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

6.2 Virtual Topologies

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

> *Rationale.* Experience with similar techniques in PARMACS [5, 8] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

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

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

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

6.3 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 5.

6.4 Overview of the Functions

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

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. A new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 5). 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.

Rationale. Similar functions are contained in EXPRESS [22] and PARMACS. (*End of rationale.*)

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

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 5, they are sufficient to implement all other topology functions. Section 6.5.7 outlines such an implementation.

6.5 Topology Constructors

6.5.1 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, int *dims, int *periods,
              int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

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, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. The call is erroneous if it specifies a grid that is larger than the group size.

6.5.2 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.

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	number of nodes in a grid (integer)
IN	ndims	number of cartesian dimensions (integer)
INOUT	dims	integer array of size ndims specifying the number of nodes in each dimension

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

The entries in the array dims are set to describe a cartesian grid with ndims dimensions and a total of 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 dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] = 0 are modified by the call.

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

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.

Example 6.1

dims before call	function call	dims on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

6.5.3 General (Graph) Constructor

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

IN	comm_old	input communicator without topology (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)

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
            int reorder, MPI_Comm *comm_graph)
```

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
            IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
```

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph 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. If the size, nnodes, of the graph is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

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 ith 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 6.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

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

Then, the input arguments are:

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

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

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

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

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

1. index,
2. edges,

which are the vectors defining the graph structure.

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

6.5.4 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 (choice)

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

```
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_UNDEFINED	no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

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

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

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

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

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

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

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

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
              int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

MPI_CARTDIM_GET(comm, ndims)

IN	comm	communicator with cartesian structure (handle)
OUT	ndims	number of dimensions of the cartesian structure (integer)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

The functions MPI_CARTDIM_GET and MPI_CART_GET return the cartesian topology information that was associated with a communicator by MPI_CART_CREATE.

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN	comm	communicator with cartesian structure (handle)
IN	maxdims	length of vectors dims, periods, and coords in the calling program (integer)
OUT	dims	number of processes for each cartesian dimension (array of integer)
OUT	periods	periodicity (true/false) for each cartesian dimension (array of logical)
OUT	coords	coordinates of calling process in cartesian structure (array of integer)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)
```

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```

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)

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

```
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)` $\geq$ `dims(i)`, it is shifted back to the interval `0` $\leq$ `coords(i)` $<$ `dims(i)` automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

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 coord in the calling program (integer)
OUT	coords	integer array (of size ndims) containing the cartesian coordinates of specified process (integer)

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

```
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.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
OUT	nneighbors	number of neighbors of specified process (integer)

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general, graph topology.

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
IN	maxneighbors	size of array neighbors (integer)
OUT	neighbors	ranks of processes that are neighbors to specified process (array of integer)

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int *neighbors)
```

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

Example 6.3 Suppose that comm is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0,1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

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

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

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

6.5.5 Cartesian Shift Coordinates

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

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)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

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.

Example 6.4 The communicator, comm, has a two-dimensional, periodic, cartesian topology associated with it. A two-dimensional array of REALs is stored one element per process, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr))
C find cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 1, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

6.5.6 Partitioning of Cartesian structures

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	communicator with cartesian structure (handle)
IN	remain_dims	the ith entry of remain_dims specifies whether the ith dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT	newcomm	communicator containing the subgrid that includes the calling process (handle)

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

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. (This function is closely related to MPI_COMM_SPLIT.)

Example 6.5 Assume that MPI_CART_CREATE(..., comm) has defined a $(2\times3\times4)$ 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.

6.5.7 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.

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)

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
              int *newrank)
```

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    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` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

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 new function for general 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, int *index, int *edges,
              int *newrank)

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 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with `color = 0` if `newrank` $\neq$ `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.*)

6.6 An Application Example

Example 6.6 The example in figure 6.1 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 all points owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor `(i-1,j)`.

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
      integer neigh_rank(num_neigh), own_position(ndims), i, j
      logical periods(ndims)
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_WORLD
C     Set process grid size and periodicity
      call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
      periods(1) = .TRUE.
      periods(2) = .TRUE.
C     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_position,ierr)
C     Look up the ranks for the neighbors.  Own process coordinates are (i,j).
C     Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
      i = own_position(1)
      j = own_position(2)
      neigh_def(1) = i-1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
        call relax (u, f)
C     Exchange data with neighbor processes
        call exchange (u, comm_cart, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

Figure 6.1: Set-up of process structure for two-dimensional parallel Poisson solver.

Chapter 7

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

7.1 Implementation information

7.1.1 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET described in Chapter 5. It is erroneous to delete these attributes or free their keys.

The list of predefined attribute keys include

MPI_TAG_UB Upper bound for tag value.

MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.

MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.

Vendors may add implementation specific parameters (such as node number, real memory size, virtual memory size, etc.)

The required parameter values are discussed in more detail below:

Tag values

Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be *at least* 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI_TAG_UB.

Host rank

The value returned for MPI_HOST gets the rank of the HOST process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a HOST, nor does it requires that a HOST exists.

IO rank

The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ANSI-C I/O operations are supported (e.g., fopen, fprintf, lseek).

If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE must be returned. If no process can provide language-standard I/O, then the value MPI_PROC_NULL must be returned. If several processes can provide I/O, then any them may be returned. The same value (rank) need not be returned by all processes.

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT	name	A unique specifier for the actual (as opposed to virtual) node.
OUT	resultlen	Length (in printable characters) of the result returned in name

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

```
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR
```

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the ouput argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

7.2 Error handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an **MPI exception.**

A user can associate an error handler with a communicator. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for a communication with this communicator. MPI calls that are not related to any communicator are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to communicators is purely local: different processes may attach different error handlers to the same communicator.

