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 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter ??, a process group in MPI is a collection of \( n \) processes. Each process in the group is assigned a rank between 0 and \( n-1 \). In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the “virtual topology.”

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [5]. On the other hand, if there is no way for the user to specify the logical process arrangement as a “virtual topology,” a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [1, 2].
Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and 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.

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 than 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 $(2 \times 2)$ 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 ??.

### 7.4 Overview of the Functions

MPI supports three topology types: **Cartesian**, graph, and distributed graph. The function `MPI_CART_CREATE` is used to create Cartesian topologies, the function `MPI_GRAPH_CREATE` is used to create graph topologies, and the functions `MPI_DIST_GRAPH_CREATE_ADJACENT` and `MPI_DIST_GRAPH_CREATE` are used to create distributed graph topologies. 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.

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. For `MPI_GRAPH_CREATE` and `MPI_CART_CREATE`, all input arguments must have identical values on all processes of the group of `comm_old`. When calling `MPI_GRAPH_CREATE`, each process specifies all nodes and edges in the graph. In contrast, the functions `MPI_DIST_GRAPH_CREATE_ADJACENT` or `MPI_DIST_GRAPH_CREATE` are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of 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 ??). In analogy to function `MPI_COMM_CREATE`, no cached information propagates from `comm_old` to `comm_topol`.

`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.

`MPI` defines functions to query a communicator for topology information. The function `MPI_TOPO_TEST` is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions `MPI_GRAPHDIMS_GET` and `MPI_GRAPH_GET` return the values that were specified in the call to `MPI_GRAPH_CREATE`. Additionally, the functions `MPI_GRAPH_NEIGHBORS_COUNT` and `MPI_GRAPH_NEIGHBORS` can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions `MPI_DIST_GRAPH_NEIGHBORS_COUNT` and `MPI_DIST_GRAPH_NEIGHBORS` can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions `MPI_CARTDIM_GET` and `MPI_CART_GET` return the values that were specified in the call to `MPI_CART_CREATE`. Additionally, the functions `MPI_CART_RANK` and `MPI_CART_COORDS` translate Cartesian coordinates into a group rank, and vice-versa. The function `MPI_CART_SHIFT` provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function `MPI_CART_SUB` can be used to extract a Cartesian subspace (analogous to `MPI_COMM_SPLIT`). This function is collective over the input communicator’s group.

The two additional functions, `MPI_GRAPH_MAP` and `MPI_CART_MAP`, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter ??, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines `MPI_NEIGHBOR_ALLGATHER`, `MPI_NEIGHBOR_ALLGATHERV`, `MPI_NEIGHBOR_ALLTOALL`, `MPI_NEIGHBOR_ALLTOALLV`, and `MPI_NEIGHBOR_ALLTOALLW` communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are `MPI_INEIGHBOR_ALLGATHER`, `MPI_INEIGHBOR_ALLGATHERV`, `MPI_INEIGHBOR_ALLTOALL`, `MPI_INEIGHBOR_ALLTOALLV`, and `MPI_INEIGHBOR_ALLTOALLW`.

Unofficial Draft for Comment Only
CHAPTER 7. PROCESS TOPOLOGIES

MPI_INEIGHBOR_ALLTOALLW.

7.5 Topology Constructors

7.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN comm_old input communicator (handle)
IN ndims number of dimensions of Cartesian grid (integer)
IN dims integer array of size ndims specifying the number of processes in each dimension
IN periods logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN reorder ranking may be reordered (true) or not (false) (logical)
OUT comm_cart communicator with new Cartesian topology (handle)

int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user.
One use is to partition all the processes (the size of MPI_COMM_WORLD’s group) into an $n$-dimensional topology.

\[\text{MPI\_DIMS\_CREATE(nnodes, ndims, dims)}\]

\begin{align*}
\text{IN} & \quad \text{nnodes} \quad \text{number of nodes in a grid (integer)} \\
\text{IN} & \quad \text{ndims} \quad \text{number of Cartesian dimensions (integer)} \\
\text{INOUT} & \quad \text{dims} \quad \text{integer array of size \text{ndims} specifying the number of nodes in each dimension}
\end{align*}

\[
\text{int MPI\_Dims\_create(int nnodes, int ndims, int dims[])}
\]

\[
\text{MPI\_Dims\_create(nnodes, ndims, dims, ierr)}
\]

\[
\text{INTEGER, INTENT(IN)} :: \quad \text{nnodes, ndims} \\
\text{INTEGER, INTENT(INOUT)} :: \quad \text{dims(ndims)} \\
\text{INTEGER, OPTIONAL, INTENT(OUT)} :: \quad \text{ierror}
\]

\[
\text{MPI\_DIMS\_CREATE(NNODES, NDIMS, DIMS, IERROR)}
\]

\[
\text{INTEGER NNODES, NDIMS, DIMS(*), IERROR}
\]

The entries in the array \text{dims} are set to describe a Cartesian grid with \text{ndims} dimensions and a total of \text{nnodes} nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array \text{dims}. If \text{dims}[i] is set to a positive number, the routine will not modify the number of nodes in dimension \text{i}; only those entries where \text{dims}[i] = 0 are modified by the call.

Negative input values of \text{dims}[i] are erroneous. An error will occur if \text{nnodes} is not a multiple of \[\prod_{i, \text{dims}[i] \neq 0} \text{dims}[i].\]

For \text{dims}[i] set by the call, \text{dims}[i] will be ordered in non-increasing order. Array \text{dims} is suitable for use as input to routine \text{MPI\_CART\_CREATE}. \text{MPI\_DIMS\_CREATE} is local.

