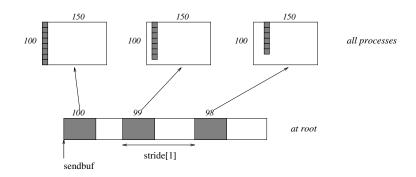


Figure 5.11: The root scatters blocks of 100-i ints into column i of a 100×150 C array. At the sending side, the blocks are stride[i] ints apart.

```
int root, *sendbuf, myrank, *stride;
14
         MPI_Datatype rtype;
15
         int i, *displs, *scounts, offset;
16
17
         MPI_Comm_size(comm, &gsize);
         MPI_Comm_rank(comm, &myrank);
19
20
         stride = (int *)malloc(gsize*sizeof(int));
21
22
         /* stride[i] for i = 0 to gsize-1 is set somehow
23
          * sendbuf comes from elsewhere
24
          */
25
         displs = (int *)malloc(gsize*sizeof(int));
27
         scounts = (int *)malloc(gsize*sizeof(int));
28
         offset = 0;
29
         for (i=0; i<gsize; ++i) {</pre>
30
             displs[i] = offset;
31
             offset += stride[i];
32
             scounts[i] = 100 - i;
         }
34
         /* Create datatype for the column we are receiving
35
          */
36
         MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
37
         MPI_Type_commit(&rtype);
38
         rptr = &recvarray[0][myrank];
39
         MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                                     root, comm);
41
42
```

47

Gather-to-all 5.7

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  IN
            sendbuf
                                        starting address of send buffer (choice)
  IN
            sendcount
                                        number of elements in send buffer (non-negative inte-
                                        ger)
  IN
            sendtype
                                        data type of send buffer elements (handle)
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                           11
                                                                                           12
  IN
            recvcount
                                        number of elements received from any process (non-
                                                                                           13
                                        negative integer)
                                                                                           14
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                           15
  IN
                                        communicator (handle)
            comm
                                                                                           16
                                                                                           17
int MPI_Allgather(const void* sendbuf, int sendcount,
                                                                                           18
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                           19
               MPI_Datatype recvtype, MPI_Comm comm)
                                                                                           20
                                                                                           21
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                           22
               comm, ierror)
                                                                                           23
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                           24
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                           27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                           28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           29
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                           30
               COMM, IERROR)
                                                                                           31
    <type> SENDBUF(*), RECVBUF(*)
                                                                                           32
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                           34
    MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive
                                                                                           35
the result, instead of just the root. The block of data sent from the j-th process is received
                                                                                           36
by every process and placed in the j-th block of the buffer recvbuf.
                                                                                           37
    The type signature associated with sendcount, sendtype, at a process must be equal to
                                                                                           38
the type signature associated with recvcount, recvtype at any other process.
                                                                                           39
    If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER(...) is as
if all processes executed n calls to
                                                                                           41
                                                                                           42
   MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                           43
                                                        recvtype,root,comm)
                                                                                           44
for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found
                                                                                           45
```

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.

5

6

7

9

10

11

12

13

14

15

16

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.)

17 18 19

20 21

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

```
IN
                  sendbuf
                                               starting address of send buffer (choice)
22
23
       IN
                  sendcount
                                               number of elements in send buffer (non-negative inte-
24
                                               ger)
25
       IN
                  sendtype
                                               data type of send buffer elements (handle)
26
                  recvbuf
       OUT
                                               address of receive buffer (choice)
27
28
       IN
                                               non-negative integer array (of length group size) con-
                  recvcounts
29
                                               taining the number of elements that are received from
30
                                               each process
31
       IN
                  displs
                                               integer array (of length group size). Entry i specifies
32
                                               the displacement (relative to recvbuf) at which to place
33
                                               the incoming data from process i
34
       IN
                  recvtype
                                               data type of receive buffer elements (handle)
35
36
       IN
                  comm
                                               communicator (handle)
37
38
      int MPI_Allgatherv(const void* sendbuf, int sendcount,
39
                     MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
40
                     const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
41
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
42
                     recvtype, comm, ierror)
43
          TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                        sendbuf
44
          TYPE(*), DIMENSION(..) :: recvbuf
45
          INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
46
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
          TYPE(MPI_Comm), INTENT(IN) ::
48
```

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

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.

42

43 44

45

46 47

48

5.8 All-to-All Scatter/Gather

```
3
4
     MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
5
       IN
                  sendbuf
                                             starting address of send buffer (choice)
6
7
                                             number of elements sent to each process (non-negative
       IN
                  sendcount
                                             integer)
       IN
                  sendtype
                                             data type of send buffer elements (handle)
10
       OUT
                  recvbuf
                                             address of receive buffer (choice)
11
12
       IN
                  recvcount
                                             number of elements received from any process (non-
13
                                             negative integer)
14
       IN
                                              data type of receive buffer elements (handle)
                  recvtype
15
       IN
                                             communicator (handle)
                  comm
16
17
     int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
18
                     void* recvbuf, int recvcount, MPI_Datatype recvtype,
19
                     MPI_Comm comm)
20
21
     MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
22
                     comm, ierror)
23
          TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
24
          TYPE(*), DIMENSION(..) :: recvbuf
25
          INTEGER, INTENT(IN) ::
                                     sendcount, recvcount
26
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
27
          TYPE(MPI_Comm), INTENT(IN) :: comm
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
30
31
                     COMM, IERROR)
32
          <type> SENDBUF(*), RECVBUF(*)
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
34
          MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process
35
     sends distinct data to each of the receivers. The j-th block sent from process i is received
36
     by process i and is placed in the i-th block of recvbuf.
37
          The type signature associated with sendcount, sendtype, at a process must be equal to
38
     the type signature associated with recvount, recvtype at any other process. This implies
39
     that the amount of data sent must be equal to the amount of data received, pairwise between
40
     every pair of processes. As usual, however, the type maps may be different.
41
```

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

 $MPI_Send(sendbuf+i\cdot sendcount\cdot extent(sendtype),sendcount,sendtype,i, ...)$, 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.)

size.

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls,
1
                     recvtype, comm)
2
3
       IN
                  sendbuf
                                               starting address of send buffer (choice)
       IN
                  sendcounts
                                               non-negative integer array (of length group size) spec-
                                               ifying the number of elements to send to each rank
6
       IN
                  sdispls
                                               integer array (of length group size). Entry i specifies
                                               the displacement (relative to sendbuf) from which to
                                               take the outgoing data destined for process i
10
       IN
                  sendtype
                                               data type of send buffer elements (handle)
11
       OUT
                  recvbuf
                                               address of receive buffer (choice)
12
       IN
                  recvcounts
                                               non-negative integer array (of length group size) spec-
13
                                               ifying the number of elements that can be received
14
                                               from each rank
15
16
       IN
                  rdispls
                                               integer array (of length group size). Entry i specifies
17
                                               the displacement (relative to recvbuf) at which to place
18
                                               the incoming data from process i
19
       IN
                                               data type of receive buffer elements (handle)
                  recvtype
20
21
       IN
                  comm
                                               communicator (handle)
22
23
      int MPI_Alltoallv(const void* sendbuf, const int sendcounts[], const
24
                     int sdispls[], MPI_Datatype sendtype, void* recvbuf, const
25
                     int recvcounts[], const int rdispls[], MPI_Datatype recvtype,
26
                     MPI_Comm comm)
27
     MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
28
                     rdispls, recvtype, comm, ierror)
29
          TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
30
          TYPE(*), DIMENSION(..) :: recvbuf
31
          INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
32
          rdispls(*)
33
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
          TYPE(MPI_Comm), INTENT(IN) ::
35
          INTEGER, OPTIONAL, INTENT(OUT) ::
36
37
     MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
38
                     RDISPLS, RECVTYPE, COMM, IERROR)
39
          <type> SENDBUF(*), RECVBUF(*)
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
41
          RECVTYPE, COMM, IERROR
42
          MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for
43
      the send is specified by sdispls and the location of the placement of the data on the receive
44
     side is specified by rdispls.
45
          If comm is an intracommunicator, then the j-th block sent from process i is received by
46
      process j and is placed in the i-th block of recvbuf. These blocks need not all have the same
47
```

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.)

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

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.

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

```
5.9.1 Reduce
1
2
3
4
     MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
       IN
                 sendbuf
                                             address of send buffer (choice)
6
       OUT
                 recvbuf
                                             address of receive buffer (choice, significant only at
7
                                             root)
8
9
       IN
                                             number of elements in send buffer (non-negative inte-
                 count
10
11
       IN
                 datatype
                                             data type of elements of send buffer (handle)
12
       IN
                                             reduce operation (handle)
                 op
13
14
       IN
                                             rank of root process (integer)
                 root
15
       IN
                 comm
                                             communicator (handle)
16
17
     int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
18
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
19
20
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
21
          TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
22
          TYPE(*), DIMENSION(..) :: recvbuf
23
          INTEGER, INTENT(IN) :: count, root
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
29
          <type> SENDBUF(*), RECVBUF(*)
30
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
31
```

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) = $global \max(sendbuf(2))$.

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.

8	Name	Meaning
9		3
10	MPI_MAX	maximum
11	MPI_MIN	minimum
12	MPI_SUM	sum
13	MPI_PROD	product
14	MPI_LAND	logical and
15	MPI_BAND	bit-wise and
16	MPI_LOR	logical or
17	MPI_BOR	bit-wise or
18	MPI_LXOR	logical exclusive or (xor)
19	MPI_BXOR	bit-wise exclusive or (xor)
20	MPI_MAXLOC	max value and location
21	MPI_MINLOC	min value and location

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

 23

 24

28 29 30 31 32 33 34 35 36 37	C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T,
38 39 40 41	Fortran integer:	MPI_UINT32_T, MPI_UINT64_T MPI_INTEGER, and handles returned from MPI_TYPE_CREATE_F90_INTEGER,
42 43 44 45 46 47 48	Floating point:	and if available: MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, MPI_INTEGER16 MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE and handles returned from

	MPI_TYPE_CREATE_F90_REAL,	
	and if available: MPI_REAL2,	
	MPI_REAL4, MPI_REAL8, MPI_REAL16	
Logical:	MPI_LOGICAL,MPI_C_BOOL,	
	MPI_CXX_BOOL	
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	
	MPI_C_FLOAT_COMPLEX (as synonym),	
	MPI_C_DOUBLE_COMPLEX,	
	MPI_C_LONG_DOUBLE_COMPLEX,	
	MPI_CXX_FLOAT_COMPLEX,	
	MPI_CXX_DOUBLE_COMPLEX,	
	MPI_CXX_LONG_DOUBLE_COMPLEX,	
	and handles returned from	
	MPI_TYPE_CREATE_F90_COMPLEX,	
	and if available: MPI_DOUBLE_COMPLEX,	
	MPI_COMPLEX4, MPI_COMPLEX8,	
	MPI_COMPLEX16, MPI_COMPLEX32	
Byte:	MPI_BYTE	
Multi-language types:	MPI_AINT, MPI_OFFSET, MPI_COUNT	
Now, the valid datatypes for each operation are specified below.		