A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

> *Advice to implementors.* A good quality implementation will, to the greatest possible, extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with communicators, and to test which error handler is associated with a communicator.

MPI_ERRHANDLER_CREATE(function, errhandler)

IN	function	user defined error handling procedure
OUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_create(MPI_Handler_function *function,
              MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

Register the user routine function for use as an MPI exception handler. Returns in errhandler a handle to the registered exception handler.

Advice to implementors. The handle returned may contain the address of the error handling routine. This call is superfluous in C, which has a referencing operator, but is necessary in Fortran. (*End of advice to implementors.*)

The user routine should be a C function of type MPI_Handler_function, which is defined as:

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine. The remaining arguments are "stdargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran.

Rationale. The variable argument list is provided because it provides an ANSI-standard hook for providing additional information to the error handler; without this hook, ANSI C prohibits additional arguments. (*End of rationale.*)

MPI_ERRHANDLER_SET(comm, errhandler)

IN	comm	communicator to set the error handler for (handle)
IN	errhandler	new MPI error handler for communicator (handle)

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

Associates the new error handler errorhandler with communicator comm at the calling process. Note that an error handler is always associated with the communicator.

MPI_ERRHANDLER_GET(comm, errhandler)

IN	comm	communicator to get the error handler from (handle)
OUT	errhandler	MPI error handler currently associated with communicator (handle)

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

Returns in errhandler (a handle to) the error handler that is currently associated with communicator comm.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

MPI_ERRHANDLER_FREE(errhandler)

IN	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
```

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all communicators associated with it have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

IN	errorcode	Error code returned by an MPI routine
OUT	string	Text that corresponds to the errorcode
OUT	resultlen	Length (in printable characters) of the result returned in string

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
```

```
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
```

Returns the error string associated with an error code. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing

the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

7.3 Error codes and classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts an error code into one of a small set of specified values, called *error classes*. Valid error classes include

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI error
MPI_ERR_LASTCODE	Last standard error code

An implementation is free to define more error classes; however, the standard error classes must be used where appropriate. The error classes satisfy,

$$0 = \text{MPI\_SUCCESS} < \text{MPI\_ERR\_...} \leq \text{MPI\_ERR\_LASTCODE}.$$

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

MPI_ERROR_CLASS(errorcode, errorclass)

IN	errorcode	Error code returned by an MPI routine
OUT	errorclass	Error class associated with errorcode

```
int MPI_Error_class(int errorcode, int *errorclass)
```

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

7.4 Timers

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers.

MPI_WTIME()

```
double MPI_Wtime(void)
```

```
DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
   double starttime, endtime;
   starttime = double MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = double MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time."

MPI_WTICK()

```
double MPI_Wtick(void)
```

```
DOUBLE PRECISION MPI_WTICK()
```

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3}.

7.5 Startup

One goal of MPI is to achieve ***source code portability***. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI_INITIALIZED).

All MPI programs must contain a call to MPI_init; this routine must be called before any other MPI routine (apart from `MPI_INITIALIZED`) is called. The version for ANSI C accepts the argc and argv that are provided by the arguments to `main`:

```
MPI_init( argc, argv );
```

The Fortran version takes only IERROR.

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

This routines cleans up all MPI state. Once this routine is called, no MPI routine (even MPI_INIT) may be called. The user must ensure that all pending communications involving a process complete before the process calls MPI_FINALIZE.

MPI_INITIALIZED(flag)

OUT	flag	Flag is true if MPI_INIT has been called and false otherwise.

```
int MPI_Initialized(int *flag)
```

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

This routine may be used to determine whether MPI_INIT has been called. It is the *only* routine that may be called before MPI_INIT is called.

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program or an `abort(errorcode)`.

MPI implementations are required to define the behavior of MPI_ABORT at least for a comm of MPI_COMM_WORLD. MPI implementations may ignore the comm argument and act as if the comm was MPI_COMM_WORLD.

Chapter 8

Profiling Interface

8.1 Requirements

To meet the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix "`MPI_`") should also be accessible with the prefix "`PMPI_`".

2. ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.

3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economise by implementing it only for the lowest level routines.

4. where the implementation of different language bindings is is done through a layered approach (e.g. the Fortran binding is a set of "wrapper" functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

   This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. provide a no-op routine MPI_PCONTROL in the MPI library.

8.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to

the source code which implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this chapter is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others which would be equally valid).

8.3 Logic of the design

Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the MPI calls which are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

8.3.1 Miscellaneous control of profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation
- Adding user events to a trace file.

These requirements are met by use of the MPI_PCONTROL.

MPI_PCONTROL(level, ...)

IN	level	Profiling level

```
int MPI_Pcontrol(const int level, ...)
```

```
MPI_PCONTROL(level)
    INTEGER LEVEL, ...
```

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics which will be provided by calls to MPI_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

- `level==0` Profiling is disabled.
- `level==1` Profiling is enabled at a normal default level of detail.
- `level==2` Profile buffers are flushed. (This may be a no-op in some profilers).
- All other values of `level` have profile library defined effects and additional arguments.

We also request that the default state after MPI_INIT has been called is for profiling to be enabled at the normal default level. (i.e. as if MPI_PCONTROL had just been called with the argument 1). This allows users to link with a profiling library and obtain profile output without having to modify their source code at all.

The provision of MPI_PCONTROL as a no-op in the standard MPI library allows them to modify their source code to obtain more detailed profiling information, but still be able to link exactly the same code against the standard MPI library.

8.4 Examples

8.4.1 Profiler implementation

Suppose that the profiler wishes to accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function. This could trivially be achieved thus

