and can be also used in initialization expressions or assignments.

The constants that are required to be compile-time constants (and can thus be used for array length declarations and labels in C/C++ switch and Fortran case/select statements) are:

MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)

and their C++ counterparts where appropriate.

The constants that cannot be used in initialization expressions or assignments in Fortran are:

MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE
MPI_ARGV_NULL
MPI_ARGVS_NULL
MPI_UNWEIGHTED

ticket33.

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through PARAMETER statements) is not possible because an implementation cannot distinguish these values from legal data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (*End of advice to implementors.*)

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses <type> to represent a choice variable; for C and C++, we use void *.

2.5.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C, MPI::Aint in C++

The MPI interface provides four communicator construction routines that apply to both intracommunicators and intercommunicators. The construction routine MPI_INTERCOMM_CREATE (discussed later) applies only to intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local* group; the other group (relative to that process) is the *remote* group. The left and right group labels give us a way to describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are).

MPI_COMM_DUP(comm, newcomm)

IN	comm	communicator (handle)
OUT	newcomm	copy of comm (handle)

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

{MPI::Intracomm MPI::Intracomm::Dup() const (binding deprecated, see Section 15.2)
              }

{MPI::Intercomm MPI::Intercomm::Dup() const (binding deprecated, see Section 15.2)
              }

{MPI::Cartcomm MPI::Cartcomm::Dup() const (binding deprecated, see Section 15.2) }

{MPI::Graphcomm MPI::Graphcomm::Dup() const (binding deprecated, see Section 15.2)
              }

{MPI::Distgraphcomm MPI::Distgraphcomm::Dup() const (binding deprecated, see
              Section 15.2) }

{MPI::Comm& MPI::Comm::Clone() const = 0 (binding deprecated, see Section 15.2) }

{MPI::Intracomm& MPI::Intracomm::Clone() const (binding deprecated, see
              Section 15.2) }

{MPI::Intercomm& MPI::Intercomm::Clone() const (binding deprecated, see
              Section 15.2) }

{MPI::Cartcomm& MPI::Cartcomm::Clone() const (binding deprecated, see
              Section 15.2) }

{MPI::Graphcomm& MPI::Graphcomm::Clone() const (binding deprecated, see
              Section 15.2) }

{MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const (binding deprecated, see
              Section 15.2) }
```

ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket33. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket150. ticket33. ticket150. ticket150.

notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. [Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

ticket33.

Rationale. Experience with similar techniques in PARMACS MPI-2.1 Correction due to Reviews to MPI-2.1 draft Feb.23, 2008 [5, 9] MPI-2.1 End of review based correction show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

]

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem then that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord (0,0): rank 0
coord (0,1): rank 1
coord (1,0): rank 2
coord (1,1): rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE, MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_DIST_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not. ticket33.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. [MPI-2.1 Ballots 1-4 All input arguments must have identical values on all processes of the group of comm_old. A MPI-2.1 Ballots 1-4] For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. For MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE the input communication graph is distributed across the calling processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol. ticket33.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

> *Rationale.* Similar functions are contained in EXPRESS [12] and PARMACS. (*End of rationale.*)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. For distributed graphs, the functions MPI_DIST_NEIGHBORS_COUNT and MPI_DIST_NEIGHBORS can be used to extract the neighbors of the calling node. The function MPI_CART_SUB is collective over the input communicator's group; all other functions are local. ticket33.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes = 4
index = 2, 3, 4, 6
edges = 1, 3, 0, 3, 0, 2

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for $0 \le j \le$ `index`$[0] - 1$ and the list of neighbors of node `i`, `i` > 0, is stored in `edges[j]`, `index`$[i-1] \le j \le$ `index`$[i] - 1$.

In Fortran, `index(1)` is the degree of node zero, and `index(i+1) - index(i)` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges(j)`, for $1 \le j \le$ `index`(1) and the list of neighbors of node `i`, `i` > 0, is stored in `edges(j)`, `index`$(i) + 1 \le j \le$ `index`$(i + 1)$.

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  1. `ndims` (number of dimensions),
  2. `dims` (numbers of processes per coordinate direction),
  3. `periods` (periodicity information),
  4. `own_position` (own position in grid, could also be computed from rank and dims)
- For a graph topology:
  1. `index`,
  2. `edges`,

  which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (*End of advice to implementors.*)

ticket33.

7.5.4 Distributed (Graph) Constructor

The general graph constructor assumes that each process passes the full (global) communication graph to the call. This limits the scalability of this constructor. With the distributed graph interface, the communication graph is specified in a fully distributed fashion. Each process specifies only the part of the communication graph of which it is aware. Typically, this could be the set of processes from which the process will eventually receive or get data, or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with each process specifying all of its incoming and outgoing (adjacent) edges in the logical communication graph and thus requires minimal communication during creation. MPI_DIST_GRAPH_CREATE provides full flexibility, and processes can indicate that communication will occur between other pairs of processes.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

IN	comm_old	input communicator (handle)
IN	indegree	size of sources and sourceweights arrays (non-negative integer)
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
IN	outdegree	size of destinations and destweights arrays (non-negative integer)
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
IN	info	hints on optimization and interpretation of weights (handle)
IN	reorder	the ranks may be reordered (true) or not (false) (logical)
OUT	comm_dist_graph	communicator with distributed graph topology (handle)

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
              int sources[], int sourceweights[], int outdegree,
```

```
              int destinations[], int destweights[], MPI_Info info,
              int reorder, MPI_Comm *comm_dist_graph)

MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
              OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
              COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

ticket150.