\section*{Example 7.1}

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
\text{dims} & \text{function call} & \text{dims on return} \\
\hline
(0,0) & \text{MPI\_DIMS\_CREATE(6, 2, dims)} & (3,2) \\
(0,0) & \text{MPI\_DIMS\_CREATE(7, 2, dims)} & (7,1) \\
(0,3,0) & \text{MPI\_DIMS\_CREATE(6, 3, dims)} & (2,3,1) \\
(0,3,0) & \text{MPI\_DIMS\_CREATE(7, 3, dims)} & \text{erroneous call} \\
\hline
\end{tabular}
\end{table}
7.5.3 Graph Constructor

(MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph))

- **IN** comm_old: input communicator (handle)
- **IN** nnodes: number of nodes in graph (integer)
- **IN** index: array of integers describing node degrees (see below)
- **IN** edges: array of integers describing graph edges (see below)
- **IN** reorder: ranking may be reordered (true) or not (false) (logical)
- **OUT** comm_graph: communicator with graph topology added (handle)

```c
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
                   const int edges[], int reorder, MPI_Comm *comm_graph)
```

The three parameters `nnodes`, `index`, and `edges` define the graph structure. `nnodes` is the number of nodes of the graph. The nodes are numbered from 0 to `nnodes-1`. The i-th entry of array `index` stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., `nnodes-1` are stored in consecutive locations in array `edges`. The array `edges` is a flattened representation of the edge lists. The total number of entries in `index` is `nnodes` and the total number of entries in `edges` is equal to the number of graph edges.

The definitions of the arguments `nnodes`, `index`, and `edges` are illustrated with the following simple example.

**Example 7.2**

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 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 48
```
Then, the input arguments are:

\[
\begin{align*}
n\text{nodes} & = 4 \\
\text{index} & = 2, 3, 4, 6 \\
\text{edges} & = 1, 3, 0, 3, 0, 2
\end{align*}
\]

Thus, in C, \(\text{index}[0]\) is the degree of node zero, and \(\text{index}[i] - \text{index}[i-1]\) is the degree of node \(i\), \(i=1, \ldots, n\text{nodes}-1\); the list of neighbors of node zero is stored in \(\text{edges}[j]\), for \(0 \leq j \leq \text{index}[0] - 1\) and the list of neighbors of node \(i\), \(i > 0\), is stored in \(\text{edges}[j]\), \(\text{index}[i-1] \leq j \leq \text{index}[i] - 1\).

In Fortran, \(\text{index}(1)\) is the degree of node zero, and \(\text{index}(i+1) - \text{index}(i)\) is the degree of node \(i\), \(i=1, \ldots, n\text{nodes}-1\); the list of neighbors of node zero is stored in \(\text{edges}(j)\), for \(1 \leq j \leq \text{index}(1)\) and the list of neighbors of node \(i\), \(i > 0\), is stored in \(\text{edges}(j)\), \(\text{index}(i)+1 \leq j \leq \text{index}(i+1)\).

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.

\textit{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. (\textit{End of advice to users.})

\textit{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. \(\text{ndims}\) (number of dimensions),
  2. \(\text{dims}\) (numbers of processes per coordinate direction),
  3. \(\text{periods}\) (periodicity information),
  4. \(\text{own\_position}\) (own position in grid, could also be computed from rank and \(\text{dims}\))
- For a graph topology:
  1. \(\text{index}\),
  2. \(\text{edges}\),

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 \(\text{index}\) simplifies access to the topology information. (\textit{End of advice to implementors.})
7.5.4 Distributed Graph Constructor

(MPI_GRAPH_CREATE requires 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 each 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 such that any process can indicate that communication will occur between any pair of processes in the graph.

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, const int sources[], const int sourceweights[], int outdegree,
7.5. **TOPOLOGY CONSTRUCTORS**

```c
const int destinations[], const 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)
```

**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 its incoming and outgoing edges 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 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 in **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. It is erroneous to supply **MPI_UNWEIGHTED** for some but not all processes of **comm_old**. If the graph is weighted but indegree or outdegree is zero, then **MPI_WEIGHTS_EMPTY** or any arbitrary array may be passed to sourceweights.
or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section ??.

Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.)

Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex ??, (End of rationale.)

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, const int sources[],
                           const int degrees[], const int destinations[],
                           const int weights[], MPI_Info info, int reorder,
                           MPI_Comm *comm_dist_graph)
7.5. TOPOLOGY CONSTRUCTORS

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

TYPE(MPI_Comm), INTENT(IN) :: comm_old
INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
INTEGER, INTENT(IN) :: weights(*)
TYPE(MPI_Info), INTENT(IN) :: info
LOGICAL, INTENT(IN) :: reorder
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_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. It is erroneous to supply MPI_UNWEIGHTED for some but not all processes of comm_old. If the graph is weighted but \( n = 0 \), then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section ??.

**Advice to users.** In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.)

**Advice to implementors.** It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

**Rationale.** To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex ??. (End of rationale.)

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 valid 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:

<table>
<thead>
<tr>
<th>process</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

Unofficial Draft for Comment Only
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:

<table>
<thead>
<tr>
<th>process</th>
<th>n</th>
<th>sources</th>
<th>degrees</th>
<th>destinations</th>
<th>weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1,3</td>
<td>1,1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0,2</td>
<td>1,1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>process</th>
<th>n</th>
<th>sources</th>
<th>degrees</th>
<th>destinations</th>
<th>weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>0,1,2,3</td>
<td>2,1,1,2</td>
<td>1,3,0,3,0,2</td>
<td>1,1,1,1,1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>process</th>
<th>indegree</th>
<th>sources</th>
<th>sourceweights</th>
<th>outdegree</th>
<th>destinations</th>
<th>destweights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1,3</td>
<td>1,1</td>
<td>2</td>
<td>1,3</td>
<td>1,1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0,2</td>
<td>1,1</td>
<td>2</td>
<td>0,2</td>
<td>1,1</td>
</tr>
</tbody>
</table>

**Example 7.4** A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled 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:

```c
/*
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 comm_dist_graph;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* get x and y dimension */
y=rank/P; x=rank%P;
```
CHAPTER 7. PROCESS TOPOLOGIES

/* 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[2] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;

/* get my communication partners along diagonals */
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 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)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

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_CART Cartesian topology
MPI_DIST_GRAPH distributed graph topology
MPI_UNDEFINED no topology
7.5. **TOPOLOGY CONSTRUCTORS**

MPI_GRAPHDIM_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)

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

MPI_GRAPHDIM_GET(comm, nnodes, nedges, ierror)

