### 4.9.2 PREDEFINED REDUCE OPERATIONS

The following predefined operations are supplied for MPI.REDUCE and related functions MPI.ALLREDUCE, MPI.REDUCE.SCATTER, and MPI.SCAN. These operations are invoked by placing the following in `op`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI.MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MPI.MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MPI.SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI.PROD</td>
<td>product</td>
</tr>
<tr>
<td>MPI.LAND</td>
<td>logical and</td>
</tr>
<tr>
<td>MPI.BAND</td>
<td>bit-wise and</td>
</tr>
<tr>
<td>MPI.LOR</td>
<td>logical or</td>
</tr>
<tr>
<td>MPI.BOR</td>
<td>bit-wise or</td>
</tr>
<tr>
<td>MPI.LXOR</td>
<td>logical xor</td>
</tr>
<tr>
<td>MPI.BXOR</td>
<td>bit-wise xor</td>
</tr>
<tr>
<td>MPI.MAXLOC</td>
<td>max value and location</td>
</tr>
</tbody>
</table>
The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Sec. 4.9.3. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

| C integer:       | MPI_INT, MPI_LONG, MPI_SHORT,          |
|                 | MPI_UNSIGNED_SHORT, MPI_UNSIGNED,     |
|                 | MPI_UNSIGNED_LONG                     |
| Fortran integer:| MPI_INTEGER                          |
| Floating point: | MPI_FLOAT, MPI_DOUBLE, MPI_REAL,       |
|                 | MPI_DOUBLE_PRECISION,                 |
|                 | MPI_LONG_DOUBLE                       |
| Logical:        | MPI_LOGICAL                           |
| Complex:        | MPI_COMPLEX                           |
| Byte:           | MPI_BYTE                              |

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

<table>
<thead>
<tr>
<th>Op</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX, MPI_MIN</td>
<td>C integer, Fortran integer, Floating point</td>
</tr>
<tr>
<td>MPI_SUM, MPI_PROD</td>
<td>C integer, Fortran integer, Floating point</td>
</tr>
<tr>
<td>MPI_BAND, MPI_BOR, MPI_BXOR</td>
<td>C integer, Logical</td>
</tr>
<tr>
<td></td>
<td>C integer, Fortran integer, Byte</td>
</tr>
</tbody>
</table>

Example 4.15 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.

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

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

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, conn, ierr)
RETURN
```
Example 4.16 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at node zero.

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

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

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

  ! return result at node zero (and garbage at the other nodes)
  RETURN
END SUBROUTINE PAR_BLAS2
```

### 4.9.3 MINLOC AND MAXLOC

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

The operation that defines MPI.MAXLOC is:

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

where

\[ w = \max(u, v) \]

and

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

MPI.MINLOC is defined similarly:

\[
\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}
\]
where
\[ w = \min(u, v) \]
and
\[
k = \begin{cases} 
i & \text{if } u < v \\
\min(i, j) & \text{if } u = v \\
j & \text{if } u > v \\
\end{cases}
\]

Both operations are associative and commutative. Note that if \texttt{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 \(\text{op} = \text{MPI\_MAXLOC}\) will return the maximum value and the rank of the first process with that value. Similarly, \texttt{MPI\_MINLOC} can be used to return a minimum and its index. More generally, \texttt{MPI\_MINLOC} computes a \textit{lexicographic minimum}, where elements are ordered according to the first component of each pair, and ties are resolved according to the second component.

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

In order to use \texttt{MPI\_MINLOC} and \texttt{MPI\_MAXLOC} in a reduce operation, one must provide a \texttt{datatype} argument that represents a pair (value and index). MPI provides seven such predefined datatypes. The operations \texttt{MPI\_MAXLOC} and \texttt{MPI\_MINLOC} can be used with each of the following datatypes.

**Fortran:**

- Name
  - \texttt{MPI\_2REAL}
  - \texttt{MPI\_2DOUBLE\_PRECISION}
  - \texttt{MPI\_2INTEGER}

- Description
  - pair of \texttt{REALS}
  - pair of \texttt{DOUBLE \_PRECISION} variables
  - pair of \texttt{INTEGERs}

**C:**

- Name
  - \texttt{MPI\_FLOAT\_INT}
  - \texttt{MPI\_DOUBLE\_INT}
  - \texttt{MPI\_LONG\_INT}
  - \texttt{MPI\_2INT}
  - \texttt{MPI\_SHORT\_INT}

- Description
  - float and int
  - double and int
  - long and int
  - pair of int
  - short and int
The datatype MPI_2REAL is as if defined by the following (see Section 3.12).

MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)

Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.

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

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

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

**Example 4.17** Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

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

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

Example 4.18 Same example, in Fortran.

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

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

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

MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root, ierr);

! At this point, the answer resides on process root

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

Example 4.19 Each process has a non-empty array of values. Find the minimum
global value, the rank of the process that holds it and its index on this process.

#define LEN 1000

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

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

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

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

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

4.9.4 USER-DEFINED OPERATIONS

MPI_OP_CREATE{ function, commute, op}

IN function user-defined function (function)
IN commute true if commutative; false otherwise.
OUT op operation (handle)
int MPI.Op.create(MPI_User_function *function, int commute, MPI.Op *op)

MPI.OP.CREATE( FUNCTION, COMMUTE, OP, IERROR)
EXTERNA Function
LOGICAL COMMUT
INTEGER OP, IERROR

MPI.OP.CREATE binds a user-defined global operation to an op handle that can subsequently be used in MPI.REDUCE, MPI.ALLREDUCE, MPI.REDUCE_SCATTER, and MPI.Scan. 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 operations is fixed and is defined to be in ascending, process rank order, beginning with process zero.