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

These operations together with all listed datatypes are valid in all supported programming languages, see also Reduce Operations on page 54 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.

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
1
     REAL a(m), b(m)
                       ! local slice of array
2
3
     REAL c
                             ! result (at node zero)
     REAL sum
4
     INTEGER m, comm, i, ierr
5
6
     ! local sum
7
     sum = 0.0
8
     D0 i = 1, m
9
        sum = sum + a(i)*b(i)
10
11
     END DO
12
     ! global sum
13
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
14
     RETURN
15
     END
16
17
18
     Example 5.16
19
         A routine that computes the product of a vector and an array that are distributed
20
     across a group of processes and returns the answer at node zero.
21
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
       D0 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)
     RETURN
41
     END
42
43
     5.9.3
            Signed Characters and Reductions
44
     The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction opera-
45
     tions. MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER (which represent printable charac-
46
     ters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR,
47
```

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:

$$\left(\begin{array}{c} u\\i \end{array}\right) \circ \left(\begin{array}{c} v\\j \end{array}\right) = \left(\begin{array}{c} w\\k \end{array}\right)$$

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:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

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

and

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

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

2

6

9

10

11

12

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.

```
13
       Fortran:
14
       Name
                                               Description
15
       MPI_2REAL
                                               pair of REALs
16
       MPI_2DOUBLE_PRECISION
                                               pair of DOUBLE PRECISION variables
17
       MPI_2INTEGER
                                               pair of INTEGERS
19
20
       C:
21
                                               Description
       Name
22
                                               float and int
       MPI_FLOAT_INT
23
                                               double and int
       MPI_DOUBLE_INT
24
                                               long and int
       MPI_LONG_INT
25
                                               pair of int
       MPI_2INT
                                               short and int
       MPI_SHORT_INT
       MPI_LONG_DOUBLE_INT
                                               long double and int
28
29
          The datatype MPI_2REAL is as if defined by the following (see Section 4.1).
30
     MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
31
32
          Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.
33
          The datatype MPI_FLOAT_INT is as if defined by the following sequence of instructions.
34
35
     type[0] = MPI_FLOAT
36
     type[1] = MPI_INT
37
     disp[0] = 0
38
     disp[1] = sizeof(float)
39
     block[0] = 1
40
     block[1] = 1
41
     MPI_TYPE_CREATE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
42
```

Example 5.17

43

44

45

46

47

48

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.

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

```
/* each process has an array of 30 double: ain[30]
                                                                                    2
     */
    double ain[30], aout[30];
    int ind[30];
    struct {
        double val;
        int
              rank;
    } in[30], out[30];
    int i, myrank, root;
                                                                                    11
    MPI_Comm_rank(comm, &myrank);
                                                                                    12
    for (i=0; i<30; ++i) {
                                                                                    13
        in[i].val = ain[i];
                                                                                    14
        in[i].rank = myrank;
                                                                                    15
    }
                                                                                    16
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                    17
    /* At this point, the answer resides on process root
     */
                                                                                    19
    if (myrank == root) {
                                                                                    20
        /* read ranks out
                                                                                    21
         */
                                                                                    22
        for (i=0; i<30; ++i) {
                                                                                    23
            aout[i] = out[i].val;
                                                                                    24
            ind[i] = out[i].rank;
        }
    }
                                                                                    27
                                                                                    28
                                                                                    29
Example 5.18
                                                                                    30
   Same example, in Fortran.
                                                                                    31
                                                                                    32
    ! each process has an array of 30 double: ain(30)
                                                                                    34
                                                                                    35
    DOUBLE PRECISION ain(30), aout(30)
                                                                                    36
    INTEGER ind(30)
                                                                                    37
    DOUBLE PRECISION in(2,30), out(2,30)
                                                                                    38
    INTEGER i, myrank, root, ierr
                                                                                    39
    CALL MPI_COMM_RANK(comm, myrank, ierr)
    DO I=1, 30
                                                                                    42
        in(1,i) = ain(i)
                                                                                    43
        in(2,i) = myrank
                           ! myrank is coerced to a double
                                                                                    44
    END DO
                                                                                    45
                                                                                    46
    CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                                   comm, ierr)
```

```
! At this point, the answer resides on process root
2
         IF (myrank .EQ. root) THEN
              ! read ranks out
              DO I = 1, 30
                  aout(i) = out(1,i)
6
                  ind(i) = out(2,i) ! rank is coerced back to an integer
              END DO
         END IF
9
10
11
     Example 5.19
12
         Each process has a non-empty array of values. Find the minimum global value, the
13
     rank of the process that holds it and its index on this process.
14
     #define LEN
                     1000
15
16
     float val[LEN];
                              /* local array of values */
17
     int count;
                              /* local number of values */
     int myrank, minrank, minindex;
19
     float minval;
20
21
     struct {
22
         float value;
23
24
         int
                index;
     } in, out;
25
         /* local minloc */
27
     in.value = val[0];
28
29
     in.index = 0;
     for (i=1; i < count; i++)
30
         if (in.value > val[i]) {
31
              in.value = val[i];
32
              in.index = i;
         }
34
35
         /* global minloc */
36
     MPI_Comm_rank(comm, &myrank);
37
     in.index = myrank*LEN + in.index;
38
     MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
39
         /* At this point, the answer resides on process root
           */
41
42
     if (myrank == root) {
         /* read answer out
43
          */
44
         minval = out.value;
45
         minrank = out.index / LEN;
46
         minindex = out.index % LEN;
47
     }
48
```

6

10 11

12

13

14 15

16 17

18

19

20

21

22

23 24

26

27

28

29

30

32

34

35 36

37

39

40 41

42

43 44

45

46

47

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
                                     true if commutative; false otherwise.
          commute
 OUT
                                     operation (handle)
          op
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) ::
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) ::
MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
    EXTERNAL USER FN
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

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, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

MPI_OP_CREATE binds a user-defined reduction operation to an

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

SUBBOUTINE MPI User function(invec_inoutvec_len_datatype)

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

```
TYPE(C_PTR), VALUE :: invec, inoutvec

INTEGER :: len
TYPE(MPI_Datatype) :: datatype

SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)

type> INVEC(LEN), INOUTVEC(LEN)
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 , ..., 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, ..., 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.

13

14 15

16

17

19 20

21

22

24

27

28

29

30

31

32

34

35 36

37

38

39

Users who plan to mix languages should define their reduction functions accordingly. (End of advice to users.)

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

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

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

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

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
1
2
     MPI_OP_FREE(OP, IERROR)
3
         INTEGER OP, IERROR
4
         Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
6
7
     Example of User-defined Reduce
     It is time for an example of user-defined reduction. The example in this section uses an
9
     intracommunicator.
10
11
     Example 5.20 Compute the product of an array of complex numbers, in C.
12
     typedef struct {
13
         double real, imag;
14
     } Complex;
15
16
     /* the user-defined function
17
      */
18
     void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
19
     {
20
21
         int i;
         Complex c;
22
         Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
23
24
         for (i=0; i< *len; ++i) {
25
              c.real = inout->real*in->real -
                          inout->imag*in->imag;
27
              c.imag = inout->real*in->imag +
28
                          inout->imag*in->real;
29
              *inout = c;
30
              in++; inout++;
31
         }
32
     }
33
34
     /* and, to call it...
35
      */
36
37
38
         /* each process has an array of 100 Complexes
39
         Complex a[100], answer[100];
41
42
         MPI_Op myOp;
         MPI_Datatype ctype;
43
44
         /* explain to MPI how type Complex is defined
45
46
         MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
47
         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
     */
                                                                                         11
Example 5.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
                                                                                         12
  subroutine my_user_function( invec, inoutvec, len, type )
                                                                                         13
    use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                         14
    use mpi_f08
                                                                                         15
    type(c_ptr), value :: invec, inoutvec
                                                                                         16
    integer :: len
                                                                                         17
    type(MPI_Datatype) :: type
                                                                                         18
    real, pointer :: invec_r(:), inoutvec_r(:)
                                                                                         19
    if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
                                                                                         20
       call c_f_pointer(invec, invec_r, (/ len /) )
                                                                                         21
       call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
                                                                                         22
       inoutvec_r = invec_r + inoutvec_r
                                                                                         23
    end if
                                                                                         24
  end subroutine
5.9.6 All-Reduce
                                                                                         27
                                                                                         28
MPI includes a variant of the reduce operations where the result is returned to all processes
                                                                                         29
in a group. MPI requires that all processes from the same group participating in these
                                                                                         30
operations receive identical results.
                                                                                         31
                                                                                         32
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
                                                                                         34
 IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                         35
 OUT
           recybuf
                                       starting address of receive buffer (choice)
                                                                                         36
                                                                                         37
 IN
           count
                                       number of elements in send buffer (non-negative inte-
                                                                                         38
                                                                                         39
 IN
           datatype
                                       data type of elements of send buffer (handle)
 IN
                                       operation (handle)
           op
                                                                                         41
                                                                                         42
 IN
           comm
                                       communicator (handle)
                                                                                         43
                                                                                         44
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
                                                                                         45
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
                                                                                         46
```

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)

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

12 13

14

15

16 17

18

19

20

21

22

23

2425

26

27

28

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

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).

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

13

14

15

16

17 18

19

20 21

22

23

24

27

28

29

30

31

35

36

37

38 39

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
                                         number of elements in inbuf and inoutbuf buffers (non-
            count
                                         negative integer)
                                         data type of elements of inbuf and inoutbuf buffers
  IN
            datatype
                                         (handle)
  IN
                                         operation (handle)
            op
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)
```

```
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
```