```
static int totalBytes;
static double totalTime;

int MPI_SEND(void * buffer, const int count, MPI_Datatype datatype,
             int dest, int tag, MPI_comm comm)
{
   double tstart = MPI_Wtime();       /* Pass on all the arguments */
   int extent;
   int result    = PMPI_Send(buffer,count,datatype,dest,tag,comm);

                                      /* Accumulate byte count */
   totalBytes += count * MPI_Type_size(datatype,&extent);
                                      /* and time */
   totalTime  += MPI_Wtime() - tstart;
```

```
   return result;
}
```

8.4.2 MPI library implementation

On a Unix system, in which the MPI library is implemented in C, then there are various possible options, of which two of the most obvious are presented here. Which is better depends on whether the linker and compiler support weak symbols.

Systems with weak symbols

If the compiler and linker support weak external symbols (e.g. Solaris 2.x, other system V.4 machines), then only a single library is required through the use of `#pragma weak` thus

```
#pragma weak MPI_Example = PMPI_Example

int PMPI_Example(/* appropriate args */)
{
    /* Useful content */
}
```

The effect of this `#pragma` is to define the external symbol `MPI_Example` as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library), however if no other definition exists, then the linker will use the weak definition.

Systems without weak symbols

In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus

```
#ifdef PROFILELIB
#    ifdef __STDC__
#        define FUNCTION(name) P##name
#    else
#        define FUNCTION(name) P/**/name
#    endif
#else
#    define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions which she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

```
% cc ... -lmyprof -lpmpi -lmpi
```

Here `libmyprof.a` contains the profiler functions which intercept some of the MPI functions. `libpmpi.a` contains the "name shifted" MPI functions, and `libmpi.a` contains the normal definitions of the MPI functions.

8.4.3 Complications

Multiple counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g. a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function which was called from a profiling function. This could lead to "double counting" of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g. it might allow one to answer the question "How much time is spent in the point to point routines when they're called from collective functions ?"), we have decided not to enforce any restrictions on the author of the MPI library which would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code which remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded !)

Linker oddities

The Unix linker traditionally operates in one pass : the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be ared out of the base library and into the profiling one.

8.5 Multiple levels of interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation which would allow multiple levels of call interception, however we were unable to construct an implementation of this which did not have the following disadvantages

- assuming a particular implementation language.
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.

Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single level implementation detailed above.

Bibliography

[1] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.

[5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[6] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[7] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Journal of Parallel Computing*, 1994. to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[8] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the parmacs message–passing library. *Parallel Computing, Special issue on message–passing interfaces*, to appear.

[9] S. Chittor and R. J. Enbody. Performance evaluation of mesh–connected wormhole–routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990.

[10] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991.

[11] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993.

[12] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[13] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of MPI. Technical report, Argonne National Laboratory, 1993.

[14] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[15] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[16] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991.

[17] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI. In *1994 International Conference on Parallel Processing*, 1994.

[18] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[19] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[20] O. Krämer and H. Mühlenbein. Mapping strategies in message–based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989.

[21] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[22] Parasoft Corporation, Pasadena, CA. *Express User's Guide*, version 3.2.5 edition, 1992.

[23] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[24] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.

[25] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[26] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.

[27] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P . Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 1994. (Invited Paper, to appear in Special Issue on Message Passing).

[28] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Charles H. Still, Alvin P. Leung, and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing. Technical Report UCRL-JC-106725 (revised 9/92, 12/93, 4/94), Lawrence Livermore National Laboratory, March 1991. To appear in Concurrency: Practice & Experience.

[29] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.

Annex A

Language Binding

A.1 Introduction

In this section we summarize the specific bindings for both Fortran and C. We present first the C bindings, then the Fortran bindings. Listings are alphabetical within chapter.

A.2 Defined Constants for C and Fortran

These are required defined constants, to be defined in the files `mpi.h` (for C) and `mpif.h` (for Fortran).

```
/* return codes (both C and Fortran) */
MPI_SUCCESS
MPI_ERR_BUFFER
MPI_ERR_COUNT
MPI_ERR_TYPE
MPI_ERR_TAG
MPI_ERR_COMM
MPI_ERR_RANK
MPI_ERR_REQUEST
MPI_ERR_ROOT
MPI_ERR_GROUP
MPI_ERR_OP
MPI_ERR_TOPOLOGY
MPI_ERR_DIMS
MPI_ERR_ARG
MPI_ERR_UNKNOWN
MPI_ERR_TRUNCATE
MPI_ERR_OTHER
MPI_ERR_INTERN
MPI_ERR_LASTCODE

/* assorted constants (both C and Fortran) */
MPI_BOTTOM
MPI_PROC_NULL
```

```
MPI_ANY_SOURCE
MPI_ANY_TAG
MPI_UNDEFINED
MPI_UB
MPI_LB

/* status size and reserved index values (Fortran) */
MPI_STATUS_SIZE
MPI_SOURCE
MPI_TAG

/* Error-handling specifiers (C and Fortran) */
MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN

/* Maximum sizes for strings */
MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING

/* elementary datatypes (C) */
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED



/* elementary datatypes (Fortran) */
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_DOUBLE_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
MPI_BYTE
MPI_PACKED

/* datatypes for reduction functions (C) */
```

```
MPI_FLOAT_INT
MPI_DOUBLE_INT
MPI_LONG_INT
MPI_2INT
MPI_SHORT_INT
MPI_LONG_DOUBLE_INT

/* datatypes for reduction functions (Fortran) */
MPI_2REAL
MPI_2DOUBLE_PRECISION
MPI_2INTEGER
MPI_2COMPLEX

/* optional datatypes (Fortran) */

MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_REAL2
MPI_REAL4
MPI_REAL8

/* optional datatypes (C) */
MPI_LONG_LONG_INT

/* reserved communicators (C and Fortran) */
MPI_COMM_WORLD
MPI_COMM_SELF

/* results of communicator and group comparisons */

MPI_IDENT
MPI_CONGRUENT
MPI_SIMILAR
MPI_UNEQUAL

/* environmental inquiry keys (C and Fortran) */
MPI_TAG_UB
MPI_IO
MPI_HOST

/* collective operations (C and Fortran) */
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MINLOC
```

```
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR

/* Null handles */
MPI_GROUP_NULL
MPI_COMM_NULL
MPI_DATATYPE_NULL
MPI_REQUEST_NULL
MPI_OP_NULL
MPI_ERRHANDLER_NULL

/* Empty group */
MPI_GROUP_EMPTY

/* topologies (C and Fortran) */
MPI_GRAPH
MPI_CART
```