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

The ANSI-C prototype for the function is the following.

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

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

FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE

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

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

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

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies. No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

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

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

Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE.

```c
if (rank > 0) {
   _RECV(tempbuf, count, datatype, rank-1,...)
    User_reduce( tempbuf, sendbuf, count, datatype)
}
if (rank < groupsize-1) {
    _SEND( sendbuf, count, datatype, rank+1, ...)
}
/* answer now resides in process groupsize-1 ... now send to root */
if (rank == groupsize-1) {
    _SEND( sendbuf, count, datatype, root, ...)
}
if (rank == root) {
    _RECV(recvbuf, count, datatype, groupsize-1,...)
}
```

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 \( \text{len} \leq \text{count} \).

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.)

\[
\text{MPI.OP.FREE( op)} \\
\text{IN op operation (handle)} \\
\]

\[
\text{int MPI.op.free( MPIOp *op)} \\
\text{MPI.OP.FREE( OP, IERROR)} \\
\text{INTEGER OP, IERROR} \\
\]

Marks a user-defined reduction operation for deallocation and sets op to MPI.OP.NULL.

**Example of User-defined Reduce**

It is time for an example of user-defined reduction.

**Example 4.20** Compute the product of an array of complex numbers, in C.

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

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

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

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

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

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

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

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

4.9.5 ALL-REDUCE

MPI includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results.

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

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

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

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

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.)

Example 4.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 4.16).

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

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

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

! return result at all nodes
RETURN

4.10 Reduce-Scatter

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

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)
   IN     sendbuf      starting address of send buffer (choice)
   OUT    recvbuf      starting address of receive buffer (choice)
   IN     recvcounts   integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, 
                        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(SENDBUF,RecvBUF,RECVCOUNTS,DATATYPE,OP,COMM,IERRO

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

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally 
equivalent to: A MPI_REDUCE operation function 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.)

4.11 Scan

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

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

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

MPI.SCAN(SENDBUF,_RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*),_RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

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, \ldots, i \text{ (inclusive)}. The type of operations supported, their semantics, and the constraints on send and receive buffers are as for \texttt{MPI\_REDUCE}.

\textit{Rationale.} We have defined an inclusive scan, that is, the prefix reduction on process $i$ includes the data from process $i$. An alternative is to define scan in an exclusive manner, where the result on $i$ only includes data up to $i-1$. Both definitions are useful. The latter has some advantages: the inclusive scan can always be computed from the exclusive scan with no additional communication; for non-invertible operations such as max and min, communication is required to compute the exclusive scan from the inclusive scan. There is, however, a complication with exclusive scan since one must define the “unit” element for the reduction in this case. That is, one must explicitly say what occurs for process 0. This was thought to be complex for user-defined operations and hence, the exclusive scan was dropped. (End of rationale.)

\subsection{Example Using \texttt{MPI\_SCAN}}

\textbf{Example 4.22} This example uses a user-defined operation to produce a \textit{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:

\begin{verbatim}
values v1 v2 v3 v4 v5 v6 v7 v8
logicals 0 0 1 1 1 0 0 1
result v1 v1+v2 v3 v3+v4 v5+v1+v5 v6 v6+v7 v8
\end{verbatim}

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 )
{


```c
int i;
SegScanPair c;

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

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

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

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

### 4.12 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.
Example 4.23 The following is erroneous.

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

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

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

Example 4.24 The following is erroneous.

```c
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.

Example 4.25 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);
        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 4.26** A correct, but 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);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm);
        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 4.10. Note that the second execution has
First Execution

\[
\begin{array}{ccc}
\text{process:} & 0 & 1 & 2 \\
\text{broadcast} & & \text{recv} \leftarrow \text{match} & \text{send} \\
\text{send} \rightarrow \text{match} & \text{broadcast} & \text{broadcast} \\
\text{recv} & & \text{broadcast} \\
\end{array}
\]

Second Execution

\[
\begin{array}{ccc}
\text{broadcast} & & \text{recv} \\
\text{send} \rightarrow \text{match} & \text{broadcast} & \text{recv} \leftarrow \text{match} \\
\text{recv} & & \text{broadcast} \\
\end{array}
\]

Fig. 4.10 **A race condition causes non-deterministic matching of sends and receives.** One cannot rely on synchronization from a broadcast to make the program deterministic.

the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multi-threaded 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 messages 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.)

Redefined Reduce Operations

Predefined operations are supplied by MPI_REDUCE and related constructs—MPI_ALLREDUCE, MPI_REDUCE_SCATTER, and MPI_SCAN. These operations are invoked by placing the following in op:

```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDUCE</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>SUM</td>
<td>sum</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>product</td>
</tr>
<tr>
<td>MAXLOC</td>
<td>maximum value and location</td>
</tr>
<tr>
<td>MAXLOC</td>
<td>maximum value and location</td>
</tr>
<tr>
<td>AND</td>
<td>logical and</td>
</tr>
<tr>
<td>OR</td>
<td>logical or</td>
</tr>
<tr>
<td>XOR</td>
<td>bit-wise or</td>
</tr>
<tr>
<td>BITXOR</td>
<td>bit-wise xor</td>
</tr>
<tr>
<td>BITXOR</td>
<td>max value and location</td>
</tr>
<tr>
<td>PRED</td>
<td>logical and</td>
</tr>
<tr>
<td>PRED</td>
<td>logical or</td>
</tr>
</tbody>
</table>
```

Redefined reduce operations may also be defined by any of the standard operators such as logical and, logical or, bit-wise and, bit-wise or, max, and max value and location. These operations may be defined using any of the standard operators such as logical and, logical or, bit-wise and, bit-wise or, max, and max value and location.