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
```

```
LOGICAL, INTENT(OUT) :: commute
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
3
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
          LOGICAL COMMUTE
          INTEGER OP, IERROR
6
             Reduce-Scatter
     5.10
9
10
     MPI includes variants of the reduce operations where the result is scattered to all processes
11
     in a group on return. One variant scatters equal-sized blocks to all processes, while another
12
     variant scatters blocks that may vary in size for each process.
13
14
     5.10.1 MPI_REDUCE_SCATTER_BLOCK
15
16
17
     MPI_REDUCE_SCATTER_BLOCK( sendbuf, recvbuf, recvcount, datatype, op, comm)
18
19
       IN
                sendbuf
                                            starting address of send buffer (choice)
20
       OUT
                recybuf
                                            starting address of receive buffer (choice)
21
       IN
                 recvcount
                                            element count per block (non-negative integer)
22
23
       IN
                datatype
                                            data type of elements of send and receive buffers (han-
24
                                            dle)
       IN
                 op
                                            operation (handle)
       IN
                comm
                                            communicator (handle)
27
28
     int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
29
                    int recvcount, MPI_Datatype datatype, MPI_Op op,
30
                    MPI Comm comm)
31
32
     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
33
                    ierror)
34
          TYPE(*), DIMENSION(..), INTENT(IN) ::
35
          TYPE(*), DIMENSION(..) :: recvbuf
36
          INTEGER, INTENT(IN) :: recvcount
37
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
          TYPE(MPI_Op), INTENT(IN) :: op
39
          TYPE(MPI_Comm), INTENT(IN) :: comm
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
                    IERROR)
43
          <type> SENDBUF(*), RECVBUF(*)
44
          INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
45
46
         If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a
47
```

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

3

4

5

6

7

8

9

18

19

20

21

22

23

24

25

26 27

28

29

30

31 32

33

34

35

36

37

38

39 40

41

42

43 44

45

46

47

```
int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const
                  int recvcounts[], MPI_Datatype datatype, MPI_Op op,
2
                  MPI_Comm comm)
    MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                  ierror)
        TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
        TYPE(*), DIMENSION(..) :: recvbuf
        INTEGER, INTENT(IN) :: recvcounts(*)
        TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
        TYPE(MPI_Op), INTENT(IN) :: op
11
        TYPE(MPI_Comm), INTENT(IN) :: comm
12
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
    MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
14
                  IERROR)
15
16
         <type> SENDBUF(*), RECVBUF(*)
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
17
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ 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 recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[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, recvcounts[i] and datatype.

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

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[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 recvcounts argument, and provide input vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ 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 recvcounts[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 recvounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (End of rationale.)

5.11. SCAN 53

5.11 Scan 5.11.1 Inclusive Scan MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm) 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) datatype IN data type of elements of input buffer (handle) operation (handle) IN op IN communicator (handle) comm 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(*)

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.

INTEGER COUNT, DATATYPE, OP, COMM, IERROR

11 12

13

14

15 16 17

18

19

20

21

22

23

24

27 28

29

30

31

36

37

38

39

40

41

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46

47

48

```
5.11.2 Exclusive Scan
```

```
2
3
4
     MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
                 recvbuf
                                            starting address of receive buffer (choice)
7
8
       IN
                count
                                            number of elements in input buffer (non-negative in-
9
                                             teger)
       IN
                datatype
                                            data type of elements of input buffer (handle)
11
       IN
                op
                                            operation (handle)
12
13
       IN
                 comm
                                            intracommunicator (handle)
14
15
     int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
16
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
17
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
18
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
19
         TYPE(*), DIMENSION(..) :: recvbuf
20
         INTEGER, INTENT(IN) ::
                                     count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Op), INTENT(IN) :: op
23
         TYPE(MPI_Comm), INTENT(IN) ::
24
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
25
26
     MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
27
          <type> SENDBUF(*), RECVBUF(*)
28
```

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 $0, \ldots, i-1$ (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.

INTEGER COUNT, DATATYPE, OP, COMM, IERROR

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. SCAN 55

1 2

6

8

11 12

13 14

15 16 17

18

19 20 21

22

23 24

25

26

27

28 29

30

31

32

34

35

36 37

38

39

41

42

43

44

45

46

47

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:

The operator that produces this effect is

$$\left(\begin{array}{c} u \\ i \end{array}\right) \circ \left(\begin{array}{c} v \\ j \end{array}\right) = \left(\begin{array}{c} w \\ j \end{array}\right),$$

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++;
    }
```

2 3

26 27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

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.

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

5.12 Nonblocking Collective Operations

As described in Section 4.7, 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 non-blocking 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 4.7.3 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 4.2.5 on page 8. 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 4.7.3). 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 4.7.4.

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

```
31
     MPI_IBARRIER(comm, request)
32
       IN
                                           communicator (handle)
                comm
33
       OUT
34
                request
                                           communication request (handle)
35
36
     int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
37
     MPI_Ibarrier(comm, request, ierror)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Request), INTENT(OUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) ::
41
42
     MPI_IBARRIER(COMM, REQUEST, IERROR)
43
         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 intracommunicator 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)

```
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
```

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.

```
MPI_Comm comm;
          int array1[100], array2[100];
2
          int root=0;
          MPI_Request req;
          . . .
          MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
          compute(array2, 100);
          MPI_Wait(&req, MPI_STATUS_IGNORE);
9
10
     5.12.3 Nonblocking Gather
11
12
13
     MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
14
                    request)
15
       IN
                 sendbuf
                                             starting address of send buffer (choice)
16
       IN
                 sendcount
17
                                             number of elements in send buffer (non-negative inte-
                                             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
       IN
                                             number of elements for any single receive (non-negative
23
                  recvcount
24
                                             integer, significant only at root)
25
       IN
                                             data type of recv buffer elements (significant only at
                  recvtype
                                             root) (handle)
27
       IN
                                             rank of receiving process (integer)
                  root
28
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,
                    root, comm, request, ierror)
37
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
39
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
41
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
42
          TYPE(MPI_Comm), INTENT(IN) :: comm
          TYPE(MPI_Request), INTENT(OUT) :: request
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
46
                    ROOT, COMM, REQUEST, IERROR)
47
          <type> SENDBUF(*), RECVBUF(*)
48
```

```
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR
    This call starts a nonblocking variant of MPI_GATHER (see Section 5.5).
MPI_IGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root,
               comm, request)
            sendbuf
  IN
                                        starting address of send buffer (choice)
            sendcount
  IN
                                        number of elements in send buffer (non-negative inte-
                                        ger)
                                                                                            11
                                                                                            12
  IN
            sendtype
                                        data type of send buffer elements (handle)
                                                                                            13
  OUT
            recvbuf
                                        address of receive buffer (choice, significant only at
                                                                                            14
                                        root)
                                                                                           15
  IN
            recvcounts
                                        non-negative integer array (of length group size) con-
                                                                                           16
                                        taining the number of elements that are received from
                                                                                            17
                                        each process (significant only at root)
                                                                                            19
  IN
            displs
                                        integer array (of length group size). Entry i specifies
                                                                                           20
                                        the displacement relative to recvbuf at which to place
                                                                                           21
                                        the incoming data from process i (significant only at
                                                                                           22
                                        root)
                                                                                           23
  IN
                                        data type of recv buffer elements (significant only at
            recvtype
                                                                                           24
                                        root) (handle)
                                                                                            25
  IN
                                        rank of receiving process (integer)
            root
                                                                                           26
                                                                                            27
  IN
            comm
                                        communicator (handle)
                                                                                           28
  OUT
                                        communication request (handle)
            request
                                                                                           29
                                                                                           30
int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                                                                                           31
               void* recvbuf, const int recvcounts[], const int displs[],
                                                                                            32
               MPI_Datatype recvtype, int root, MPI_Comm comm,
               MPI_Request *request)
                                                                                           34
                                                                                           35
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                           36
               recvtype, root, comm, request, ierror)
                                                                                           37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                           38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                           39
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                            41
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                            42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            43
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                            44
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                            45
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                            46
               RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                            47
    <type> SENDBUF(*), RECVBUF(*)
```

```
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
          COMM, REQUEST, IERROR
2
         This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).
     5.12.4 Nonblocking Scatter
6
8
     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
9
                     request)
10
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
       IN
                  sendtype
                                             data type of send buffer elements (significant only at
15
                                             root) (handle)
16
17
       OUT
                  recvbuf
                                             address of receive buffer (choice)
       IN
                  recvcount
                                             number of elements in receive buffer (non-negative in-
19
                                             teger)
20
       IN
                  recvtype
                                             data type of receive buffer elements (handle)
21
22
       IN
                                             rank of sending process (integer)
                  root
23
       IN
                                             communicator (handle)
                  comm
24
       OUT
                 request
                                             communication request (handle)
25
26
27
     int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
28
29
                    MPI_Comm comm, MPI_Request *request)
30
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
31
                    root, comm, request, ierror)
32
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
34
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
35
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
39
40
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
41
                    ROOT, COMM, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
44
          IERROR
45
          This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).
46
```

```
MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,
               comm, request)
  IN
            sendbuf
                                        address of send buffer (choice, significant only at root)
  IN
            sendcounts
                                        non-negative integer array (of length group size) spec-
                                        ifying the number of elements to send to each rank
  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
                                                                                           10
  IN
            sendtype
                                        data type of send buffer elements (handle)
                                                                                           11
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                           12
  IN
            recvcount
                                        number of elements in receive buffer (non-negative in-
                                                                                           13
                                        teger)
                                                                                           14
                                                                                           15
                                        data type of receive buffer elements (handle)
  IN
            recvtype
                                                                                           16
  IN
                                        rank of sending process (integer)
            root
                                                                                           17
  IN
            comm
                                        communicator (handle)
                                                                                           18
                                                                                           19
  OUT
            request
                                        communication request (handle)
                                                                                           20
                                                                                           21
int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const
                                                                                          22
               int displs[], MPI_Datatype sendtype, void* recvbuf,
                                                                                           23
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
                                                                                          24
               MPI_Request *request)
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                           26
               recvtype, root, comm, request, ierror)
                                                                                           27
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                           28
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                           29
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                           30
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                          31
    TYPE(MPI_Datatype), INTENT(IN) ::
                                            sendtype, recvtype
                                                                                           32
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                           34
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                          35
                                                                                           36
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                          37
               RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                           38
    <type> SENDBUF(*), RECVBUF(*)
                                                                                           39
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, REQUEST, IERROR
                                                                                           41
    This call starts a nonblocking variant of MPI_SCATTERV (see Section 5.6).
                                                                                           42
                                                                                           43
                                                                                           44
```

```
5.12.5
            Nonblocking Gather-to-all
2
3
4
     MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
5
                    request)
6
                 sendbuf
       IN
                                            starting address of send buffer (choice)
       IN
                 sendcount
                                            number of elements in send buffer (non-negative inte-
                                            ger)
10
       IN
                 sendtype
                                            data type of send buffer elements (handle)
11
       OUT
                 recvbuf
                                            address of receive buffer (choice)
12
       IN
                                            number of elements received from any process (non-
                 recvcount
13
                                            negative integer)
14
15
       IN
                 recvtype
                                            data type of receive buffer elements (handle)
16
       IN
                                            communicator (handle)
                 comm
17
       OUT
                 request
                                            communication request (handle)
18
19
     int MPI_Iallgather(const void* sendbuf, int sendcount,
20
21
                    MPI_Datatype sendtype, void* recvbuf, int recvcount,
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
22
23
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
24
                    comm, request, ierror)
25
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: sendcount, recvcount
28
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) :: request
31
          INTEGER, OPTIONAL, INTENT(OUT) ::
32
33
     MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
34
                    COMM, REQUEST, IERROR)
          <type> SENDBUF(*), RECVBUF(*)
35
36
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).
38
```

```
MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm,
               request)
  IN
            sendbuf
                                       starting address of send buffer (choice)
  IN
            sendcount
                                        number of elements in send buffer (non-negative inte-
  IN
            sendtype
                                        data type of send buffer elements (handle)
  OUT
            recvbuf
                                       address of receive buffer (choice)
  IN
            recvcounts
                                       non-negative integer array (of length group size) con-
                                        taining the number of elements that are received from
                                        each process
                                                                                          12
  IN
            displs
                                       integer array (of length group size). Entry i specifies
                                                                                          13
                                        the displacement (relative to recvbuf) at which to place
                                                                                          14
                                        the incoming data from process i
                                                                                          15
                                                                                          16
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                          17
  IN
            comm
                                       communicator (handle)
                                                                                          18
  OUT
            request
                                       communication request (handle)
                                                                                          19
                                                                                          20
                                                                                          21
int MPI_Iallgatherv(const void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
                                                                                          22
                                                                                          23
               const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
                                                                                          24
               MPI_Request* request)
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
               recvtype, comm, request, ierror)
                                                                                          27
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                          28
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                          29
    INTEGER, INTENT(IN) :: sendcount
                                                                                          30
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                          31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          32
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                          34
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                          35
MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                          36
                                                                                          37
               RECVTYPE, COMM, REQUEST, IERROR)
                                                                                          38
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          39
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    REQUEST, IERROR
                                                                                          41
    This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).
                                                                                          42
                                                                                          43
                                                                                          44
```

```
5.12.6
            Nonblocking All-to-All Scatter/Gather
2
3
4
     MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)
5
6
       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)
11
       OUT
                 recvbuf
                                            address of receive buffer (choice)
12
       IN
                                            number of elements received from any process (non-
                 recvcount
13
                                            negative integer)
14
15
       IN
                 recvtype
                                            data type of receive buffer elements (handle)
16
       IN
                                            communicator (handle)
                 comm
17
       OUT
                 request
                                            communication request (handle)
18
19
     int MPI_Ialltoall(const void* sendbuf, int sendcount,
20
21
                    MPI_Datatype sendtype, void* recvbuf, int recvcount,
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
22
23
     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
24
                    comm, request, ierror)
25
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: sendcount, recvcount
28
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) :: request
31
          INTEGER, OPTIONAL, INTENT(OUT) ::
32
33
     MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
34
                    COMM, REQUEST, IERROR)
          <type> SENDBUF(*), RECVBUF(*)
35
36
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8).
38
```

```
MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls,
               recvtype, comm, request)
  IN
            sendbuf
                                        starting address of send buffer (choice)
  IN
            sendcounts
                                        non-negative integer array (of length group size) spec-
                                        ifying the number of elements to send to each rank
  IN
            sdispls
                                        integer array (of length group size). Entry i specifies
                                        the displacement (relative to sendbuf) from which to
                                        take the outgoing data destined for process i
                                                                                            10
  IN
            sendtype
                                        data type of send buffer elements (handle)
                                                                                            11
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                            12
  IN
            recvcounts
                                        non-negative integer array (of length group size) spec-
                                                                                            13
                                        ifying the number of elements that can be received
                                                                                            14
                                        from each rank
                                                                                            15
                                                                                            16
  IN
            rdispls
                                        integer array (of length group size). Entry i specifies
                                                                                            17
                                        the displacement (relative to recvbuf) at which to place
                                                                                            18
                                        the incoming data from process i
                                                                                            19
  IN
            recvtype
                                        data type of receive buffer elements (handle)
                                                                                            20
  IN
            comm
                                        communicator (handle)
                                                                                            21
                                                                                            22
  OUT
            request
                                        communication request (handle)
                                                                                            23
                                                                                            24
int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const
               int sdispls[], MPI_Datatype sendtype, void* recvbuf, const
               int recvcounts[], const int rdispls[], MPI_Datatype recvtype,
                                                                                            27
               MPI_Comm comm, MPI_Request *request)
                                                                                            28
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                            29
               rdispls, recvtype, comm, request, ierror)
                                                                                            30
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                            31
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                            32
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
                                                                                            34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                            35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            36
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                            37
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
                                                                                            38
                                                                                            39
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
               RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
                                                                                            41
     <type> SENDBUF(*), RECVBUF(*)
                                                                                            42
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                            43
    RECVTYPE, COMM, REQUEST, IERROR
                                                                                            44
    This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 5.8).
                                                                                            45
                                                                                            46
                                                                                            47
```