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

```
/* opaque types (C) */
MPI_Aint
MPI_Status

/* handles to assorted structures (C) */
MPI_Group
MPI_Comm
MPI_Datatype
MPI_Request
MPI_Op

/* prototypes for user-defined functions (C) */
typedef int  MPI_Copy_function(MPI_Comm *oldcomm, *newcomm, int *keyval,
                    void *extra_state)
typedef int  MPI_Delete_function(MPI_Comm *comm, int *keyval,
                                 void *extra_state)}
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
typedef void MPI_User_function( void *invec, void *inoutvec, int *len,
                     MPI_Datatype *datatype);
```

For Fortran, here are examples of how each of the user-defined functions should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

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

The copy-function argument to MPI_KEYVAL_CREATE should be declared like this:

```
FUNCTION COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                ATTRIBUTE_VAL_OUT, FLAG)
 INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
 LOGICAL FLAG
```

The delete-function argument to MPI_KEYVAL_CREATE should be declared like this:

```
FUNCTION DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE)
 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE
```

A.3 C bindings for Point-to-Point Communication

These are presented here in the order of their appearance in the chapter.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)

int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Buffer_attach( void* buffer, int size)

int MPI_Buffer_detach( void** buffer, int* size)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

int MPI_Request_free(MPI_Request *request)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
            MPI_Status *status)

int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
            int *flag, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
            MPI_Status *array_of_statuses)

int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
            MPI_Status *array_of_statuses)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
            int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
            int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
            MPI_Status *status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Cancel(MPI_Request *request)

int MPI_Test_cancelled(MPI_Status status, int *flag)

int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
            int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Start(MPI_Request *request)

int MPI_Startall(int count, MPI_Request *array_of_requests)

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            int dest, int sendtag, void *recvbuf, int recvcount,
            MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
            MPI_Comm comm, MPI_Status *status)
```

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
             int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
             MPI_Status *status)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_vector(int count, int blocklength, int stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_indexed(int count, int *array_of_blocklengths,
             int *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_struct(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
             MPI_Datatype *newtype)

int MPI_Address(void* location, MPI_Aint *address)

int MPI_Type_extent(MPI_Datatype datatype, int *extent)

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_count(MPI_Datatype datatype, int *count)

int MPI_Type_lb(MPI_Datatype datatype, int* displacement)

int MPI_Type_ub(MPI_Datatype datatype, int* displacement)

int MPI_Type_commit(MPI_Datatype *datatype)

int MPI_Type_free(MPI_Datatype *datatype)

int MPI_Get_elements(MPI_Status status, MPI_Datatype datatype, int *count)

int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
             int outcount, MPI_Datatype datatype, MPI_Comm comm)

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)
```

A.4 C Bindings for Collective Communication

```
int MPI_Barrier(MPI_Comm comm )

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm )

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

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

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
              MPI_Datatype sendtype, void* recvbuf, int recvcount,
              MPI_Datatype recvtype, int root, MPI_Comm comm)

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

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

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

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
              MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
              int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

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

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

int MPI_Op_free( MPI_Op *op)

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

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

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

A.5 C Bindings for Groups, Contexts, and Communicators

```
int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_rank(MPI_Group group, int *rank)

int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)

int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)

int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

int MPI_Group_free(MPI_Group *group)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

int MPI_Comm_compare(MPI_Comm comm1, comm2, int *result)

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

int MPI_Comm_free(MPI_Comm *comm)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
              MPI_Comm peer_comm, int remote_leader, int tag,
              MPI_Comm *newintercomm)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
              MPI_Comm *newintracomm)
```

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
              *delete_fn, int *keyval, void* extra_state)

int MPI_Keyval_free(int *keyval)

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)

int MPI_Attr_get(MPI_Comm comm, int keyval, void **attribute_val, int *flag)

int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

A.6 C Bindings for Process Topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
              int reorder, MPI_Comm *comm_cart)

int MPI_Dims_create(int nnodes, int ndims, int *dims)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
              int reorder, MPI_Comm *comm_graph)

int MPI_Topo_test(MPI_Comm comm, int *status)

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

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
              int *edges)

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int *neighbors)

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

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
              int *newrank)

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
              int *newrank)
```

A.7 C bindings for Environmental Inquiry

```
int MPI_Get_processor_name(char *name, int *resultlen)

int MPI_Errhandler_create(MPI_Handler_function *function,
             MPI_Errhandler *errhandler)

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

int MPI_Error_string(int errorcode, char *string, int *resultlen)

int MPI_Error_class(int errorcode, int *errorclass)

int double MPI_Wtime(void)

int double MPI_Wtick(void)

int MPI_Init(int *argc, char ***argv)

int MPI_Finalize(void)

int MPI_Initialized(int *flag)

int MPI_Abort(MPI_Comm comm, int errorcode)
```

A.8 C Bindings for Profiling

```
int MPI_Pcontrol(const int level, ...)
```

A.9 Fortran Bindings for Point-to-Point Communication

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    ARRAY_OF_TYPES(*), NEWTYPE, IERROR

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)
    INTEGER DATATYPE, COUNT, IERROR

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTCOUNT, POSITION, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTCOUNT, POSITION, COMM, IERROR

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

A.10 Fortran Bindings for Collective Communication

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR

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

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

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

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

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

```
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR

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

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

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

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR

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

MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_FREE( OP, IERROR)
    INTEGER OP, IERROR

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

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

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

A.11 Fortran Bindings for Groups, Contexts, etc.