```c
MPI_Graphdims_get(comm, nnodes, nedges, ierror)
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: nnodes, nedges
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_GRAPHDIM_GET(comm, nnodes, nedges, ierror)

```c
MPI_GRAPHDIM_GET(COMM, NNODES, NEDGES, IERROR)
```

**IN**  
comm  
communicator with graph structure (handle)

**IN**  
maxindex  
length of vector index in the calling program (integer)

**IN**  
maxedges  
length of vector edges in the calling program (integer)

**OUT**  
index  
array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)

**OUT**  
edges  
array of integers containing the graph structure

```c
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[])
```

MPI_GRAPH_get(comm, maxindex, maxedges, index, edges, ierror)

```c
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: maxindex, maxedges
INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges, ierror)

```c
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
```

**IN**  
comm  
communicator with graph structure (handle)

**IN**  
maxindex  
length of vector index in the calling program (integer)

**IN**  
maxedges  
length of vector edges in the calling program (integer)

**OUT**  
index  
array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)

**OUT**  
edges  
array of integers containing the graph structure

```c
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
                  int edges[])
The functions `MPI_CARTDIM_GET` and `MPI_CART_GET` return the Cartesian topology information that was associated with a communicator by `MPI_CART_CREATE`. If `comm` is associated with a zero-dimensional Cartesian topology, `MPI_CARTDIM_GET` returns `ndims=0` and `MPI_CART_GET` will keep all output arguments unchanged.

```c
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: ndims
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

```c
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                 int coords[])
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: maxdims
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
    LOGICAL, INTENT(OUT) :: periods(maxdims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```
7.5. **TOPOLOGY CONSTRUCTORS**

`MPI_CART_RANK(comm, coords, rank)`

**IN**  
`comm`  
communicator with Cartesian structure (handle)

**IN**  
`coords`  
integer array (of size `ndims`) specifying the Cartesian coordinates of a process

**OUT**  
`rank`  
rank of specified process (integer)

```c
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

**MPI_Cart_rank(comm, coords, rank, ierror)**

```
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: coords(*)
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**MPI_CART_RANK(COMM, COORDS, RANK, IERROR)**

```
INTEGER COMM, COORDS(*), RANK, IERROR
```

For a process group with Cartesian structure, the function `MPI_CART_RANK` translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension `i` with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i) < 0` or `coords(i) ≥ dims(i)`, it is shifted back to the interval `0 ≤ coords(i) < dims(i)` automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If `comm` is associated with a zero-dimensional Cartesian topology, `coords` is not significant and 0 is returned in `rank`.

`MPI_CART_COORDS(comm, rank, maxdims, coords)`

**IN**  
`comm`  
communicator with Cartesian structure (handle)

**IN**  
`rank`  
rank of a process within group of `comm` (integer)

**IN**  
`maxdims`  
length of vector `coords` in the calling program (integer)

**OUT**  
`coords`  
integer array (of size `ndims`) containing the Cartesian coordinates of specified process (array of integers)

```c
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

**MPI_Cart_coords(comm, rank, maxdims, coords, ierror)**