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

```
<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)
                                                                                         11
 IN
            sendbuf
                                       address of send buffer (choice)
                                                                                         12
 OUT
            recvbuf
                                       address of receive buffer (choice, significant only at
                                                                                         13
                                       root)
                                                                                         14
 IN
            count
                                       number of elements in send buffer (non-negative inte-
                                                                                         15
                                       ger)
                                                                                         16
 IN
                                       data type of elements of send buffer (handle)
                                                                                         17
            datatype
                                                                                         18
 IN
            op
                                       reduce operation (handle)
                                                                                         19
 IN
                                       rank of root process (integer)
            root
                                                                                         20
 IN
                                       communicator (handle)
                                                                                         21
            comm
                                                                                         22
 OUT
                                       communication request (handle)
           request
                                                                                         23
                                                                                         24
int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                                                                                         26
              MPI_Request *request)
                                                                                         27
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                         28
                                                                                         29
               ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
                                                                                         30
                                                                                         31
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                         32
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) ::
                                           datatype
                                                                                         34
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         35
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         36
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                         37
                                                                                         38
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
                                                                                         39
               IERROR)
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         41
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
                                                                                         42
    This call starts a nonblocking variant of MPI_REDUCE (see Section 5.9.1).
                                                                                         43
                                                                                         44
     Advice to implementors.
                              The implementation is explicitly allowed to use different
                                                                                         45
     algorithms for blocking and nonblocking reduction operations that might change the
                                                                                         46
     order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly
                                                                                         47
     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
2
           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 deliv-
           ered by the blocking reduction, even when specifying the same arguments in the same
           order. (End of advice to users.)
9
     5.12.8
            Nonblocking All-Reduce
10
11
12
13
     MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)
14
       IN
                 sendbuf
                                             starting address of send buffer (choice)
15
       OUT
                 recybuf
                                             starting address of receive buffer (choice)
16
17
       IN
                 count
                                             number of elements in send buffer (non-negative inte-
                                             ger)
19
       IN
                 datatype
                                             data type of elements of send buffer (handle)
20
       IN
                 ор
                                             operation (handle)
21
22
       IN
                 comm
                                             communicator (handle)
23
       OUT
                                             communication request (handle)
                 request
24
25
     int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,
26
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
27
                    MPI_Request *request)
28
29
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
30
31
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
                                                                       sendbuf
32
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
          INTEGER, INTENT(IN) :: count
34
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
          TYPE(MPI_Op), INTENT(IN) :: op
36
          TYPE(MPI_Comm), INTENT(IN) ::
37
          TYPE(MPI_Request), INTENT(OUT) ::
38
          INTEGER, OPTIONAL, INTENT(OUT) ::
39
     MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
40
                     IERROR)
41
          <type> SENDBUF(*), RECVBUF(*)
42
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
43
44
         This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6).
45
```

tion 5.10.1).

38 39

```
5.12.9 Nonblocking Reduce-Scatter with Equal Blocks
                                                                                       2
MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, request)
  IN
           sendbuf
                                      starting address of send buffer (choice)
  OUT
           recvbuf
                                      starting address of receive buffer (choice)
  IN
           recvcount
                                      element count per block (non-negative integer)
  IN
                                      data type of elements of send and receive buffers (han-
           datatype
                                                                                       11
                                      dle)
                                                                                       12
  IN
                                      operation (handle)
           op
                                                                                       13
                                                                                       14
  IN
                                      communicator (handle)
           comm
                                                                                       15
  OUT
           request
                                      communication request (handle)
                                                                                       16
                                                                                       17
int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,
               int recvcount, MPI_Datatype datatype, MPI_Op op,
                                                                                       19
              MPI_Comm comm, MPI_Request *request)
                                                                                       20
                                                                                       21
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                       22
              request, ierror)
                                                                                       23
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                       24
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       27
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                       28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       31
MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
              REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
                                                                                       34
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                       35
                                                                                       36
    This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
                                                                                       37
```

```
5.12.10 Nonblocking Reduce-Scatter
1
2
3
4
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
7
8
       IN
                 recvcounts
                                             non-negative integer array specifying the number of
9
                                             elements in result distributed to each process. Array
10
                                             must be identical on all calling processes.
11
       IN
                 datatype
                                             data type of elements of input buffer (handle)
12
       IN
                                             operation (handle)
                 op
13
14
       IN
                                             communicator (handle)
                 comm
15
       OUT
                 request
                                             communication request (handle)
16
17
     int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, const
18
                    int recvcounts[], MPI_Datatype datatype, MPI_Op op,
19
                    MPI_Comm comm, MPI_Request *request)
20
21
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
22
                    request, ierror)
23
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
24
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
31
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
32
                    REQUEST, IERROR)
33
          <type> SENDBUF(*), RECVBUF(*)
34
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
35
36
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).
37
38
39
```

5.12.11 Nonblocking Inclusive Scan

2 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 number of elements in input buffer (non-negative incount teger) IN data type of elements of input buffer (handle) datatype 11 IN operation (handle) op 12 13 IN communicator (handle) comm 14 OUT communication request (handle) request 15 16 int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, 17 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 18 MPI_Request *request) 19 20 MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) 21 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 22 23 INTEGER, INTENT(IN) :: count 24 TYPE(MPI_Datatype), INTENT(IN) :: datatype

INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

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

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

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

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

Unofficial Draft for Comment Only

37 38 39

26

27

28 29

30

31

32

34 35 36

40 41 42

43 44 45

45 46 47

36 37

38

39

40

41

42

43

44

45

46

47 48

```
5.12.12 Nonblocking Exclusive Scan
```

```
2
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                            starting address of send buffer (choice)
6
       OUT
                 recvbuf
                                            starting address of receive buffer (choice)
8
       IN
                count
                                            number of elements in input buffer (non-negative in-
9
                                            teger)
       IN
                datatype
                                            data type of elements of input buffer (handle)
11
       IN
                                            operation (handle)
                op
12
13
       IN
                                            intracommunicator (handle)
                 comm
14
       OUT
                                            communication request (handle)
                 request
15
16
     int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
17
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                    MPI_Request *request)
19
20
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
21
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
                                                                     sendbuf
22
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
23
          INTEGER, INTENT(IN) :: count
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          TYPE(MPI_Request), INTENT(OUT) :: request
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
29
     MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
31
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
32
33
         This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).
```

5.13 Persistent Collective Operations

Often a collective communication with the same argument list is repeatedly executed within a parallel computation. As with persistent point-to-point operations, persistent collective operations allow the MPI programmer to specify operations that will be reused often (with fixed arguments). MPI can be designed to select a more efficient way to perform the collective operation based on the parameters specified at initialization. This "planned-transfer" approach can offer significant performance benefits for programs with repetitive communication patterns. In terms of data movements (as observed in the output buffers), each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intracommunicators and intercommunicators after completion. Likewise, upon completion, persistent collective reduction operations have the same effect as their blocking

6

8

11

12

13

14

15 16

17

19

20

21

22

23

24

25 26

27 28 29

30 31

32

34

35 36

37

38

39

41

42 43

and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply.

MPI nonblocking collective operation initialization calls follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are deemed erroneous rather than unsafe.

The request argument is an out argument in each process of the calling group of the communicator that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Locally starting the operation makes the request active. Once a process starts a persistent collective operation, all other processes in the communicator must eventually start the same persistent collective operation. Persistent collective calls cannot be matched with blocking or nonblocking collective calls. All processes in the group must complete the operation. Local completion of the operation makes the corresponding request inactive.

Once initialized, persistent collective operations can be started in any order. A request must be inactive when it is started.

Advice to users. This allows MPI_STARTALL to be used without any new requirements. (End of advice to users.)

Completing a persistent collective request, for example using MPI_TEST or MPI_WAIT, does not free the request. This is the same behavior as for persistent point-to-point requests. As with persistent point-to-point operations, MPI_REQUEST_FREE can be used to free an inactive persistent collective request.

For every nonblocking collective communication operation in MPI, as a matter of design, there is a corresponding persistent collective operation with the analogous API signature.

5.13.1 Persistent Barrier Synchronization Request

```
MPI_BARRIER_INIT(comm, request)
 IN
                                     communicator (handle)
           comm
 OUT
           request
                                     communication request (handle)
int MPI_Barrier_init(MPI_Comm comm, MPI_Request *request)
MPI_Barrier_init(comm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::
    TYPE(MPI_Request), INTENT(OUT) ::
                                         request
    INTEGER, OPTIONAL, INTENT(OUT) ::
MPI_BARRIER_INIT(COMM, REQUEST, IERROR)
    INTEGER COMM, REQUEST, IERROR
    Creates a persistent collective communication request for the barrier operation.
```

```
5.13.2 Persistent Broadcast Request
2
3
4
     MPI_BCAST_INIT(buffer, count, datatype, root, comm, request)
5
       INOUT
                  buffer
                                             starting address of buffer (choice)
6
       IN
                 count
                                             number of entries in buffer (non-negative integer)
7
8
                                             data type of buffer (handle)
       IN
                 datatype
9
       IN
                 root
                                             rank of broadcast root (integer)
       IN
                                             communicator (handle)
                 comm
11
12
       OUT
                                             communication request (handle)
                 request
13
14
     int MPI_Bcast_init(void* buffer, int count, MPI_Datatype datatype,
15
                    int root, MPI_Comm comm, MPI_Request *request)
16
     MPI_Bcast_init(buffer, count, datatype, root, comm, request, ierror)
17
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
          INTEGER, INTENT(IN) :: count, root
19
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
20
          TYPE(MPI_Comm), INTENT(IN) :: comm
21
          TYPE(MPI_Request), INTENT(OUT) ::
                                                 request
22
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
23
24
     MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
25
          <type> BUFFER(*)
26
          INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
27
          Creates a persistent collective communication request for the broadcast operation.
28
29
30
31
32
34
35
36
37
38
39
41
42
43
44
45
```