```
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

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

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
             NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

A.12 Fortran Bindings for Process Topologies

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
             IERROR)
```

```
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER

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

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

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)

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

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

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)

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

A.13 Fortran Bindings for Environmental Inquiry

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN, IERROR

MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
    EXTERNAL FUNCTION
```

```
    INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING

MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR

DOUBLE PRECISION MPI_WTIME()

DOUBLE PRECISION MPI_WTICK()

MPI_INIT(IERROR)
    INTEGER IERROR

MPI_FINALIZE(IERROR)
    INTEGER IERROR

MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

A.14 Fortran Bindings for Profiling

```
MPI_PCONTROL(level)
    INTEGER LEVEL, ...
```

MPI Function Index

MPI_ABORT, 197
MPI_ADDRESS, 67
MPI_ALLGATHER, 107
MPI_ALLGATHERV, 108
MPI_ALLREDUCE, 122
MPI_ALLTOALL, 109
MPI_ALLTOALLV, 110
MPI_ATTR_DELETE, 169
MPI_ATTR_GET, 169
MPI_ATTR_PUT, 168

MPI_BARRIER, 93
MPI_BCAST, 93
MPI_BSEND, 27
MPI_BSEND_INIT, 53
MPI_BUFFER_ATTACH, 33
MPI_BUFFER_DETACH, 33

MPI_CANCEL, 51
MPI_CART_COORDS, 182
MPI_CART_CREATE, 177
MPI_CART_GET, 181
MPI_CART_MAP, 186
MPI_CART_RANK, 182
MPI_CART_SHIFT, 184
MPI_CART_SUB, 185
MPI_CARTDIM_GET, 181
MPI_COMM_COMPARE, 142
MPI_COMM_CREATE, 143
MPI_COMM_DUP, 143
MPI_COMM_FREE, 145
MPI_COMM_GROUP, 137
MPI_COMM_RANK, 142
MPI_COMM_REMOTE_GROUP, 155
MPI_COMM_REMOTE_SIZE, 155
MPI_COMM_SIZE, 141
MPI_COMM_SPLIT, 144
MPI_COMM_TEST_INTER, 154

MPI_DIMS_CREATE, 177

MPI_ERRHANDLER_CREATE, 192
MPI_ERRHANDLER_FREE, 193
MPI_ERRHANDLER_GET, 193
MPI_ERRHANDLER_SET, 192
MPI_ERROR_CLASS, 195
MPI_ERROR_STRING, 193

MPI_FINALIZE, 196

MPI_GATHER, 94
MPI_GATHERV, 95
MPI_GET_COUNT, 21
MPI_GET_ELEMENTS, 73
MPI_GET_PROCESSOR_NAME, 190
MPI_GRAPH_CREATE, 178
MPI_GRAPH_GET, 181
MPI_GRAPH_MAP, 187
MPI_GRAPH_NEIGHBORS, 183
MPI_GRAPH_NEIGHBORS_COUNT, 183
MPI_GRAPHDIMS_GET, 180
MPI_GROUP_COMPARE, 136
MPI_GROUP_DIFFERENCE, 138
MPI_GROUP_EXCL, 139
MPI_GROUP_FREE, 140
MPI_GROUP_INCL, 138
MPI_GROUP_INTERSECTION, 137
MPI_GROUP_RANGE_EXCL, 140
MPI_GROUP_RANGE_INCL, 139
MPI_GROUP_RANK, 135
MPI_GROUP_SIZE, 135
MPI_GROUP_TRANSLATE_RANKS, 136
MPI_GROUP_UNION, 137

MPI_IBSEND, 37
MPI_INIT, 196
MPI_INITIALIZED, 197
MPI_INTERCOMM_CREATE, 157
MPI_INTERCOMM_MERGE, 157
MPI_IPROBE, 48
MPI_IRECV, 38
MPI_IRSEND, 38

MPI_ISEND, 36
MPI_ISSEND, 37

MPI_KEYVAL_CREATE, 166
MPI_KEYVAL_FREE, 168

MPI_OP_CREATE, 118
MPI_OP_FREE, 120

MPI_PACK, 83
MPI_PACK_SIZE, 86
MPI_PCONTROL, 199
MPI_PROBE, 49

MPI_RECV, 19
MPI_RECV_INIT, 55
MPI_REDUCE, 111
MPI_REDUCE_SCATTER, 123
MPI_REQUEST_FREE, 41
MPI_RSEND, 28
MPI_RSEND_INIT, 54

MPI_SCAN, 124
MPI_SCATTER, 103
MPI_SCATTERV, 104
MPI_SEND, 16
MPI_SEND_INIT, 53
MPI_SENDRECV, 57
MPI_SENDRECV_REPLACE, 58
MPI_SSEND, 27
MPI_SSEND_INIT, 54
MPI_START, 55
MPI_STARTALL, 56