```
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int
              indegree, const int sources[], const int sourceweights[],
              int outdegree, const int destinations[],
              const int destweights[], const MPI::Info& info, bool reorder)
              const (binding deprecated, see Section 15.2) }
```

ticket150.

ticket150.

```
{MPI::Distgraphcomm
              MPI::Intracomm::Dist_graph_create_adjacent(int indegree,
              const int sources[], int outdegree, const int destinations[],
              const MPI::Info& info, bool reorder) const (binding deprecated,
              see Section 15.2) }
```

ticket150.

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about the edges to its neighbors in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication topology is the combination of all edges shown in the sources arrays of all processes in comm_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and that thus do not occur as source or destination rank in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. In C++, this constant does not exist and the weights argument may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or in C++ omit the weight arrays, for some but not all processes of comm_old. Note that

MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph)

IN	comm_old	input communicator (handle)
IN	n	number of source nodes for which this process specifies edges (non-negative integer)
IN	sources	array containing the n source nodes for which this process specifies edges (array of non-negative integers)
IN	degrees	array specifying the number of destinations for each source node in the source node array (array of non-negative integers)
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)
IN	weights	weights for source to destination edges (array of non-negative integers)
IN	info	hints on optimization and interpretation of weights (handle)
IN	reorder	the process may be reordered (true) or not (false) (logical)
OUT	comm_dist_graph	communicator with distributed graph topology added (handle)

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
              int degrees[], int destinations[], int weights[],
              MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
              INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER

{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[], const int
              destinations[], const int weights[], const MPI::Info& info,
              bool reorder) const (binding deprecated, see Section 15.2) }

{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[],
              const int destinations[], const MPI::Info& info, bool reorder)
              const (binding deprecated, see Section 15.2) }
```

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source,destination) communication edges as described below. Every process passes an array of n source nodes in the sources array. For each source node, a non-negative number of destination nodes is specified in the degrees array. The destination nodes are stored in the corresponding consecutive segment of the destinations array. More precisely if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the j-th such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge is stored in weights[degrees[0]+...+degrees[i-1]+j]. Both the sources and the destinations arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must be process ranks of comm_old. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_Dist_graph_create is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in comm_old. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. In C++, this constant does not exist and the weights argument may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or in C++ omit the weight arrays, for some but not all processes of comm_old. Note that MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total array argument. In C, one would expect it to be a NULL. In Fortran, MPI_UNWEIGHTED is an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is legal for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally

supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

Example 7.3 As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	1,1,1,1,1,1
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

In both cases above, the application could supply MPI_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT could be used to specify this graph using the following arguments:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

Example 7.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modelled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

```
/*
Input:     dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
           ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* get x and y dimension */
y=rank/P; x=rank%P;

/* get my communication partners along x dimension */
destinations[0] = P*y+(x+1)%P; weights[0] = 2;
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;

/* get my communication partners along y dimension */
destinations[2] = P*((y+1)%Q)+x; weights[3] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[4] = 2;

/* get my communication partners along diagonals */
destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[5] = 1;
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[6] = 1;
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[7] = 1;
destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[8] = 1;

sources[0] = rank;
degrees[0] = 8;
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
                      weights, MPI_INFO_NULL, 1, comm_dist_graph)
```

7.5.5 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator `comm` (state)

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

ticket150.
ticket150. {`int MPI::Comm::Get_topology() const` *(binding deprecated, see Section 15.2)* }

The function MPI_TOPO_TEST returns the type of topology that is assigned to a communicator.