5.13.3 Persistent Gather Request MPI_GATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request) IN sendbuf starting address of send buffer (choice) IN sendcount number of elements in send buffer (non-negative integer) IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf address of receive buffer (choice, significant only at 12 root) 13 IN recvcount number of elements for any single receive (non-negative 14 integer, significant only at root) 15 16 IN data type of recv buffer elements (significant only at recvtype 17 root) (handle) 18 IN rank of receiving process (integer) root 19 IN comm communicator (handle) 20 21 OUT request communication request (handle) 22 23 int MPI_Gather_init(const void* sendbuf, int sendcount, 24 MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request) 27 MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 28 root, comm, request, ierror) 29 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: 30 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 31 INTEGER, INTENT(IN) :: sendcount, recvcount, root 32 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm 34 TYPE(MPI_Request), INTENT(OUT) :: 35 INTEGER, OPTIONAL, INTENT(OUT) :: 36 37 MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 38 ROOT, COMM, REQUEST, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 41 **IERROR** 42 Creates a persistent collective communication request for the gather operation. 43

```
MPI_GATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root,
1
                     comm, request)
2
3
       IN
                  sendbuf
                                              starting address of send buffer (choice)
       IN
                  sendcount
                                              number of elements in send buffer (non-negative inte-
                                              ger)
6
       IN
                  sendtype
7
                                              data type of send buffer elements (handle)
       OUT
                  recvbuf
                                              address of receive buffer (choice, significant only at
9
                                              root)
       IN
                  recvcounts
                                              non-negative integer array (of length group size) con-
11
                                              taining the number of elements that are received from
12
                                              each process (significant only at root)
13
       IN
                  displs
                                              integer array (of length group size). Entry i specifies
14
                                              the displacement relative to recvbuf at which to place
15
16
                                              the incoming data from process i (significant only at
                                              root)
17
18
       IN
                                              data type of recv buffer elements (significant only at
                  recvtype
19
                                              root) (handle)
20
       IN
                                              rank of receiving process (integer)
                  root
21
       IN
                                              communicator (handle)
22
                  comm
23
       OUT
                 request
                                              communication request (handle)
24
25
     int MPI_Gatherv_init(const void* sendbuf, int sendcount,
26
                     MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
27
                     const int displs[], MPI_Datatype recvtype, int root,
28
                     MPI_Comm comm, MPI_Request *request)
29
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
30
31
                     recvtype, root, comm, request, ierror)
32
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
          INTEGER, INTENT(IN) :: sendcount, root
34
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
35
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Request), INTENT(OUT) ::
38
          INTEGER, OPTIONAL, INTENT(OUT) ::
39
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
41
                     RECVTYPE, ROOT, COMM, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
44
          COMM, REQUEST, IERROR
45
46
          Creates a persistent collective communication request for the gathery operation.
```

```
5.13.4 Persistent Scatter Request
MPI_SCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
               request)
  IN
            sendbuf
                                        address of send buffer (choice, significant only at root)
  IN
            sendcount
                                        number of elements sent to each process (non-negative
                                        integer, significant only at root)
  IN
            sendtype
                                        data type of send buffer elements (significant only at
                                                                                          11
                                        root) (handle)
                                                                                          12
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                          13
  IN
            recvcount
                                        number of elements in receive buffer (non-negative in-
                                                                                          14
                                                                                          15
                                                                                          16
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                          17
  IN
                                        rank of sending process (integer)
            root
                                                                                          18
  IN
            comm
                                        communicator (handle)
                                                                                          19
                                                                                          20
  OUT
            request
                                        communication request (handle)
                                                                                          21
                                                                                          22
int MPI_Scatter_init(const void* sendbuf, int sendcount,
                                                                                          23
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                          24
               MPI_Datatype recvtype, int root, MPI_Comm comm,
                                                                                          25
               MPI_Request *request)
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                          27
               recvtype, root, comm, request, ierror)
                                                                                          28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                          29
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                          30
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                          31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          32
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                          34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          35
                                                                                          36
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                          37
               RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                          38
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          39
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR
                                                                                          41
    Creates a persistent collective communication request for the scatter operation.
                                                                                          42
```

```
MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype,
1
                     root, comm, request)
2
3
       IN
                  sendbuf
                                             address of send buffer (choice, significant only at root)
       IN
                  sendcounts
                                             non-negative integer array (of length group size) spec-
                                             ifying the number of elements to send to each rank
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
       IN
                  sendtype
                                             data type of send buffer elements (handle)
11
       OUT
                  recvbuf
                                             address of receive buffer (choice)
12
       IN
                  recvcount
                                             number of elements in receive buffer (non-negative in-
13
                                             teger)
14
15
       IN
                                             data type of receive buffer elements (handle)
                  recvtype
16
       IN
                  root
                                             rank of sending process (integer)
17
       IN
18
                  comm
                                             communicator (handle)
19
       OUT
                 request
                                             communication request (handle)
20
21
     int MPI_Scatterv_init(const void* sendbuf, const int sendcounts[], const
22
                     int displs[], MPI_Datatype sendtype, void* recvbuf,
23
                     int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
24
                    MPI_Request *request)
25
     MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
26
27
                    recvcount, recvtype, root, comm, request, ierror)
28
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
29
          INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
30
31
          INTEGER, INTENT(IN) :: recvcount, root
32
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                  sendtype, recvtype
          TYPE(MPI_Comm), INTENT(IN) :: comm
34
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
35
36
     MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
37
                    RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR)
38
          <type> SENDBUF(*), RECVBUF(*)
39
          INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
          COMM, REQUEST, IERROR
41
42
          Creates a persistent collective communication request for the scattery operation.
```

```
5.13.5 Persistent Gather-to-all Request
MPI_ALLGATHER_INIT(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 inte-
                                       ger)
  IN
            sendtype
                                       data type of send buffer elements (handle)
                                                                                         11
  OUT
            recvbuf
                                       address of receive buffer (choice)
                                                                                         12
                                       number of elements received from any process (non-
  IN
            recvcount
                                                                                         13
                                       negative integer)
                                                                                         14
                                                                                         15
  IN
            recvtype
                                       data type of receive buffer elements (handle)
                                                                                         16
  IN
                                       communicator (handle)
            comm
                                                                                         17
  OUT
           request
                                       communication request (handle)
                                                                                         18
                                                                                         19
int MPI_Allgather_init(const void* sendbuf, int sendcount,
                                                                                         20
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                         21
               MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                         22
                                                                                         23
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                         24
               recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                         27
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                         28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                         29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         30
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         31
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                         32
MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
               RECVTYPE, COMM, REQUEST, IERROR)
                                                                                         34
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         35
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                         36
                                                                                         37
    Creates a persistent collective communication request for the allgather operation.
                                                                                         38
```

```
MPI_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype,
1
                    comm, request)
2
3
       IN
                  sendbuf
                                             starting address of send buffer (choice)
       IN
                  sendcount
                                             number of elements in send buffer (non-negative inte-
                                             ger)
6
       IN
                  sendtype
                                             data type of send buffer elements (handle)
       OUT
                  recvbuf
                                             address of receive buffer (choice)
       IN
                  recvcounts
                                             non-negative integer array (of length group size) con-
10
                                             taining the number of elements that are received from
11
                                             each process
12
       IN
                  displs
                                             integer array (of length group size). Entry i specifies
13
                                             the displacement (relative to recvbuf) at which to place
14
                                             the incoming data from process i
15
16
       IN
                  recvtype
                                             data type of receive buffer elements (handle)
17
       IN
                  comm
                                             communicator (handle)
18
       OUT
                 request
                                             communication request (handle)
19
20
21
     int MPI_Allgatherv_init(const void* sendbuf, int sendcount,
22
                    MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
23
                    const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
24
                    MPI_Request* request)
25
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
26
                    displs, recvtype, comm, request, ierror)
27
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
28
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
29
          INTEGER, INTENT(IN) :: sendcount
30
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
31
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
          TYPE(MPI_Comm), INTENT(IN) :: comm
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
34
          INTEGER, OPTIONAL, INTENT(OUT) ::
35
36
     MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
37
                    DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
38
          <type> SENDBUF(*), RECVBUF(*)
39
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
          REQUEST, IERROR
41
          Creates a persistent collective communication request for the allgathery operation.
42
43
44
```

```
5.13.6 Persistent All-to-All Scatter/Gather Request
MPI_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, re-
               quest)
            sendbuf
  IN
                                       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)
                                                                                         11
  OUT
            recvbuf
                                       address of receive buffer (choice)
                                                                                         12
  IN
                                       number of elements received from any process (non-
            recvcount
                                                                                         13
                                       negative integer)
                                                                                         14
                                                                                         15
  IN
            recvtype
                                       data type of receive buffer elements (handle)
                                                                                         16
  IN
                                       communicator (handle)
            comm
                                                                                         17
  OUT
           request
                                       communication request (handle)
                                                                                         18
                                                                                         19
int MPI_Alltoall_init(const void* sendbuf, int sendcount,
                                                                                         20
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                         21
               MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                         22
                                                                                         23
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                         24
               recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                         27
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                         28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                         29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         30
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         31
    INTEGER, OPTIONAL, INTENT(OUT) ::
MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
               RECVTYPE, COMM, REQUEST, IERROR)
                                                                                         34
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         35
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                         36
                                                                                         37
    Creates a persistent collective communication request for the alltoall operation.
                                                                                         38
```

```
MPI_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls,
1
                     recvtype, comm, request)
2
3
       IN
                  sendbuf
                                              starting address of send buffer (choice)
       IN
                  sendcounts
                                              non-negative integer array (of length group size) spec-
                                              ifying the number of elements to send to each rank
6
       IN
                  sdispls
                                              integer array (of length group size). Entry i specifies
                                              the displacement (relative to sendbuf) from which to
                                              take the outgoing data destined for process i
9
10
                                              data type of send buffer elements (handle)
       IN
                  sendtype
11
       OUT
                  recvbuf
                                              address of receive buffer (choice)
12
       IN
                  recvcounts
                                              non-negative integer array (of length group size) spec-
13
                                              ifying the number of elements that can be received
14
                                              from each rank
15
16
       IN
                  rdispls
                                              integer array (of length group size). Entry i specifies
17
                                              the displacement (relative to recvbuf) at which to place
                                              the incoming data from process i
19
       IN
                  recvtype
                                              data type of receive buffer elements (handle)
20
21
       IN
                  comm
                                              communicator (handle)
22
       OUT
                 request
                                              communication request (handle)
23
24
     int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[], const
25
                     int sdispls[], MPI_Datatype sendtype, void* recvbuf, const
26
                     int recvcounts[], const int rdispls[], MPI_Datatype recvtype,
27
                     MPI_Comm comm, MPI_Request *request)
28
29
     MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                     recvcounts, rdispls, recvtype, comm, request, ierror)
30
31
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
32
          INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
          recvcounts(*), rdispls(*)
34
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
35
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
          TYPE(MPI_Request), INTENT(OUT) ::
37
                                                   request
          INTEGER, OPTIONAL, INTENT(OUT) ::
38
39
     MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
40
                     RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
41
          <type> SENDBUF(*), RECVBUF(*)
42
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
43
          RECVTYPE, COMM, REQUEST, IERROR
44
45
          Creates a persistent collective communication request for the alltoally operation.
```

```
MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls,
                recvtypes, comm, request)
  IN
            sendbuf
                                         starting address of send buffer (choice)
  IN
            sendcounts
                                         integer array (of length group size) specifying the num-
                                         ber of elements to send to each rank (array of non-
                                         negative integers)
  IN
            sdispls
                                         integer array (of length group size). Entry i specifies
                                         the displacement in bytes (relative to sendbuf) from
                                         which to take the outgoing data destined for process i
                                         (array of integers)
                                                                                              11
                                                                                              12
  IN
            sendtypes
                                         array of datatypes (of length group size). Entry i spec-
                                                                                              13
                                         ifies the type of data to send to process i (array of
                                         handles)
                                                                                              14
                                                                                              15
  OUT
            recvbuf
                                         address of receive buffer (choice)
                                                                                              16
  IN
            recvcounts
                                         integer array (of length group size) specifying the num-
                                                                                              17
                                         ber of elements that can be received from each rank
                                         (array of non-negative integers)
                                                                                              19
  IN
            rdispls
                                         integer array (of length group size). Entry i specifies
                                                                                              20
                                         the displacement in bytes (relative to recvbuf) at which
                                                                                              21
                                         to place the incoming data from process i (array of
                                                                                              22
                                         integers)
                                                                                              23
                                                                                              24
  IN
                                         array of datatypes (of length group size). Entry i spec-
            recvtypes
                                                                                              25
                                         ifies the type of data received from process i (array of
                                                                                              26
                                         handles)
                                                                                              27
  IN
            comm
                                         communicator (handle)
                                                                                              28
  OUT
            request
                                         communication request (handle)
                                                                                              29
                                                                                              30
int MPI_Alltoallw_init(const void* sendbuf, const int sendcounts[], const
                                                                                              31
                int sdispls[], const MPI_Datatype sendtypes[], void* recvbuf,
                                                                                              32
                const int recvcounts[], const int rdispls[], const
                MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)
                                                                                              34
                                                                                              35
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                              36
                recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                              37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                              38
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                              39
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
                                                                                              41
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                              42
    recvtypes(*)
                                                                                              43
    TYPE(MPI_Comm), INTENT(IN) ::
                                                                                              44
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                              45
     INTEGER, OPTIONAL, INTENT(OUT) ::
                                              ierror
                                                                                              46
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                              47
                RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
2
          RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
          Creates a persistent collective communication request for the alltoally operation.
6
     5.13.7 Persistent Reduce Request
8
9
     MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, request)
10
11
       IN
                 sendbuf
                                            address of send buffer (choice)
12
       OUT
                 recvbuf
                                            address of receive buffer (choice, significant only at
13
                                            root)
14
       IN
                 count
                                            number of elements in send buffer (non-negative inte-
15
                                            ger)
16
17
       IN
                                            data type of elements of send buffer (handle)
                 datatype
       IN
                                            reduce operation (handle)
                 op
19
       IN
                                            rank of root process (integer)
                 root
20
21
       IN
                 comm
                                            communicator (handle)
22
       OUT
                                            communication request (handle)
                 request
23
24
     int MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,
25
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
26
                    MPI_Request *request)
27
28
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, request,
29
                    ierror)
30
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
32
          INTEGER, INTENT(IN) :: count, root
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
          TYPE(MPI_Op), INTENT(IN) :: op
35
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
          TYPE(MPI_Request), INTENT(OUT) :: request
37
          INTEGER, OPTIONAL, INTENT(OUT) ::
38
     MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
39
                    IERROR)
40
          <type> SENDBUF(*), RECVBUF(*)
41
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
42
43
          Creates a persistent collective communication request for the reduce operation.
44
```

```
5.13.8 Persistent All-Reduce Request
                                                                                        2
MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, request)
  IN
           sendbuf
                                      starting address of send buffer (choice)
  OUT
           recvbuf
                                      starting address of receive buffer (choice)
  IN
                                      number of elements in send buffer (non-negative inte-
           count
                                      ger)
  IN
           datatype
                                      data type of elements of send buffer (handle)
                                                                                       11
  IN
                                      operation (handle)
           op
                                                                                       12
                                                                                       13
  IN
                                      communicator (handle)
           comm
                                                                                       14
  OUT
                                      communication request (handle)
           request
                                                                                       15
                                                                                       16
int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count,
                                                                                       17
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
              MPI_Request *request)
                                                                                       19
                                                                                       20
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                       21
              ierror)
                                                                                       22
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                       23
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                       24
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       26
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                       27
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       28
                                                                                       29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       30
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
                                                                                       31
              IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                       34
```

Creates a persistent collective communication request for the all reduce operation.

```
5.13.9 Persistent Reduce-Scatter with Equal Blocks Request
2
3
4
     MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm, re-
5
                    quest)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
8
9
       IN
                                            element count per block (non-negative integer)
                 recvcount
       IN
                                            data type of elements of send and receive buffers (han-
                datatype
11
                                            dle)
12
       IN
                                            operation (handle)
                op
13
14
       IN
                                            communicator (handle)
                comm
15
       OUT
                 request
                                            communication request (handle)
16
17
     int MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,
18
                    int recvcount, MPI_Datatype datatype, MPI_Op op,
19
                    MPI_Comm comm, MPI_Request *request)
20
21
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
22
                    comm, request, ierror)
23
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN) :: recvcount
26
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
31
     MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
32
                    COMM, REQUEST, IERROR)
33
          <type> SENDBUF(*), RECVBUF(*)
34
          INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
35
36
          Creates a persistent collective communication request for the reduce-scatter with equal
37
     blocks operation.
38
```