```
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: rank, maxdims
INTEGER, INTENT(OUT) :: coords(maxdims)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)**

```
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

The inverse mapping, rank-to-coordinates translation is provided by `MPI_CART_COORDS`.

Unofficial Draft for Comment Only
If \texttt{comm} is associated with a zero-dimensional Cartesian topology, \texttt{coords} will be unchanged.

\texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} (\texttt{comm}, \texttt{rank}, \texttt{nneighbors})

\begin{verbatim}
IN    comm        communicator with graph topology (handle)
IN    rank       rank of process in group of \texttt{comm} (integer)
OUT   nneighbors number of neighbors of specified process (integer)
\end{verbatim}

\begin{verbatim}
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
\end{verbatim}

\texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} provides adjacency information for a graph topology. The returned count and array of neighbors for the queried rank will both include all neighbors and reflect the same edge ordering as was specified by the original call to \texttt{MPI\_GRAPH\_CREATE}. Specifically, \texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} and \texttt{MPI\_GRAPH\_NEIGHBORS} will return values based on the original index and edges array passed to \texttt{MPI\_GRAPH\_CREATE} (for the purpose of this example, we assume that \texttt{index[-1]} is zero):

\texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT\_COUNT} and \texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} provide adjacency information for a graph topology. The returned count and array of neighbors for the queried rank will both include all neighbors and reflect the same edge ordering as was specified by the original call to \texttt{MPI\_GRAPH\_CREATE}. Specifically, \texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} and \texttt{MPI\_GRAPH\_NEIGHBORS} will return values based on the original index and edges array passed to \texttt{MPI\_GRAPH\_CREATE} (for the purpose of this example, we assume that \texttt{index[-1]} is zero):
7.5. **TOPOLOGY CONSTRUCTORS**

- The number of neighbors \( n_{\text{neighbors}} \) returned from \texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} will be \( \text{index}[\text{rank}] - \text{index}[\text{rank-1}] \).

- The neighbors array returned from \texttt{MPI\_GRAPH\_NEIGHBORS} will be \( \text{edges}[\text{index}[\text{rank-1}]] \) through \( \text{edges}[\text{index}[\text{rank}]-1] \).

**Example 7.5**

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

<table>
<thead>
<tr>
<th>process</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 1, 3</td>
</tr>
<tr>
<td>1</td>
<td>0, 0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0, 2, 2</td>
</tr>
</tbody>
</table>

Thus, the input arguments to \texttt{MPI\_GRAPH\_CREATE} are:

\[
\begin{align*}
\text{nndes} & = 4 \\
\text{index} & = 3, 5, 6, 9 \\
\text{edges} & = 1, 1, 3, 0, 0, 3, 0, 2, 2
\end{align*}
\]

Therefore, calling \texttt{MPI\_GRAPH\_NEIGHBORS\_COUNT} and \texttt{MPI\_GRAPH\_NEIGHBORS} for each of the 4 processes will return:

<table>
<thead>
<tr>
<th>Input rank</th>
<th>Count</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1, 1, 3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0, 0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0, 2, 2</td>
</tr>
</tbody>
</table>

**Example 7.6**

Suppose that \texttt{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\( (a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1 \)
- unshuffle\( (a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1} \)

The graph adjacency list is illustrated below for \( n = 3 \).

<table>
<thead>
<tr>
<th>node</th>
<th>exchange (neighbors(1))</th>
<th>shuffle (neighbors(2))</th>
<th>unshuffle (neighbors(3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 (001)</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2 (010)</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3 (011)</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4 (100)</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5 (101)</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6 (110)</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7 (111)</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Unofficial Draft for Comment Only
Suppose that the communicator \texttt{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.

\begin{verbatim}
! assume: each process has stored a real number A.
! extract neighborhood information
   CALL MPI_COMM_RANK(comm, myrank, ierr)
   CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
! perform exchange permutation
   CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, 
                          neighbors(1), 0, comm, status, ierr)
! perform shuffle permutation
   CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, 
                          neighbors(3), 0, comm, status, ierr)
! perform unshuffle permutation
   CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, 
                          neighbors(2), 0, comm, status, ierr)
\end{verbatim}

\texttt{MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT} and \texttt{MPI\_DIST\_GRAPH\_NEIGHBORS} provide adjacency information for a distributed graph topology.

\begin{verbatim}
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)
\end{verbatim}
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)

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. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in comm_old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then 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_NEIGHBORS_COUNT, then only the first part of the full list is returned.

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 may 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.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN comm communicator with Cartesian structure (handle)
IN direction coordinate dimension of shift (integer)
IN disp displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT rank_source rank of source process (integer)
OUT rank_dest rank of destination process (integer)

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)

The direction argument indicates the coordinate dimension to be traversed by the shift. The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions. Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

Unofficial Draft for Comment Only
It is erroneous to call \texttt{MPI\_CART\_SHIFT} with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call \texttt{MPI\_CART\_SHIFT} with a \texttt{comm} that is associated with a zero-dimensional Cartesian topology.

\textbf{Example 7.7}

The communicator, \texttt{comm}, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of \texttt{REALs} is stored one element per process, in variable \texttt{A}. One wishes to skew this array, by shifting column \texttt{i} (vertically, i.e., along the column) by \texttt{i} steps.

\begin{verbatim}
! find process rank
CALL MPI_COMM_RANK(comm, rank, ierr)
! find Cartesian coordinates
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
! compute shift source and destination
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
! skew array
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
   status, ierr)
\end{verbatim}

\textit{Advice to users.} In Fortran, the dimension indicated by $\text{DIRECTION} = i$ has $\text{DIMS}(i+1)$ nodes, where \texttt{DIMS} is the array that was used to create the grid. In C, the dimension indicated by $\text{direction} = i$ is the dimension specified by \texttt{dims}[i]. \textit{(End of advice to users.)}

\section*{7.5.7 Partitioning of Cartesian Structures}

\texttt{MPI\_CART\_SUB}(\texttt{comm}, \texttt{remain_dims}, \texttt{newcomm})

\begin{verbatim}
IN       \texttt{comm}          communicator with Cartesian structure (handle)
IN       \texttt{remain_dims}  the i-th entry of \texttt{remain_dims} specifies whether the
                           i-th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT      \texttt{newcomm}      communicator containing the subgrid that includes
                           the calling process (handle)
\end{verbatim}

\begin{verbatim}
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
\end{verbatim}

Unofficial Draft for Comment Only
If a Cartesian topology has been created with `MPI_CART_CREATE`, the function `MPI_CART_SUB` can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in `remain_dims` are false or `comm` is already associated with a zero-dimensional Cartesian topology then `newcomm` is associated with a zero-dimensional Cartesian topology. (This function is closely related to `MPI_COMM_SPLIT`.)

**Example 7.8**

Assume that `MPI_CART_CREATE(…, comm)` has defined a `(2 × 3 × 4)` grid. Let `remain_dims = (true, false, true)`. Then a call to

```
MPI_CART_SUB(comm, remain_dims, comm_new);
```

will create three communicators each with eight processes in a `2 × 4` Cartesian topology. If `remain_dims = (false, false, true)` then the call to `MPI_CART_SUB(comm, remain_dims, comm_new)` will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

### 7.5.8 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI. The two calls are both local.

**MPI_CART_MAP(comm, ndims, dims, periods, newrank)**

```
IN comm input communicator (handle)
IN ndims number of dimensions of Cartesian structure (integer)
IN dims integer array of size `ndims` specifying the number of processes in each coordinate direction
IN periods logical array of size `ndims` specifying the periodicity specification in each coordinate direction
OUT newrank reordered rank of the calling process; `MPI_UNDEFINED` if calling process does not belong to grid (integer)
```

```c
int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
                 const int periods[], int *newrank)
```

```c
MPI_Cart_map(comm, ndims, dims, periods, newrank, ierr)
```

```c
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
    LOGICAL, INTENT(IN) :: periods(ndims)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierr
```