MPI_TEST, 40
MPI_TEST_CANCELLED, 52
MPI_TESTALL, 45
MPI_TESTANY, 44
MPI_TESTSOME, 46
MPI_TOPO_TEST, 180
MPI_TYPE_COMMIT, 70
MPI_TYPE_CONTIGUOUS, 60
MPI_TYPE_COUNT, 69
MPI_TYPE_EXTENT, 68
MPI_TYPE_FREE, 71
MPI_TYPE_HINDEXED, 65
MPI_TYPE_HVECTOR, 63
MPI_TYPE_INDEXED, 64
MPI_TYPE_LB, 70
MPI_TYPE_SIZE, 68
MPI_TYPE_STRUCT, 66
MPI_TYPE_UB, 70
MPI_TYPE_VECTOR, 61

MPI_UNPACK, 84

MPI_WAIT, 39
MPI_WAITALL, 44
MPI_WAITANY, 43
MPI_WAITSOME, 46
MPI_WTICK, 195
MPI_WTIME, 195

https://web.archive.org/web/20040331061807/http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf

MATLAB®

The Language of Technical Computing

- Computation
- Visualization
- Programming

External Interfaces

Version 6



How to Contact The MathWorks:



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup



`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information



508-647-7000 Phone



508-647-7001 Fax



The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB External Interfaces

© COPYRIGHT 1984 - 2003 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:

December 1996	First printing	New for MATLAB 5
July 1997	Online only	Revised for MATLAB 5.1
January 1998	Second printing	Revised for MATLAB 5.2
October 1998	Third printing	Revised for MATLAB 5.3 (Release 11)
November 2000	Fourth printing	Revised and renamed for MATLAB 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
September 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)

External Interfaces/API

MATLAB® provides interfaces to external routines written in other programming languages, data that needs to be shared with external routines, clients or servers communicating via Component Object Model (COM) or Dynamic Data Exchange (DDE), and peripheral devices that communicate directly with MATLAB. Much of this interface capability was formerly referred to under the title of the MATLAB Application Program Interface, or API.

This section introduces the external interfaces that are available with MATLAB. It also contains numerous C and Fortran examples that highlight this functionality. The topics included are

Importing and Exporting Data	Describes techniques for importing data to and exporting data from the MATLAB environment using MAT-files.
MATLAB Interface to Generic DLLs	Describes how to call functions in external, shared `.dll` libraries from MATLAB.
Calling C and Fortran Programs from MATLAB	Describes how to build C and Fortran subroutines into callable MEX-files.
Creating C Language MEX-Files	Shows the components of a C MEX-file and provides specific examples of MEX-files written in C.
Creating Fortran MEX-Files	Shows the components of a Fortran MEX-file and provides specific examples of MEX-files written in Fortran.
Calling MATLAB from C and Fortran Programs	Discusses how to use the MATLAB engine library to call MATLAB from C and Fortran programs.
Calling Java from MATLAB	Describes how to use the MATLAB interface to Java classes and objects.

Ex.1049 / Page 251 of 748
Intel Corporation et al

COM and DDE Support	Describes how to use Component Object Model (COM) and Dynamic Data Exchange (DDE) with MATLAB.
Serial Port I/O	Describes how to communicate with peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port.

Contents

1 Importing and Exporting Data

Using MAT-Files **1-2**
Importing Data to MATLAB **1-2**
Exporting Data from MATLAB **1-3**
Exchanging Data Files Between Platforms **1-4**
Reading and Writing MAT-Files **1-5**
Finding Associated Files **1-8**

Examples of MAT-Files **1-10**
Creating a MAT-File in C **1-10**
Reading a MAT-File in C **1-15**
Creating a MAT-File in Fortran **1-19**
Reading a MAT-File in Fortran **1-24**

Compiling and Linking MAT-File Programs **1-28**
Masking Floating Point Exceptions **1-28**
Compiling and Linking on UNIX **1-29**
Compiling and Linking on Windows **1-30**

2 MATLAB Interface to Generic DLLs

Overview **2-2**

Loading and Unloading the Library **2-3**
Loading the Library **2-3**
Unloading the Library **2-3**

Getting Information About the Library **2-4**

Invoking Library Functions **2-7**

Passing Arguments **2-8**
Some General Rules **2-9**
Passing References **2-10**

Data Conversion **2-11**
Primitive Types **2-11**
Enumerated Types **2-15**
Structures **2-15**
Creating References **2-21**
Reference Pointers **2-25**

3 Calling C and Fortran Programs from MATLAB

Introducing MEX-Files **3-2**
Using MEX-Files **3-2**
The Distinction Between mx and mex Prefixes **3-3**

MATLAB Data **3-5**
The MATLAB Array **3-5**
Data Storage **3-5**
Data Types in MATLAB **3-6**
Using Data Types **3-8**

Building MEX-Files **3-10**
Compiler Requirements **3-10**
Testing Your Configuration on UNIX **3-11**
Testing Your Configuration on Windows **3-13**
Specifying an Options File **3-16**

Custom Building MEX-Files **3-19**
Who Should Read This Chapter **3-19**
MEX Script Switches **3-19**
Default Options File on UNIX **3-21**
Default Options File on Windows **3-22**
Custom Building on UNIX **3-23**
Custom Building on Windows **3-25**

Ex.1049 / Page 254 of 748
Intel Corporation et al

Troubleshooting **3-31**
Configuration Issues **3-31**
Understanding MEX-File Problems **3-33**
Compiler and Platform-Specific Issues **3-36**
Memory Management Compatibility Issues **3-36**

Additional Information **3-41**
Files and Directories - UNIX Systems **3-41**
Files and Directories - Windows Systems **3-43**
Examples **3-45**
Technical Support **3-47**

4 Creating C Language MEX-Files

C MEX-Files **4-2**
The Components of a C MEX-File **4-2**
Required Arguments to a MEX-File **4-4**