Unofficial Draft for Comment Only

```
5.13.10 Persistent Reduce-Scatter Request
MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
           sendbuf
  IN
                                       starting address of send buffer (choice)
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
  IN
                                       non-negative integer array specifying the number of
           recvcounts
                                       elements in result distributed to each process. Array
                                                                                        11
                                       must be identical on all calling processes.
                                                                                        12
  IN
                                       data type of elements of input buffer (handle)
           datatype
                                                                                        13
  IN
                                       operation (handle)
           op
                                                                                        14
                                                                                        15
  IN
           comm
                                       communicator (handle)
                                                                                        16
  OUT
           request
                                       communication request (handle)
                                                                                        17
int MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf, const
               int recvcounts[], MPI_Datatype datatype, MPI_Op op,
                                                                                        20
               MPI_Comm comm, MPI_Request *request)
                                                                                        21
                                                                                        22
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                        23
               request, ierror)
                                                                                        24
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                        27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        28
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        30
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
               REQUEST, IERROR)
                                                                                        34
    <type> SENDBUF(*), RECVBUF(*)
                                                                                        35
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                        36
                                                                                        37
    Creates a persistent collective communication request for the reduce-scatter operation.
```

```
5.13.11 Persistent Inclusive Scan Request
1
2
3
4
     MPI_SCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
7
8
       IN
                                             number of elements in input buffer (non-negative in-
                 count
9
                                             teger)
       IN
                 datatype
                                             data type of elements of input buffer (handle)
11
       IN
                                             operation (handle)
                 op
12
13
       IN
                                             communicator (handle)
                 comm
14
       OUT
                                             communication request (handle)
                 request
15
16
     int MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,
17
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                    MPI_Request *request)
19
20
     MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
21
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
22
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
23
          INTEGER, INTENT(IN) :: count
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          TYPE(MPI_Request), INTENT(OUT) :: request
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
29
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
31
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
32
          Creates a persistent collective communication request for the inclusive scan operation.
34
35
36
37
38
39
41
42
43
44
45
46
```

12 13

14

15 16

17

18

19 20

21

22

23

24

27

28 29

30

31

32

34 35

36 37

38 39

41

42 43

44

45 46 47

5.13.12 Persistent Exclusive Scan Request MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, request) IN sendbuf starting address of send buffer (choice) OUT recvbuf starting address of receive buffer (choice) IN number of elements in input buffer (non-negative incount teger) IN datatype data type of elements of input buffer (handle) IN operation (handle) op IN intracommunicator (handle) comm OUT communication request (handle) request int MPI_Exscan_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request) MPI_Exscan_init(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) :: MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)

5.14 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.

Creates a persistent collective communication request for the exclusive scan operation.

INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

Example 5.25

The following is erroneous.

```
switch(rank) {
1
         case 0:
2
             MPI_Bcast(buf1, count, type, 0, comm);
             MPI_Bcast(buf2, count, type, 1, comm);
             break:
         case 1:
6
             MPI_Bcast(buf2, count, type, 1, comm);
             MPI_Bcast(buf1, count, type, 0, comm);
             break;
9
     }
10
11
```

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.

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

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

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

Example 5.28

An unsafe, non-deterministic program.

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

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

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

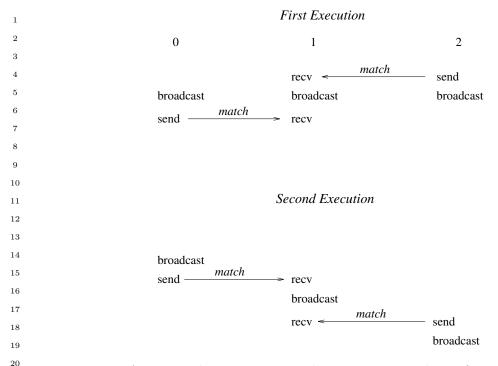


Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

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.

```
MPI_Request req;

MPI_Ibarrier(comm, &req);

MPI_Bcast(buf1, count, type, 0, comm);

MPI_Wait(&req, MPI_STATUS_IGNORE);
```

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

Example 5.30

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.

This ordering would match MPI_lbarrier 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:

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

48

}

break;

```
MPI_Wait(&req, MPI_STATUS_IGNORE);
1
              break;
2
     }
3
4
           Advice to users. The use of different communicators offers some flexibility regarding
5
           the matching of nonblocking collective operations. In this sense, communicators could
6
           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
10
          Nonblocking collective operations can rely on the same progression rules as nonblocking
11
     point-to-point messages. Thus, if started with two processes, the following program is a
12
     valid MPI program and is guaranteed to terminate:
13
14
     MPI_Request req;
15
16
     switch(rank) {
17
          case 0:
18
            MPI_Ibarrier(comm, &req);
19
            MPI_Wait(&req, MPI_STATUS_IGNORE);
20
            MPI_Send(buf, count, dtype, 1, tag, comm);
21
            break;
22
          case 1:
23
            MPI_Ibarrier(comm, &req);
24
            MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
            MPI_Wait(&req, MPI_STATUS_IGNORE);
26
            break;
27
     }
28
         The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait
29
     call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls
30
     eventually return.
31
32
     Example 5.32
33
          Blocking and nonblocking collective operations do not match. The following example
34
     is erroneous.
35
     MPI_Request req;
36
37
     switch(rank) {
38
          case 0:
39
            /* erroneous false matching of Alltoall and Ialltoall */
            MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
41
42
            MPI_Wait(&req, MPI_STATUS_IGNORE);
            break;
43
          case 1:
44
            /* erroneous false matching of Alltoall and Ialltoall */
45
            MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
46
```

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

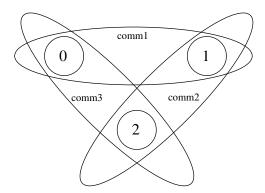


Figure 5.13: Example with overlapping communicators.

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];
22
23
     switch(rank) {
24
         case 0:
25
           MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
26
           MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
27
           break;
28
         case 1:
29
           MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
30
           MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
31
           break;
32
         case 2:
           MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
34
           MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
35
           break;
36
37
     MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
38
```

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.

```
MPI_Request reqs[2];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
/* nothing is known about the status of the first bcast here */
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. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [2] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. 5.12
- [3] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007. 5.12
- [4] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [5] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University Dept. of Computer Science, April 1994. http://www.erc.msstate.edu/mpi/mpix.html. 5.2.2

Index

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

INDEX

1	CONST:MPI_Op, 34, <u>43</u> , 45, 47, 49–51, 53,	EXAMPLES:MPI_Iallreduce, 97
2	54, 69–74, 86–91	EXAMPLES:MPI_Ialltoall, 96
3	CONST:MPI_OP_NULL, 46	EXAMPLES:MPI_Ibarrier, 94–97
4	CONST:MPI_PROC_NULL, 6, 8, 10, 12, 20,	EXAMPLES:MPI_Ibcast, 59, 97, 98
5	22, 35	EXAMPLES:MPI_Irecv, 97
6	CONST:MPI_PROD, 36, 37	EXAMPLES:MPI_Op_create, 46, 47, 55
7	CONST:MPI_REAL, 36	EXAMPLES:MPI_Recv, 96
8	CONST:MPI_REAL16, 37	EXAMPLES:MPI_REDUCE, 37, 38, 41
9	CONST:MPI_REAL2, 37	EXAMPLES:MPI_Reduce, 40, 42, 46, 47
10	CONST:MPI_REAL4, 37	EXAMPLES:MPI_Scan, 55
11	CONST:MPI_REAL8, 37	EXAMPLES:MPI_Scatter, 22
12	CONST:MPI_ROOT, 6	EXAMPLES:MPI_Scattery, 22, 23
13	CONST:MI I_ROOT, 0	EXAMPLES:MPI_Send, 96, 97
14	CONST:MI I_SHORT_INT, 40	EXAMPLES:MPI_Type_commit, 13–17, 23,
	CONST:MI1_SHORT_INT, 40 CONST:MPI_SIGNED_CHAR, 36, 38, 39	55
15	CONST:MI LSIGNED_CHAR, 30, 38, 39 CONST:MPI_SOURCE, 57	EXAMPLES:MPI_Type_contiguous, 13
16	CONST:MFI_SOURCE, 37 CONST:MPI_SUM, 36, 37	
17	, ,	EXAMPLES:MPI_Type_create_struct, 16, 17, 55
18	CONST:MPI_TAG, 57 CONST:MPI_UINT16_T, 36	
19	,	EXAMPLES:MPI_Type_vector, 14, 15, 17,
20	CONST:MPI_UINT32_T, 36	23 EXAMPLES:MPI_User_function, 47
21	CONST:MPI_UINT64_T, 36	,
22	CONST:MPI_UINT8_T, 36	EXAMPLES:MPI_Wait, 94–97
23	CONST:MPI_UNSIGNED, 36	EXAMPLES:MPI_Waitall, 97
24	CONST:MPI_UNSIGNED_CHAR, 36, 38, 39	9
25	CONST:MPI_UNSIGNED_LONG, 36	Nonblocking collective operations, 96
26	•	EXAMPLES:Non-deterministic program with
27	CONST:MPI_UNSIGNED_SHORT, 36	MPI_Bcast, 93
28	CONST:MPI_WCHAR, 38, 39	EXAMPLES: Overlapping Communicators, 97
29	CONST:Multi-language types:, 37	EXAMPLES:Pipelining nonblocking collec-
30	CONST:Name, 36, 40	tive operations, 97
31	CONST:Op, 37	EXAMPLES:Progression of nonblocking col-
32	EXAMPLES:Deadlock	lective operations, 96
33	with MPI_Bcast, 91, 92	MPI_ABORT, 44
34	EXAMPLES: False matching of collective op-	MPI_ALLGATHER, 1, 5, 6, 25–28, 64
35	erations, 95	MPI_ALLGATHER(), 25
36	•	MPI_ALLGATHER(sendbuf, sendcount, send-
37	erations, 98	type, recvbuf, recvcount, recvtype,
38	EXAMPLES: Mixing blocking and nonblock-	comm), $\underline{25}$
39	ing collective operations, 94	MPI_ALLGATHER_INIT(sendbuf, sendcount,
40	EXAMPLES: Mixing collective and point-to-	sendtype, recvbuf, recvcount, recv-
41	point requests, 97	type, comm, request), <u>81</u>
42	EXAMPLES:MPI_Allgather, 27	MPI_ALLGATHERV, 1, 5, 6, 27, 65
43	EXAMPLES:MPI_ALLREDUCE, 48	MPI_ALLGATHERV(sendbuf, sendcount, send-
44	EXAMPLES:MPI_Alltoall, 96	type, recvbuf, recvcounts, displs, recv-
45	EXAMPLES:MPI_Bcast, 9, 91–95	type, comm), $\frac{26}{}$
46	EXAMPLES:MPI_Gather, 12, 13, 17	MPI_ALLGATHERV_INIT(sendbuf, sendcount,
47	EXAMPLES:MPI_Gathery, 14–17	sendtype, recvbuf, recvcounts, dis-
48		solida, po, isonodi, ioonodiib, dib