The output value `status` is one of the following:

MPI_GRAPH	graph topology
MPI_DIST_GRAPH	distributed graph topology
MPI_CART	Cartesian topology
MPI_UNDEFINED	no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	comm	communicator for group with graph structure (handle)
OUT	nnodes	number of nodes in graph (integer) (same as number of processes in the group)
OUT	nedges	number of edges in graph (integer)

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

ticket150.
ticket150. {`void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const` *(binding deprecated, see Section 15.2)* }

Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE.

The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors `index` and `edges` correctly for the following call to MPI_GRAPH_GET.

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

Example 7.6 Suppose that `comm` is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

node		exchange neighbors(1)	shuffle neighbors(2)	unshuffle neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator `comm` has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology. ticket33.

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

IN	comm	communicator with distributed graph topology (handle)
OUT	indegree	number of edges into this process (non-negative integer)
OUT	outdegree	number of edges out of this process (non-negative integer)
OUT	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical)

```
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
              int *outdegree, int *weighted)
```

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED
```

ticket150.

ticket150.

```
{void MPI::Distgraphcomm::Get_dist_neighbors_count(int rank,
              int indegree[], int outdegree[], bool& weighted) const (binding
              deprecated, see Section 15.2) }
```

MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)

IN	comm	communicator with distributed graph topology (handle)
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)

```
int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
              int sourceweights[], int maxoutdegree, int destinations[],
              int destweights[])
```

```
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
              MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
```

```
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR
```

ticket150.

{`void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[])` *(binding deprecated, see Section 15.2)* }

ticket150.

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. The only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBOR_COUNT, then only the first part of the full list is returned. Note, that the order of returned edges does need not to be identical to the order that was provided in the creation of comm for the case that MPI_DIST_GRAPH_CREATE_ADJACENT was used.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

16.1.3 C++ Classes for MPI

All MPI classes, constants, and functions are declared within the scope of an `MPI` `namespace`. Thus, instead of the `MPI_` prefix that is used in C and Fortran, MPI functions essentially have an `MPI::` prefix.

The members of the `MPI` namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the `MPI` namespace and its member classes is as follows:

```
namespace MPI {
  class Comm                              {...};
  class Intracomm : public Comm           {...};
  class Graphcomm : public Intracomm      {...};
[ticket33.]  class Distgraphcomm : public Intracomm {...};
  class Cartcomm  : public Intracomm      {...};
  class Intercomm : public Comm           {...};
  class Datatype                          {...};
  class Errhandler                        {...};
  class Exception                         {...};
  class File                              {...};
  class Group                             {...};
  class Info                              {...};
  class Op                                {...};
  class Request                           {...};
  class Prequest  : public Request        {...};
  class Grequest  : public Request        {...};
  class Status                            {...};
  class Win                               {...};
};
```

Note that there are a small number of derived classes, and that virtual inheritance is *not* used.

16.1.4 Class Member Functions for MPI

Besides the member functions which constitute the C++ language bindings for MPI, the C++ language interface has additional functions (as required by the C++ language). In particular, the C++ language interface must provide a constructor and destructor, an assignment operator, and comparison operators.

The complete set of C++ language bindings for MPI is presented in Annex A.4. The bindings take advantage of some important C++ features, such as references and `const`. Declarations (which apply to all `MPI` member classes) for construction, destruction, copying, assignment, comparison, and mixed-language operability are also provided.

Except where indicated, all non-static member functions (except for constructors and the assignment operator) of `MPI` member classes are virtual functions.

Rationale. Providing virtual member functions is an important part of design for inheritance. Virtual functions can be bound at run-time, which allows users of libraries to re-define the behavior of objects already contained in a library. There is a small

Op	Allowed Types
MPI::MAX, MPI::MIN	C integer, Fortran integer, Floating point
MPI::SUM, MPI::PROD	C integer, Fortran integer, Floating point, Complex
MPI::LAND, MPI::LOR, MPI::LXOR	C integer, Logical
MPI::BAND, MPI::BOR, MPI::BXOR	C integer, Fortran integer, Byte

MPI::MINLOC and MPI::MAXLOC perform just as their C and Fortran counterparts; see Section 5.9.4 on page 171.

16.1.7 Communicators

The MPI::Comm class hierarchy makes explicit the different kinds of communicators implicitly defined by MPI and allows them to be strongly typed. Since the original design of MPI defined only one type of handle for all types of communicators, the following clarifications are provided for the C++ design.

ticket33. Types of communicators There are [five]six different types of communicators: MPI::Comm, ticket33. MPI::Intercomm, MPI::Intracomm, MPI::Cartcomm, [and] MPI::Graphcomm, and ticket33. MPI::Distgraphcomm. MPI::Comm is the abstract base communicator class, encapsulating the functionality common to all MPI communicators. MPI::Intercomm and ticket33. MPI::Intracomm are derived from MPI::Comm. MPI::Cartcomm[and], MPI::Graphcomm, ticket33. and MPI::Distgraphcomm are derived from MPI::Intracomm.

Advice to users. Initializing a derived class with an instance of a base class is not legal in C++. For instance, it is not legal to initialize a Cartcomm from an Intracomm. Moreover, because MPI::Comm is an abstract base class, it is non-instantiable, so that it is not possible to have an object of class MPI::Comm. However, it is possible to have a reference or a pointer to an MPI::Comm.

Example 16.4 The following code is erroneous.