Examples of C MEX-Files **4-6**
A First Example — Passing a Scalar **4-7**
Passing Strings **4-10**
Passing Two or More Inputs or Outputs **4-13**
Passing Structures and Cell Arrays **4-16**
Handling Complex Data **4-20**
Handling 8-,16-, and 32-Bit Data **4-23**
Manipulating Multidimensional Numerical Arrays **4-25**
Handling Sparse Matrices **4-29**
Calling Functions from C MEX-Files **4-33**

Advanced Topics **4-36**
Help Files **4-36**
Linking Multiple Files **4-36**
Workspace for MEX-File Functions **4-37**
Memory Management **4-37**
Using LAPACK and BLAS Functions **4-40**

Ex.1049 / Page 255 of 748
Intel Corporation et al

Debugging C Language MEX-Files **4-48**
Debugging on UNIX **4-48**
Debugging on Windows **4-49**

5 Creating Fortran MEX-Files

Fortran MEX-Files **5-2**
The Components of a Fortran MEX-File **5-2**
The %val Construct **5-7**

Examples of Fortran MEX-Files **5-8**
A First Example — Passing a Scalar **5-9**
Passing Strings **5-11**
Passing Arrays of Strings **5-13**
Passing Matrices **5-16**
Passing Two or More Inputs or Outputs **5-18**
Handling Complex Data **5-21**
Dynamically Allocating Memory **5-25**
Handling Sparse Matrices **5-27**
Calling Functions from Fortran MEX-Files **5-31**

Advanced Topics **5-35**
Help Files **5-35**
Linking Multiple Files **5-35**
Workspace for MEX-File Functions **5-35**
Memory Management **5-36**

Debugging Fortran Language MEX-Files **5-37**
Debugging on UNIX **5-37**
Debugging on Windows **5-38**

Ex.1049 / Page 256 of 748
Intel Corporation et al

6 Calling MATLAB from C and Fortran Programs

Using the MATLAB Engine **6-2**
The Engine Library **6-2**
GUI-Intensive Applications **6-4**

Examples of Calling Engine Functions **6-5**
Calling MATLAB From a C Application **6-5**
Calling MATLAB From a Fortran Application **6-10**
Attaching to an Existing MATLAB Session **6-14**

Compiling and Linking Engine Programs **6-16**
Masking Floating-Point Exceptions **6-16**
Compiling and Linking on UNIX **6-17**
Compiling and Linking on Windows **6-18**

7 Calling Java from MATLAB

Using Java from MATLAB: An Overview **7-2**
Java Interface Is Integral to MATLAB **7-2**
Benefits of the MATLAB Java Interface **7-2**
Who Should Use the MATLAB Java Interface **7-2**
To Learn More About Java Programming **7-2**
Platform Support for the Java Virtual Machine **7-3**

Bringing Java Classes into MATLAB **7-4**
Sources of Java Classes **7-4**
Defining New Java Classes **7-4**
Making Java Classes Available to MATLAB **7-5**
Loading Java Class Definitions **7-6**
Simplifying Java Class Names **7-7**

Creating and Using Java Objects **7-9**
Constructing Java Objects **7-9**
Concatenating Java Objects **7-11**

Ex.1049 / Page 257 of 748
Intel Corporation et al

Saving and Loading Java Objects to MAT-Files 7-13
Finding the Public Data Fields of an Object 7-14
Accessing Private and Public Data 7-15
Determining the Class of an Object 7-17

Invoking Methods on Java Objects 7-18
Using Java and MATLAB Calling Syntax 7-18
Invoking Static Methods on Java Classes 7-20
Obtaining Information About Methods 7-21
Java Methods That Affect MATLAB Commands 7-25
How MATLAB Handles Undefined Methods 7-26
How MATLAB Handles Java Exceptions 7-27

Working with Java Arrays 7-28
How MATLAB Represents the Java Array 7-29
Creating an Array of Objects Within MATLAB 7-33
Accessing Elements of a Java Array 7-35
Assigning to a Java Array 7-38
Concatenating Java Arrays 7-42
Creating a New Array Reference 7-43
Creating a Copy of a Java Array 7-44

Passing Data to a Java Method 7-46
Conversion of MATLAB Argument Data 7-46
Passing Built-In Data Types 7-48
Passing String Arguments 7-49
Passing Java Objects 7-50
Other Data Conversion Topics 7-53
Passing Data to Overloaded Methods 7-54

Handling Data Returned from a Java Method 7-56
Conversion of Java Return Data 7-56
Built-In Data Types 7-57
Java Objects 7-57
Converting Objects to MATLAB Data Types 7-57

Introduction to Programming Examples 7-62

Ex.1049 / Page 258 of 748
Intel Corporation et al

Example – Reading a URL **7-63**
Description of URLdemo **7-63**
Running the Example **7-64**

Example – Finding an Internet Protocol Address **7-66**
Description of resolveip **7-66**
Running the Example **7-67**

Example – Communicating Through a Serial Port **7-68**
Description of Serial Example **7-69**
Running the serialexample Program **7-72**

Example – Creating and Using a Phone Book **7-73**
Description of Function phonebook **7-74**
Description of Function pb_lookup **7-78**
Description of Function pb_add **7-79**
Description of Function pb_remove **7-80**
Description of Function pb_change **7-80**
Description of Function pb_listall **7-82**
Description of Function pb_display **7-82**
Description of Function pb_keyfilter **7-83**
Running the phonebook Program **7-83**

8 COM and DDE Support

Introducing MATLAB COM Integration **8-2**
Concepts and Terminology **8-2**
Overview of MATLAB COM Support **8-4**

MATLAB COM Client Support **8-8**
Creating COM Objects **8-8**
Object Properties **8-10**
Invoking Methods **8-17**
COM Control Events **8-20**
Identifying Objects **8-23**
Saving Your Work **8-24**