INDEX 103

pls, recvtype, comm, request), <u>82</u>	MPI_GATHER_INIT(sendbuf, sendcount, send-	. 1
MPI_ALLREDUCE, 1, 4–6, 36, 43, 48, 70	type, recvbuf, recvcount, recvtype,	2
MPI_ALLREDUCE(sendbuf, recvbuf, count,	root, comm, request), $\overline{77}$	3
datatype, op, comm), 47	MPI_GATHERV, 1, 5, 6, 11–13, 21, 27, 62	4
MPI_ALLREDUCE_INIT(sendbuf, recvbuf,	MPI_GATHERV(sendbuf, sendcount, send-	5
count, datatype, op, comm, request),	type, recvbuf, recvcounts, displs, recv-	6
87	type, root, comm), $\underline{11}$	7
MPI_ALLTOALL, 1, 5, 6, 28–31, 66	MPI_GATHERV_INIT(sendbuf, sendcount,	8
MPI_ALLTOALL(sendbuf, sendcount, send-	sendtype, recvbuf, recvcounts, dis-	9
type, recvbuf, recvcount, recvtype,	pls, recvtype, root, comm, request)	10
comm), 28	$\frac{1}{1}$, $\frac{78}{1}$	11
MPI_ALLTOALL_INIT(sendbuf, sendcount,	MPI_IALLGATHER, 1, 5, 6	12
sendtype, recvbuf, recvcount, recv-		13
type, comm, request), <u>83</u>	type, recvbuf, recvcount, recvtype,	14
MPI_ALLTOALLV, 1, 5, 6, 30, 31, 33, 67	comm, request), 64	15
MPI_ALLTOALLV(sendbuf, sendcounts, sdis-		16
· · · · · · · · · · · · · · · · · · ·	MPI_IALLGATHERV(sendbuf, sendcount, send	-17
rdispls, recvtype, comm), 30	type, recvbuf, recvcounts, displs, recv-	18
MPI_ALLTOALLV_INIT(sendbuf, sendcount		19
sdispls, sendtype, recvbuf, recvcounts,	, , , , , , , , , , , , , , , , , , , ,	20
	MPI_IALLREDUCE(sendbuf, recvbuf, count,	21
MPI_ALLTOALLW, 1, 5, 6, 33, 69	datatype, op, comm, request), 70	22
MPI_ALLTOALLW(sendbuf, sendcounts, sdis		23
	MPI_IALLTOALL(sendbuf, sendcount, send-	24
rdispls, recvtypes, comm), <u>32</u>	type, recvbuf, recvcount, recvtype,	25
MPI_ALLTOALLW_INIT(sendbuf, sendcount		26
sdispls, sendtypes, recybuf, recycounts	· ·	27
	MPI_IALLTOALLV(sendbuf, sendcounts, sdis-	28
85	pls, sendtype, recvbuf, recvcounts,	29
MPI_BARRIER, 1, 5, 7, 8, 58	rdispls, recvtype, comm, request), 67	30
MPI_BARRIER(comm), 7	MPI_IALLTOALLW, 1, 5, 6	31
MPI_BARRIER_INIT(comm, request), 75	MPI_IALLTOALLW(sendbuf, sendcounts, sdis-	
MPI_BCAST, 1, 5, 8, 9, 35, 59, 95	pls, sendtypes, recvbuf, recvcounts,	33
MPI_BCAST(buffer, count, datatype, root,	rdispls, recvtypes, comm, request),	34
comm), $\underline{8}$	68	35
MPI_BCAST_INIT(buffer, count, datatype,	MPI_IBARRIER, 1, 5, 57–59, 95	36
root, comm, request), <u>76</u>	MPI_IBARRIER(comm, request), 58	37
MPI_CANCEL, 57	MPI_IBCAST, 1, 5, 59, 99	38
MPI_EXSCAN, 2, 5, 36, 43, 54, 74	MPI_IBCAST(buffer, count, datatype, root,	39
MPI_EXSCAN(sendbuf, recybuf, count, datat		40
op, comm), 54	MPI_IEXSCAN, 2, 5	
- / / · · · · · · · · · · · · · · · · ·	,MPI_IEXSCAN(sendbuf, recvbuf, count, dataty	41
datatype, op, comm, request), 91	op, comm, request), $\frac{74}{}$	₽ #2 :
MPI_GATHER, 1, 4–6, 11, 12, 19, 20, 25,	MPI_IGATHER, 1, 5, 6	
		44
35, 61 MPI CATHER (condbuf condcount condition	MPI_IGATHER(sendbuf, sendcount, sendtype,	45
MPI_GATHER(sendbuf, sendcount, sendtype	recvbuf, recvcount, recvtype, root, comm, request), <u>60</u>	46
recvbuf, recvcount, recvtype, root,	· - / /	47
comm), $\underline{9}$	MPI_IGATHERV, 1, 5, 6	48

INDEX

```
MPI_IGATHERV(sendbuf, sendcount, send-
                                                       recvcounts, datatype, op, comm), 51
             type, recvbuf, recvcounts, displs, recv-MPI_REDUCE_SCATTER_BLOCK, 1, 5, 6,
2
3
             type, root, comm, request), 61
                                                       36, 43, 50, 51, 71
     MPI_IREDUCE, 1, 5, 6, 69
                                               MPI_REDUCE_SCATTER_BLOCK( send-
4
     MPI_IREDUCE(sendbuf, recvbuf, count, datatype,
                                                       buf, recvbuf, recvcount, datatype, op,
5
            op, root, comm, request), 69
                                                       comm), 50
6
     MPI_IREDUCE_SCATTER, 1, 5, 6
                                               MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf,
7
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf,
                                                       recvbuf, recvcount, datatype, op, comm,
8
            recvcounts, datatype, op, comm, re-
                                                       request), 88
9
             quest), 72
                                               MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf,
10
     MPI_IREDUCE_SCATTER_BLOCK, 1, 5,
                                                       recvcounts, datatype, op, comm, re-
11
                                                       quest), 89
12
     MPI_IREDUCE_SCATTER_BLOCK(sendbufMPI_REQUEST_FREE, 57, 75
13
            recvbuf, recvcount, datatype, op, comnMPI_SCAN, 2, 5, 36, 43, 53, 55, 73
14
                                               MPI_SCAN(sendbuf, recvbuf, count, datatype,
            request), 71
15
     MPI_ISCAN, 2, 5
16
                                                       op, comm), 53
     MPI_ISCAN(sendbuf, recvbuf, count, datatypeMPI_SCAN_INIT(sendbuf, recvbuf, count,
17
             op, comm, request), 73
                                                       datatype, op, comm, request), 90
18
     MPI_ISCATTER, 1, 5, 6
                                               MPI_SCATTER, 1, 5, 6, 19, 21, 22, 51, 62
19
     MPI_ISCATTER(sendbuf, sendcount, send-
                                               MPI_SCATTER(sendbuf, sendcount, send-
20
             type, recvbuf, recvcount, recvtype,
                                                       type, recvbuf, recvcount, recvtype,
21
             root, comm, request), 62
                                                       root, comm), 19
22
     MPI_ISCATTERV, 1, 5, 6
                                               MPI_SCATTER_INIT(sendbuf, sendcount, send-
23
     MPI_ISCATTERV(sendbuf, sendcounts, dis-
                                                       type, recvbuf, recvcount, recvtype,
24
             pls, sendtype, recvbuf, recvcount, recv-
                                                       root, comm, request), 79
25
             type, root, comm, request), 63
                                               MPI_SCATTERV, 1, 5, 6, 21, 22, 52, 63
26
     MPI_OP_COMMUTATIVE( op, commute),
                                               MPI_SCATTERV(sendbuf, sendcounts, dis-
27
             49
                                                       pls, sendtype, recvbuf, recvcount, recv-
28
     MPI_OP_CREATE, 43, 45
                                                       type, root, comm), 21
29
     MPI_OP_CREATE(user_fn, commute, op),
                                               MPI_SCATTERV_INIT(sendbuf, sendcounts,
30
             43
                                                       displs, sendtype, recvbuf, recvcount,
31
     MPI_OP_FREE(op), \underline{45}
                                                       recvtype, root, comm, request), 80
32
                                               MPI_SEND, 96
     MPI_RECV, 2, 96
33
                                               MPI_START, 75
     MPI_RECV(recvbuf, recvcount·n, recvtype,
34
                                               MPI_STARTALL, 75
            ...), 10
35
     MPI_REDUCE, 1, 5, 6, 34–36, 43–45, 48,
                                               MPI_TEST, 75
36
                                               MPI_TYPE_CREATE_F90_COMPLEX, 37
            51–54. 69
37
     MPI_REDUCE(sendbuf, recvbuf, count, datatyMPI_TYPE_CREATE_F90_INTEGER, 36
38
                                               MPI_TYPE_CREATE_F90_REAL, 37
             op, root, comm), 34
39
     MPI_REDUCE_INIT(sendbuf, recvbuf, count, MPI_TYPE_CREATE_STRUCT, 33
40
             datatype, op, root, comm, request),
                                               MPI_WAIT, 57, 75, 96
41
                                                MPI_WAITALL, 57, 97
42
             86
     MPI_REDUCE_LOCAL, 35, 36, 43
43
     MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, YPEDEF: MPI_User_function, 47
44
                                                TYPEDEF:MPI_User_function(void* invec,
             datatype, op), 49
45
                                                       void* inoutvec, int *len, MPI_Datatype
     MPI_REDUCE_SCATTER, 1, 5, 6, 36, 43,
46
                                                       *datatype), 43
             51, 52, 72
47
     MPI_REDUCE_SCATTER( sendbuf, recvbuf,
48
```