```c
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
```

Unofficial Draft for Comment Only
INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
LOGICAL PERIODS(*)

MPI_CART_MAP computes an “optimal” placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart), with reorder = true can be implemented by calling

MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling

MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank ≠ MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims is zero then a zero-dimensional Cartesian topology is created.

The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

The corresponding function for graph structures is as follows.

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)
IN comm input communicator (handle)
IN nnodes number of graph nodes (integer)
IN index integer array specifying the graph structure, see MPI_GRAPH_CREATE
IN edges integer array specifying the graph structure
OUT newrank reordered rank of the calling process;
MPI_UNDEFINED if the calling process does not belong to graph (integer)

int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],
    const int edges[], int *newrank)
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with reorder = true can be implemented by calling

Unofficial Draft for Comment Only
MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank), then calling

MPI\_COMM\_SPLIT(comm, color, key, comm\_graph), with color = 0 if newrank ≠

MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

### 7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All of these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 5 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

If the graph was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT with sources and destinations containing 0, ..., n-1, where n is the number of processes in the group of comm\_old (i.e., the graph is fully connected and also includes an edge from each node to itself), then the sparse neighborhood communication routine performs the same data exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank\_source and rank\_dest in MPI\_CART\_SHIFT with input disp=1.

*Rationale.* Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [4]. This functionality can significantly simplify the implementation of neighbor exchanges [3]. (*End of rationale.*)

For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI\_GRAPH\_CREATE, the use of neighborhood collective communication is restricted to adjacency matrices, where the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix). In this case, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI\_GRAPH\_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI\_CART\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are 2*ndims with ndims defined in MPI\_CART\_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., periods[...]==false), then this neighbor is defined to be MPI\_PROC\_NULL.

If a neighbor in any of the functions is MPI\_PROC\_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI\_PROC\_NULL in
7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

this direction. That is, the buffer is still part of the sequence of neighbors but it is neither
communicated nor updated.

7.6.1 Neighborhood Gather

In this function, each process $i$ gathers data items from each process $j$ if an edge $(j, i)$ exists
in the topology graph, and each process $i$ sends the same data items to all processes $j$ where
an edge $(i, j)$ exists. The send buffer is sent to each neighboring process and the $l$-th block
in the receive buffer is received from the $l$-th neighbor.

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 
comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements sent to each neighbor (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf starting address of receive buffer (choice)
IN recvcount number of elements received from each neighbor (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN comm communicator with topology structure (handle)

int MPI_Neighbor_allgather(const void* sendbuf, int sendcount, 
MPI_Datatype sendtype, void* recvbuf, int recvcount, 
MPI_Datatype recvtype, MPI_Comm comm)

MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 
recvtype, comm, ierror)

This function supports Cartesian communicators, graph communicators, and distributed
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
and receives from each of its incoming neighbors:

MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                        outdegree,dsts,MPI_UNWEIGHTED);

int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
            srcs[l],...);
MPI_Waitall(...);

Figure 7.1 shows the neighborhood gather communication of one process with outgoing neighbors \( d_0 \ldots d_3 \) and incoming neighbors \( s_0 \ldots s_5 \). The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.

Figure 7.1: Neighborhood gather communication example.

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

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

Rationale. For optimization reasons, the same type signature is required independently of whether the topology graph is connected or not. (End of rationale.)
The "in place" option is not meaningful for this operation.

The vector variant of MPI\_NEIGHBOR\_ALLGATHER allows one to gather different numbers of elements from each neighbor.

**MPI\_NEIGHBOR\_ALLGATHERV**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>starting address of send buffer (choice)</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements sent to each neighbor (non-negative integer)</td>
</tr>
<tr>
<td>sendtype</td>
<td>data type of send buffer elements (handle)</td>
</tr>
<tr>
<td>recvbuf</td>
<td>starting address of receive buffer (choice)</td>
</tr>
<tr>
<td>recvcounts</td>
<td>non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor</td>
</tr>
<tr>
<td>displs</td>
<td>integer array (of length indegree). Entry (i) specifies the displacement (relative to (\text{recvbuf})) at which to place the incoming data from neighbor (i)</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements (handle)</td>
</tr>
<tr>
<td>comm</td>
<td>communicator with topology structure (handle)</td>
</tr>
</tbody>
</table>

```c
int MPI\_Neighbor\_allgatherv(const void* \text{sendbuf}, int \text{sendcount},
                               MPI\_Datatype \text{sendtype}, void* \text{recvbuf},
                               const int \text{recvcounts}[],
                               const int \text{displs}[],
                               MPI\_Datatype \text{recvtype}, MPI\_Comm \text{comm})
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If \text{comm} is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```c
MPI\_Dist\_graph\_neighbors\_count(\text{comm}, \text{indegree}, \text{outdegree}, \text{weighted});

int *\text{srcs}=(\text{int}*)\text{malloc}(\text{indegree}^{*}\text{sizeof(\text{int}}));
```
int *dsts=(int*)malloc(outdegree*sizeof(int));

MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
    outdegree,dsts,MPI_UNWEIGHTED);

int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
    MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);

for(l=0; l<indegree; ++l)
    MPI_Irecv(recvbuf+displs[l]*extent(recvtype),recvcounts[l],recvtype,
        srcs[l],...);

MPI_Waitall(...);

The type signature associated with sendcount, sendtype, at process j must be equal
to the type signature associated with recvcounts[l], recvtype at any other process with
srcs[l]==j. This implies that the amount of data sent must be equal to the amount of
data received, pairwise between every pair of communicating processes. Distinct type maps
between sender and receiver are still allowed. The data received from the l-th neighbor is
placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).

The “in place” option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identi-
tical values on all processes.

7.6.2 Neighbor Alltoall

In this function, each process i receives data items from each process j if an edge (j, i)
exists in the topology graph or Cartesian topology. Similarly, each process i sends data
items to all processes j where an edge (i, j) exists. This call is more general than
MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor.
The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in
the receive buffer is received from the l-th neighbor.
7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

**MPI_NEIGHBOR_ALLTOALL** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendbuf</td>
</tr>
<tr>
<td>IN</td>
<td>sendcount</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
</tr>
<tr>
<td>IN</td>
<td>recvcount</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
</tr>
</tbody>
</table>