```
Intracomm intra = MPI::COMM_WORLD.Dup();
Cartcomm cart(intra);          // This is erroneous
```

(*End of advice to users.*)

MPI::COMM_NULL The specific type of MPI::COMM_NULL is implementation dependent. MPI::COMM_NULL must be able to be used in comparisons and initializations with all types of communicators. MPI::COMM_NULL must also be able to be passed to a function that expects a communicator argument in the parameter list (provided that MPI::COMM_NULL is an allowed value for the communicator argument).

Rationale. There are several possibilities for implementation of MPI::COMM_NULL. Specifying its required behavior, rather than its realization, provides maximum flexibility to implementors. (*End of rationale.*)

Example 16.5 The following example demonstrates the behavior of assignment and comparison using MPI::COMM_NULL.

```
MPI::Intercomm comm;
comm = MPI::COMM_NULL;            // assign with COMM_NULL
if (comm == MPI::COMM_NULL)       // true
  cout << "comm is NULL" << endl;
if (MPI::COMM_NULL == comm)       // note -- a different function!
  cout << "comm is still NULL" << endl;
```

`Dup()` is not defined as a member function of `MPI::Comm`, but it is defined for the derived classes of `MPI::Comm`. `Dup()` is not virtual and it returns its OUT parameter by value.

`MPI::Comm::Clone()` The C++ language interface for MPI includes a new function `Clone()`. `MPI::Comm::Clone()` is a pure virtual function. For the derived communicator classes, `Clone()` behaves like `Dup()` except that it returns a new object by reference. The `Clone()` functions are prototyped as follows:

```
Comm& Comm::Clone() const = 0

Intracomm& Intracomm::Clone() const

Intercomm& Intercomm::Clone() const

Cartcomm& Cartcomm::Clone() const

Graphcomm& Graphcomm::Clone() const

Distgraphcomm& Distgraphcomm::Clone() const
```

Rationale. `Clone()` provides the "virtual dup" functionality that is expected by C++ programmers and library writers. Since `Clone()` returns a new object by reference, users are responsible for eventually deleting the object. A new name is introduced rather than changing the functionality of `Dup()`. (*End of rationale.*)

Advice to implementors. Within their class declarations, prototypes for `Clone()` and `Dup()` would look like the following:

```
namespace MPI {
  class Comm {
     virtual Comm& Clone() const = 0;
  };
  class Intracomm : public Comm {
     Intracomm Dup() const { ... };
     virtual Intracomm& Clone() const { ... };
  };
  class Intercomm : public Comm {
     Intercomm Dup() const { ... };
     virtual Intercomm& Clone() const { ... };
  };
  // Cartcomm[ticket33.][ and], Graphcomm[ticket33.], and Distgraphcomm are similarly c
};
```

(*End of advice to implementors.*)

MPI defines two levels of Fortran support, described in Sections 16.2.3 and 16.2.4. A third level of Fortran support is envisioned, but is deferred to future standardization efforts. In the rest of this section, "Fortran" shall refer to Fortran 90 (or its successor) unless qualified.

1. **Basic Fortran Support** An implementation with this level of Fortran support provides the original Fortran bindings specified in MPI-1, with small additional requirements specified in Section 16.2.3.

2. **Extended Fortran Support** An implementation with this level of Fortran support provides Basic Fortran Support plus additional features that specifically support Fortran 90, as described in Section 16.2.4.

A compliant MPI-2 implementation providing a Fortran interface must provide Extended Fortran Support unless the target compiler does not support modules or KIND-parameterized types.

16.2.2 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail. It supersedes and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran 90, not Fortran 77).

The following MPI features are inconsistent with Fortran 90.

1. An MPI subroutine with a choice argument may be called with different argument types.

2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.

3. Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.

4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls.

5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_UNWEIGHTED, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.

ticket33.

Topologies

[ticket107.]C type: `const int` (or unnamed `enum`) [ticket107.]Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_GRAPH	MPI::GRAPH
[ticket33.]MPI_DIST_GRAPH	[ticket33.]MPI::DIST_GRAPH
MPI_CART	MPI::CART

ticket107.

Predefined functions

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_COMM_NULL_COPY_FN `MPI_Comm_copy_attr_function` / `COMM_COPY_ATTR_FN`	MPI_COMM_NULL_COPY_FN same as in C [1])
MPI_COMM_DUP_FN `MPI_Comm_copy_attr_function` / `COMM_COPY_ATTR_FN`	MPI_COMM_DUP_FN same as in C [1])
MPI_COMM_NULL_DELETE_FN `MPI_Comm_delete_attr_function` / `COMM_DELETE_ATTR_FN`	MPI_COMM_NULL_DELETE_FN same as in C [1])
MPI_WIN_NULL_COPY_FN `MPI_Win_copy_attr_function` / `WIN_COPY_ATTR_FN`	MPI_WIN_NULL_COPY_FN same as in C [1])
MPI_WIN_DUP_FN `MPI_Win_copy_attr_function` / `WIN_COPY_ATTR_FN`	MPI_WIN_DUP_FN same as in C [1])
MPI_WIN_NULL_DELETE_FN `MPI_Win_delete_attr_function` / `WIN_DELETE_ATTR_FN`	MPI_WIN_NULL_DELETE_FN same as in C [1])
MPI_TYPE_NULL_COPY_FN `MPI_Type_copy_attr_function` / `TYPE_COPY_ATTR_FN`	MPI_TYPE_NULL_COPY_FN same as in C [1])
MPI_TYPE_DUP_FN `MPI_Type_copy_attr_function` / `TYPE_COPY_ATTR_FN`	MPI_TYPE_DUP_FN same as in C [1])
MPI_TYPE_NULL_DELETE_FN `MPI_Type_delete_attr_function` / `TYPE_DELETE_ATTR_FN`	MPI_TYPE_NULL_DELETE_FN same as in C [1])

[1] See the advice to implementors on MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2 on page 241

File Operation Constants, Part 2

[ticket107.]C type: `const int` (or unnamed `enum`) [ticket107.]Fortran type: `INTEGER`	[ticket107.]C++ type: [ticket107.]`const int` (or unnamed `enum`)
MPI_DISTRIBUTE_BLOCK	MPI::DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG
MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE
MPI_ORDER_C	MPI::ORDER_C
MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN
MPI_SEEK_CUR	MPI::SEEK_CUR
MPI_SEEK_END	MPI::SEEK_END
MPI_SEEK_SET	MPI::SEEK_SET

F90 Datatype Matching Constants

[ticket107.]C type: `const int` (or unnamed `enum`) [ticket107.]Fortran type: `INTEGER`	[ticket107.]C++ type: [ticket107.]`const int` (or unnamed `enum`)
MPI_TYPECLASS_COMPLEX	MPI::TYPECLASS_COMPLEX
MPI_TYPECLASS_INTEGER	MPI::TYPECLASS_INTEGER
MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL

ticket107.

[

Handles to Assorted Structures in C and C++ (no Fortran)

MPI_File	MPI::File
MPI_Info	MPI::Info
MPI_Win	MPI::Win

]

Constants Specifying Empty or Ignored Input

[ticket107.]C/Fortran name [ticket107.]C type / Fortran type	[ticket107.]C++ name [ticket107.]C++ type
MPI_ARGVS_NULL	MPI::ARGVS_NULL
[ticket107.]`char***` / 2-dim. array of `CHARACTER*(*)`	[ticket107.]`const char ***`
MPI_ARGV_NULL	MPI::ARGV_NULL
[ticket107.]`char**` / array of `CHARACTER*(*)`	[ticket107.]`const char **`
MPI_ERRCODES_IGNORE	Not defined for C++
[ticket107.]`int*` / `INTEGER` array	
MPI_STATUSES_IGNORE	Not defined for C++
[ticket107.]`MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE,*)`	
MPI_STATUS_IGNORE	Not defined for C++
[ticket107.]`MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE)`	
[ticket33.]MPI_UNWEIGHTED	[ticket33.]Not defined for C++

C Constants Specifying Ignored Input (no C++ or Fortran)
[ticket107.]C type: `MPI_Fint*`
MPI_F_STATUSES_IGNORE
MPI_F_STATUS_IGNORE

C and C++ preprocessor Constants and Fortran Parameters
[ticket107.]C/C++ type: `const int` (or unnamed `enum`)
[ticket107.]Fortran type: `INTEGER`
MPI_SUBVERSION
MPI_VERSION

A.1.2 Types

The following are defined C type definitions, included in the file `mpi.h`.

```
/* C opaque types */
MPI_Aint
MPI_Fint
MPI_Offset
MPI_Status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Op
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)
MPI::Aint
MPI::Offset
MPI::Status

// C++ handles to assorted structures (classes,
// all within the MPI namespace)
MPI::Comm
MPI::Intracomm
MPI::Graphcomm
MPI::Distgraphcomm
MPI::Cartcomm
MPI::Intercomm
MPI::Datatype
MPI::Errhandler
MPI::Exception
```

ticket33.

```
            Datatype& sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], const Datatype& recvtype, int root)
            const = 0 (binding deprecated, see Section 15.2) }

  {void Op::Init(User_function* function, bool commute) (binding deprecated,
            see Section 15.2) }

  {bool Op::Is_commutative() const (binding deprecated, see Section 15.2) }

  {void Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
            const Datatype& datatype, const Op& op, int root) const = 0
            (binding deprecated, see Section 15.2) }

  {void Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,
            const Datatype& datatype) const (binding deprecated, see
            Section 15.2) }

  {void Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf,
            int recvcount, const Datatype& datatype, const Op& op)
            const = 0 (binding deprecated, see Section 15.2) }

  {void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
            int recvcounts[], const Datatype& datatype, const Op& op)
            const = 0 (binding deprecated, see Section 15.2) }

  {void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
            const Datatype& datatype, const Op& op) const (binding
            deprecated, see Section 15.2) }

  {void Comm::Scatter(const void* sendbuf, int sendcount, const
            Datatype& sendtype, void* recvbuf, int recvcount,
            const Datatype& recvtype, int root) const = 0 (binding
            deprecated, see Section 15.2) }

  {void Comm::Scatterv(const void* sendbuf, const int sendcounts[],
            const int displs[], const Datatype& sendtype, void* recvbuf,
            int recvcount, const Datatype& recvtype, int root) const = 0
            (binding deprecated, see Section 15.2) }

};
```

A.4.4 Groups, Contexts, Communicators, and Caching C++ Bindings

```
namespace MPI {

  {Comm& Comm::Clone() const = 0 (binding deprecated, see Section 15.2) }

  {Cartcomm& Cartcomm::Clone() const (binding deprecated, see Section 15.2) }

  {Distgraphcomm& Distgraphcomm::Clone() const (binding deprecated, see
            Section 15.2) }