```c
int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
                          MPI_Datatype sendtype, void* recvbuf, int recvcount,
                          MPI_Datatype recvtype, MPI_Comm comm)
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If `comm` is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```c
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
                          outdegree,dsts,MPI_UNWEIGHTED);
int k,l;
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k) {
    MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
              dsts[k],...);
```
for(l=0; l<indegree; ++l)
    MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
              srcs[l],...);
MPI_Waitall(...);

The type signature associated with sendcount, sendtype, at a process must be equal to
the type signature associated with recvcount, recvtype at any other process. This implies
that the amount of data sent must be equal to the amount of data received, pairwise between
every pair of communicating processes. Distinct type maps between sender and receiver are
still allowed.

The “in place” option is not meaningful for this operation.

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

The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
numbers of elements to and from each neighbor.

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

IN   sendbuf                  starting address of send buffer (choice)
IN   sendcounts               non-negative integer array (of length outdegree) specifying
                               the number of elements to send to each neighbor
IN   sdispls                  integer array (of length outdegree). Entry j specifies
                               the displacement (relative to sendbuf) from which to
                               send the outgoing data to neighbor j
IN   sendtype                 data type of send buffer elements (handle)
OUT  recvbuf                  starting address of receive buffer (choice)
IN   recvcounts               non-negative integer array (of length indegree) specifying
                               the number of elements that are received from
each neighbor
IN   rdispls                  integer array (of length indegree). Entry i specifies the
                               displacement (relative to recvbuf) at which to place the
                               incoming data from neighbor i
IN   recvtype                 data type of receive buffer elements (handle)
IN   comm                     communicator with topology structure (handle)

int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
                           const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
                           const int recvcounts[], const int rdispls[],
                           MPI_Datatype recvtype, MPI_Comm comm)

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

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

Unofficial Draft for Comment Only
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS,
RECVTYPE, COMM, IERROR)

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
outdegree,dsts,MPI_UNWEIGHTED);

int k,l;
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
            dsts[k],...);
for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+rdispls[l]*extent(recvtype),recvcounts[l],recvtype,
            srcs[l],...);

MPI_Waitall(...);

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==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 communicating processes. Distinct type maps between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the l-th incoming neighbor is placed into recvbuf beginning at offset rdispls[l] elements (in terms of the recvtype).

The “in place” option is not meaningful for this operation.

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

MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.
MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcounts non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN sdispls integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
IN sendtypes array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
OUT recvbuf starting address of receive buffer (choice)
IN recvcounts non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
IN rdispls integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
IN recvtypes array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
IN comm communicator with topology structure (handle)

int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)

MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, ierror)

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

This function supports Cartesian communicators, graph communicators, and distributed
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
and receives from each of its incoming neighbors:

```c
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
outdegree,dsts,MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
    MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);
for(l=0; l<indegree; ++l)
    MPI_Irecv(recvbuf+rdispls[l],recvcounts[l], recvtypes[l],srcs[l],...);
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at pro-
cess i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with
srcs[l]==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 communicating processes. Distinct
type maps between sender and receiver are still allowed.

The “in place” option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have iden-
tical values on all processes.

7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchroniza-
tion and overlapping of computation and communication. The semantics are similar to
nonblocking collective operations as described in Section 5.12.
7.7.1 Nonblocking Neighborhood Gather

**MPI_INEIGHBOR_ALLGATHER**

```
IN sendbuf          starting address of send buffer (choice)
IN sendcount        number of elements sent to each neighbor (non-negative integer)
IN sendtype         data type of send buffer elements (handle)
OUT recvbuf         starting address of receive buffer (choice)
IN recvcount        number of elements received from each neighbor (non-negative integer)
IN recvtype         data type of receive buffer elements (handle)
IN comm             communicator with topology structure (handle)
OUT request         communication request (handle)
```

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

This call starts a nonblocking variant of **MPI_NEIGHBOR_ALLGATHER**.
7.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION

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

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements sent to each neighbor (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf starting address of receive buffer (choice)
IN recvcounts non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
IN displs integer array (of length indegree). Entry $i$ specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor $i$
IN recvtype data type of receive buffer elements (handle)
IN comm communicator with topology structure (handle)
OUT request communication request (handle)

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

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

This call starts a nonblocking variant of MPI_INEIGHBOR_ALLGATHERV.
7.7.2 Nonblocking Neighborhood Alltoall

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

IN sendbuf    starting address of send buffer (choice)
IN sendcount  number of elements sent to each neighbor (non-negative integer)
IN sendtype   data type of send buffer elements (handle)
OUT recvbuf   starting address of receive buffer (choice)
IN recvcount  number of elements received from each neighbor (non-negative integer)
IN recvtype   data type of receive buffer elements (handle)
IN comm       communicator with topology structure (handle)
OUT request   communication request (handle)

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

MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 
                        recvtype, comm, request, ierror)

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.
7.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION

MPI_INEIGHBOR_ALLTOALLV(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 outdegree) specifying the number of elements to send to each neighbor

IN
dsdispls
integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j

IN
sendtype
data type of send buffer elements (handle)

OUT
recvbuf
starting address of receive buffer (choice)

IN
recvcounts
non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor

INdispls
integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i

IN
recvtype
data type of receive buffer elements (handle)

IN
comm
communicator with topology structure (handle)

OUT
request
communication request (handle)

int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

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

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

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

INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)

TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype

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

TYPE(MPI_Request), INTENT(OUT) :: request

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)

This call starts a nonblocking variant of MPI_INEIGHBOR_ALLTOALLV.
MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

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

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

MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request, ierr)
This call starts a nonblocking variant of MPI\_NEIGHBOR\_ALLTOALLW.

7.8 Persistent Neighborhood Communication on Process Topologies

Persistent variants of the neighborhood collective operations can offer significant performance benefits for programs with repetitive communication patterns. The semantics are similar to persistent collective operations as described in Section 5.13.

7.8.1 Persistent Neighborhood Gather

\begin{verbatim}
MPI\_NEIGHBOR\_ALLGATHER\_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recv-
type, comm, request)
\end{verbatim}

\begin{verbatim}
IN    sendbuf     starting address of send buffer (choice)
IN    sendcount   number of elements sent to each neighbor (non-negative integer)
IN    sendtype    data type of send buffer elements (handle)
OUT   recvbuf     starting address of receive buffer (choice)
IN    recvcount   number of elements received from each neighbor (non-negative integer)
IN    recvtype    data type of receive buffer elements (handle)
IN    comm        communicator with topology structure (handle)
OUT   request     communication request (handle)
\end{verbatim}

\begin{verbatim}
int MPI\_Neighbor\_allgather\_init(const void* sendbuf, int sendcount,
   MPI\_Datatype sendtype, void* recvbuf, int recvcount,
   MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request *request)
\end{verbatim}
CHAPTER 7. PROCESS TOPOLOGIES

MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
    RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR)

\textbf{<type>} SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

Creates a persistent collective communication request for the neighborhood allgather operation.