  {Graphcomm& Graphcomm::Clone() const (binding deprecated, see Section 15.2) }
```

{Intercomm& Intercomm::Clone() const *(binding deprecated, see Section 15.2)* }

{Intracomm& Intracomm::Clone() const *(binding deprecated, see Section 15.2)* }

{static int Comm::Compare(const Comm& comm1, const Comm& comm2) *(binding deprecated, see Section 15.2)* }

{static int Group::Compare(const Group& group1, const Group& group2) *(binding deprecated, see Section 15.2)* }

{Intercomm Intercomm::Create(const Group& group) const *(binding deprecated, see Section 15.2)* }

{Intracomm Intracomm::Create(const Group& group) const *(binding deprecated, see Section 15.2)* }

{Intercomm Intracomm::Create_intercomm(int local_leader, const Comm& peer_comm, int remote_leader, int tag) const *(binding deprecated, see Section 15.2)* }

{static int Comm::Create_keyval(Comm::Copy_attr_function* comm_copy_attr_fn, Comm::Delete_attr_function* comm_delete_attr_fn, void* extra_state) *(binding deprecated, see Section 15.2)* }

{static int Datatype::Create_keyval(Datatype::Copy_attr_function* type_copy_attr_fn, Datatype::Delete_attr_function* type_delete_attr_fn, void* extra_state) *(binding deprecated, see Section 15.2)* }

{static int Win::Create_keyval(Win::Copy_attr_function* win_copy_attr_fn, Win::Delete_attr_function* win_delete_attr_fn, void* extra_state) *(binding deprecated, see Section 15.2)* }

{void Comm::Delete_attr(int comm_keyval) *(binding deprecated, see Section 15.2)* }

{void Datatype::Delete_attr(int type_keyval) *(binding deprecated, see Section 15.2)* }

{void Win::Delete_attr(int win_keyval) *(binding deprecated, see Section 15.2)* }

{static Group Group::Difference(const Group& group1, const Group& group2) *(binding deprecated, see Section 15.2)* }

{Cartcomm Cartcomm::Dup() const *(binding deprecated, see Section 15.2)* }

{Distgraphcomm Distgraphcomm::Dup() const *(binding deprecated, see Section 15.2)* }

{Graphcomm Graphcomm::Dup() const *(binding deprecated, see Section 15.2)* }

{Intercomm Intercomm::Dup() const *(binding deprecated, see Section 15.2)* }

{Intracomm Intracomm::Dup() const *(binding deprecated, see Section 15.2)* }

{Group Group::Excl(int n, const int ranks[]) const *(binding deprecated, see*

6. Section 3.7 on page 50.
ticket143. The Advice to users for IBSEND and IRSEND was slightly changed.

7. Section 3.7.3 on page 55.
The advice to free an active request was removed in the Advice to users for
ticket137. MPI_REQUEST_FREE.

8. Section 3.7.6 on page 67.
ticket31. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.

9. Section 5.8 on page 161.
"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and
ticket64. MPI_ALLTOALLW for intracommunicators.

10. Section 5.9.2 on page 169.
Predefined parameterized datatypes (e.g., returned by MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. MPI_REAL8) have been added to the
ticket18. list of valid datatypes in reduction operations.

11. Section 5.9.2 on page 169.
MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran integer types. MPI_C_BOOL is considered a Logical type.
MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and
ticket24. MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.

12. Section 5.9.7 on page 180.
The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been
ticket27. added.

13. Section 5.10.1 on page 182.
The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-
ticket94. dard.

14. Section 5.11.2 on page 185.
ticket19. Added in place argument to MPI_EXSCAN.

15. Section 6.4.2 on page 204, and Section 6.6 on page 224.
Implementations that did not implement MPI_COMM_CREATE on intercommunicators will need to add that functionality. As the standard described the behavior of this operation on intercommunicators, it is believed that most implementations already provide this functionality. Note also that the C++ binding for both
ticket66. MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.

16. Section 6.4.2 on page 204.
MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all
ticket33. processes in the same local group of comm must specify the same value for group.

17. Section 7.5.4 on page 268.
New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and

MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added. ticket33.

18. Section 7.5.5 on page 273.
For the scalable distributed graph topology interface, the functions MPI_DIST_NEIGHBORS_COUNT and MPI_DIST_NEIGHBORS and the constant MPI_DIST_GRAPH were added. ticket3.

19. Section 7.5.5 on page 273.
Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT. ticket101.

20. Section 8.1.1 on page 287.
The subversion number changed from 1 to 2. ticket7.

21. Section 8.3 on page 292, Section 15.2 on page 484, and Annex A.1.3 on page 539.
Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names. ticket71.

22. Section 8.7.1 on page 311.
Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD. ticket43.

23. Section 11.3.4 on page 361.
The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others. ticket6.

24. Section 12.2 on page 391.
Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments. ticket18.

25. Section 13.5.2 on page 449, and Table 13.2 on page 451.
MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the external32 representation. ticket55.

26. Section 16.3.7 on page 523.
The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples ??, ??, and ?? on pages ??-?? explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated. ticket4.

27. Annex A.1.1 on page 527.
Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 538). ticket18.