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

\textbf{IN} sendbuf \quad \text{starting address of send buffer (choice)}

\textbf{IN} sendcount \quad \text{number of elements sent to each neighbor (non-negative integer)}

\textbf{IN} sendtype \quad \text{data type of send buffer elements (handle)}

\textbf{OUT} recvbuf \quad \text{starting address of receive buffer (choice)}

\textbf{IN} recvcounts \quad \text{non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor}

\textbf{IN} displs \quad \text{integer array (of length indegree). Entry } i \text{ specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor } i

\textbf{IN} recvtype \quad \text{data type of receive buffer elements (handle)}

\textbf{IN} comm \quad \text{communicator with topology structure (handle)}

\textbf{OUT} request \quad \text{communication request (handle)}

\begin{verbatim}
int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
    const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
    MPI_Request *request)

MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
    recvcounts, displs, recvtype, comm, request, ierror)
\end{verbatim}

Unofficial Draft for Comment Only
7.8. PERSISTENT NEIGHBORHOOD COMMUNICATION

Creates a persistent collective communication request for the neighborhood allgatherv operation.

7.8.2 Persistent Neighborhood Alltoall

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

- **IN** sendbuf: starting address of send buffer (choice)
- **IN** sendcount: number of elements sent to each neighbor (non-negative integer)
- **IN** sendtype: data type of send buffer elements (handle)
- **OUT** recvbuf: starting address of receive buffer (choice)
- **IN** recvcount: number of elements received from each neighbor (non-negative integer)
- **IN** recvtype: data type of receive buffer elements (handle)
- **IN** comm: communicator with topology structure (handle)
- **OUT** request: communication request (handle)

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

MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror)

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

MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR)

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

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

Creates a persistent collective communication request for the neighborhood alltoall operation.
MPI_NEIGHBOR_ALLTOALLV_INIT(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 outdegree) specifying the number of elements to send to each neighbor
IN  sdispls  integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
IN  sendtype  data type of send buffer elements (handle)
OUT recvbuf  starting address of receive buffer (choice)
IN  recvcounts  non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
IN  rdispls  integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN  recvtype  data type of receive buffer elements (handle)
IN  comm  communicator with topology structure (handle)
OUT request  communication request (handle)

int MPI_Neighbor_alltoallv_init(const void* sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)

Creates a persistent collective communication request for the neighborhood alltoallv operation.

Unofficial Draft for Comment Only
MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

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

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

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
   <type>  SENDBUF(*), RECVBUF(*)
   INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
   INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
    REQUEST, IERROR

Creates a persistent collective communication request for the neighborhood alltoallw operation.

7.9 An Application Example

Example 7.9 The example in Figures 7.2-7.4 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution grid function at the points $u(1:100,1:100)$ owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in $u(1,1:100)$ must be sent into the halo cells $u(101,1:100)$ of the left-hand neighbor with coordinates $(\text{own\_coord}(1)-1,\text{own\_coord}(2))$. 
INTEGER ndims, num_neigh
LOGICAL reorder
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
LOGICAL periods(ndims)
REAL u(0:101,0:101), f(0:101,0:101)
DATA dims / ndims * 0 /
comm = MPI_COMM_WORLD
! Set process grid size and periodicity
CALL MPI_DIMS_CREATE(comm, ndims, dims, ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
! Create a grid structure in WORLD group and inquire about own position
CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
comm_cart, ierr)
CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords, ierr)
i = own_coords(1)
j = own_coords(2)
! Look up the ranks for the neighbors. Own process coordinates are (i,j).
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3),neigh_rank(4), ierr)
! Initialize the grid functions and start the iteration
CALL init (u, f)
DO it=1,100
   CALL relax (u, f)
   ! Exchange data with neighbor processes
   CALL exchange (u, comm_cart, neigh_rank, num_neigh)
END DO
CALL output (u)

Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u( 1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100, 1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
         comm_cart, ierr)
! instead of
! DO i=1,num_neigh
!     CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1],&
!         ierr)
!     CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),&
!         ierr)
! END DO
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)

u( 0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100, 0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END

Figure 7.3: Communication routine with local data copying and sparse neighborhood all-to-all.
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
IMPLICIT NONE
USE MPI
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
INTEGER type_vec, ierr
! The following initialization need to be done only once
! before the first call of exchange.
CALL MPI_TYPE_GET_EXTENT (MPI_REAL, lb, sizeofreal, ierr)
CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
CALL MPI_TYPE_COMMIT (type_vec, ierr)
sndtypes(1:2) = type_vec
sndcounts(1:2) = 1
sndtypes(3:4) = MPI_REAL
sndcounts(3:4) = 100
rcvtypes = sndtypes
rcvcounts = sndcounts
sdispls(1) = ( 1 + 1*102) * sizeofreal ! first element of u( 1 , 1:100)
sdispls(2) = (100 + 1*102) * sizeofreal ! first element of u(100 , 1:100)
sdispls(3) = ( 1 + 1*102) * sizeofreal ! first element of u( 1:100, 1 )
sdispls(4) = ( 1 + 100*102) * sizeofreal ! first element of u( 1:100,100 )
rdispls(1) = ( 0 + 1*102) * sizeofreal ! first element of u( 0 , 1:100)
rdispls(2) = (101 + 1*102) * sizeofreal ! first element of u(101 , 1:100)
rdispls(3) = ( 1 + 0*102) * sizeofreal ! first element of u( 1:100, 0 )
rdispls(4) = ( 1 + 101*102) * sizeofreal ! first element of u( 1:100,101 )
! the following communication has to be done in each call of exchange
CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes, &
   u, rcvcounts, rdispls, rcvtypes, &
   comm_cart, ierr)
! The following finalizing need to be done only once
! after the last call of exchange.
CALL MPI_TYPE_FREE (type_vec, ierr)
END

Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local data copying.
Bibliography


Index

CONST:DIMS, 23
CONST:DIMS(i+1), 23
CONST:dims[i], 23
CONST:DIRECTION = i, 23
CONST:direction = i, 23
CONST:false, 4, 6, 8, 10, 16, 20
CONST:MPI_CART, 14
CONST:MPI_COMM_NULL, 4, 6
CONST:MPI_COMM_WORLD, 5
CONST:MPI_DIST_GRAPH, 14
CONST:MPI_GRAPH, 14
CONST:MPI_INFO_NULL, 12
CONST:MPI_PROC_NULL, 22, 26
CONST:MPI_UNDEFINED, 14, 24, 25
CONST:MPI_UNWEIGHTED, 9, 10, 12, 13, 20, 21
CONST:MPI_WEIGHTS_EMPTY, 9, 10, 12
CONST:NULL, 10, 12
CONST:true, 4, 6, 8, 10, 16, 20

EXAMPLES: Cartesian virtual topologies, 46
EXAMPLES: MPI_CARTCOORDS, 3, 17
EXAMPLES: MPI_CARTCOORDS(comm, rank, maxdims, coords), 17
EXAMPLES: MPI_CARTCREATE, 2–6, 16, 24, 26
EXAMPLES: MPI_CARTCREATE(comm, ndims, dims, periods, reorder, comm_cart), 25
EXAMPLES: MPI_CARTCREATE(comm_old, ndims, dims, periods, reorder, comm_cart), 4
EXAMPLES: MPI_CARTGET, 3, 16
EXAMPLES: MPI_CARTGET(comm, maxdims, dims, periods, coords), 16
EXAMPLES: MPI_CARTMAP, 3, 25
EXAMPLES: MPI_CARTMAP(comm, ndims, dims, periods, newrank), 24, 25
EXAMPLES: MPI_CARTRANK, 3, 17
EXAMPLES: MPI_CARTRANK(comm, coords, rank), 17
EXAMPLES: MPI_CARTSHIFT, 3, 22, 23, 26
EXAMPLES: MPI_CARTSHIFT(comm, direction, disp, rank_source, rank_dest), 22
EXAMPLES: MPI_CARTSUB, 3, 24
EXAMPLES: MPI_CARTSUB(comm, remain_dims, comm_new), 24, 25
EXAMPLES: MPI_CARTSUB(comm, remain_dims, newcomm), 23
EXAMPLES: MPI_CARTDIMGET, 3, 16
EXAMPLES: MPI_CARTDIMGET(comm, ndims), 16
EXAMPLES: MPI_GRAPHCREATE, 6, 19
EXAMPLES: MPI_GRAPHNEIGHBORS, 19
EXAMPLES: MPI_GRAPHNEIGHBORS_COUNT, 25
EXAMPLES: MPI_SENDRECV_REPLACE, 23
EXAMPLES: Neighborhood collective communication, 46
EXAMPLES: Topologies, 46
EXAMPLES: Virtual topologies, 46
MPI_DIMS_CREATE(6, 2, dims), 5
MPI_DIMS_CREATE(6, 3, dims), 5
MPI_DIMS_CREATE(7, 2, dims), 5
MPI_DIMS_CREATE(7, 3, dims), 5
MPI_DIMS_CREATE(nnodes, ndims, dims), 5
MPI_DIST_GRAPH_CREATE, 2, 3, 8, 11, 13, 21, 22, 26
MPI_DIST_GRAPH_CREATE(comm_old, n, MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 36
MPI_INEIGHBOR_ALLGATHER, 3
MPI_DIST_GRAPH_NEIGHBORS, 3, 20, 21
MPI_DIST_GRAPH_NEIGHBORS(comm, maxdegree, destinations, destweights, info, reorder, comm_dist_graph), 8
MPI_DIST_GRAPH_NEIGHBORS_COUNT, 3, 20–22
MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, request), 40
MPI_GRAPH_CREATE, 2, 3, 6, 8, 12, 15, 18, 19, 25, 26
MPI_GRAPH_CREATE(comm, mnodes, index, edges, reorder, comm_graph), 25
MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph), 6
MPI_GRAPH_GET, 3, 15
MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges), 15
MPI_GRAPH_MAP, 3
MPI_GRAPH_MAP(comm, mnodes, index, edges, newrank), 25, 26
MPI_GRAPH_NEIGHBORS, 3, 18, 19, 26
MPI_GRAPH_NEIGHBORS(comm, rank, mneighbors, bors, neighbors), 18
MPI_GRAPH_NEIGHBORS_COUNT, 3, 18, 19
MPI_GRAPH_NEIGHBORS_COUNT(comm, MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 31
MPI_INEIGHBOR_ALLTOALL, 3
MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 38
MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun, recv types, comm, rdispls, recvtype, comm, request), 39
MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sendtypes, recvtype, comm, request), 31
MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 27
MPI_INEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 41
MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, comm, request), 37
MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 36
MPI_NEIGHBOR_ALLGATHER, 3, 29, 30
MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm), 27
MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request), 41
MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcounts, displs, recvtype, comm, request), 37
MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm), 31
INDEX

MPI_NEIGHBOR_ALLTOALLV, 3, 39
MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm), 32
MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request), 44
MPI_NEIGHBOR_ALLTOALLW, 3, 33, 41
MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm), 34
MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request), 45
MPI_SENDRECV, 22
MPI_TOPO_TEST, 3, 14
MPI_TOPO_TEST(comm, status), 14

TERM:Cartesian – topology, 2, 4
TERM:collective communication neighborhood, 26
TERM:distributed graph – topology, 2, 8
TERM:graph – topology, 2, 6
TERM:neighborhood collective communication, 26
nonblocking, 36
TERM:topologies, 1
TERM:topology
  Cartesian, 2, 4
distributed graph, 2, 8
graph, 2, 6
virtual, 2
TERM:virtual topology, 2
TERMnoindex:Cartesian, 2
TERMnoindex:distributed graph, 2
TERMnoindex:graph, 2