Ex.1049 / Page 259 of 748
Intel Corporation et al

Releasing COM Interfaces and Objects 8-25
Writing Event Handlers 8-26
Examples of MATLAB as an Automation Client 8-30

Additional COM Client Information 8-32
Using COM Collections 8-32
Converting Data 8-33
Using MATLAB as a DCOM Server Client 8-34
MATLAB COM Support Limitations 8-35

MATLAB Automation Server Support 8-36
Creating the Automation Server 8-36
Automation Server Functions 8-37
MATLAB Automation Properties 8-39

Additional Automation Server Information 8-40
Creating the Server Manually 8-40
Specifying a Shared or Dedicated Server 8-40
Using MATLAB as a DCOM Server 8-41

Dynamic Data Exchange (DDE) 8-42
DDE Concepts and Terminology 8-42
Accessing MATLAB as a Server 8-44
The DDE Name Hierarchy 8-45
Example: Using Visual Basic and the MATLAB DDE Server 8-48
Using MATLAB as a Client 8-50
DDE Advisory Links 8-51

9 Serial Port I/O

Introduction 9-2
What Is the MATLAB Serial Port Interface? 9-2
Supported Serial Port Interface Standards 9-2
Supported Platforms 9-2
Using the Examples with Your Device 9-3

Ex.1049 / Page 260 of 748
Intel Corporation et al

Overview of the Serial Port **9-4**
What Is Serial Communication? **9-4**
The Serial Port Interface Standard **9-4**
Connecting Two Devices with a Serial Cable **9-5**
Serial Port Signals and Pin Assignments **9-6**
Serial Data Format **9-11**
Finding Serial Port Information for Your Platform **9-15**
Selected Bibliography **9-17**

Getting Started with Serial I/O **9-18**
Example: Getting Started **9-18**
The Serial Port Session **9-19**
Configuring and Returning Properties **9-20**

Creating a Serial Port Object **9-24**
Configuring Properties During Object Creation **9-25**
The Serial Port Object Display **9-25**
Creating an Array of Serial Port Objects **9-26**

Connecting to the Device **9-27**

Configuring Communication Settings **9-28**

Writing and Reading Data **9-29**
Example: Introduction to Writing and Reading Data **9-29**
Controlling Access to the MATLAB Command Line **9-29**
Writing Data **9-31**
Reading Data **9-36**
Example: Writing and Reading Text Data **9-42**
Example: Parsing Input Data Using strread **9-44**
Example: Reading Binary Data **9-45**

Events and Callbacks **9-48**
Example: Introduction to Events and Callbacks **9-48**
Event Types and Callback Properties **9-49**
Storing Event Information **9-50**
Creating and Executing Callback Functions **9-52**
Enabling Callback Functions After They Error **9-53**
Example: Using Events and Callbacks **9-54**

Ex.1049 / Page 261 of 748
Intel Corporation et al

Using Control Pins **9-56**
Signaling the Presence of Connected Devices **9-56**
Controlling the Flow of Data: Handshaking **9-59**

Debugging: Recording Information to Disk **9-62**
Example: Introduction to Recording Information **9-62**
Creating Multiple Record Files **9-63**
Specifying a Filename **9-63**
The Record File Format **9-64**
Example: Recording Information to Disk **9-65**

Saving and Loading **9-68**
Using Serial Port Objects on Different Platforms **9-68**

Disconnecting and Cleaning Up **9-69**
Disconnecting a Serial Port Object **9-69**
Cleaning Up the MATLAB Environment **9-69**

Property Reference **9-70**
The Property Reference Page Format **9-70**
Serial Port Object Properties **9-71**

1

Importing and Exporting Data

You can use MAT-files, the data file format MATLAB uses for saving data to disk, to import data to and export data from the MATLAB environment. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms in a highly portable manner. In addition, they provide a means to import and export your data to other stand-alone MATLAB applications.

Using MAT-Files (p. 1-2)	Methods of importing and exporting MATLAB data, and MAT-file routines that enable you to do this
Examples of MAT-Files (p. 1-10)	Programs to create and read a MAT-file in C and Fortran
Compiling and Linking MAT-File Programs (p. 1-28)	Compiling and linking on Windows and UNIX

Using MAT-Files

This section describes the various techniques for importing data to and exporting data from the MATLAB environment. The main topics that are discussed are

- "Importing Data to MATLAB" on page 1-2
- "Exporting Data from MATLAB" on page 1-3
- "Exchanging Data Files Between Platforms" on page 1-4
- "Reading and Writing MAT-Files" on page 1-5
- "Finding Associated Files" on page 1-8

The most important approach to importing and exporting data involves the use of MAT-files, the data file format that MATLAB uses for saving data to your disk. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms and for importing and exporting your data to other stand-alone MATLAB applications.

To simplify your use of MAT-files in applications outside of MATLAB, we have developed a library of access routines with a `mat` prefix that you can use in your own C or Fortran programs to read and write MAT-files. Programs that access MAT-files also use the `mx` prefixed API routines discussed in Chapter 4, "Creating C Language MEX-Files" and Chapter 5, "Creating Fortran MEX-Files."

Importing Data to MATLAB

You can introduce data from other programs into MATLAB by several methods. The best method for importing data depends on how much data there is, whether the data is already in machine-readable form, and what format the data is in. Here are some choices. Select the one that best meets your needs:

- **Enter the data as an explicit list of elements.**

  If you have a small amount of data, less than 10-15 elements, it is easy to type the data explicitly using brackets [ ]. This method is awkward for larger amounts of data because you can't edit your input if you make a mistake.

- **Create data in an M-file.**

  Use your text editor to create an M-file that enters your data as an explicit list of elements. This method is useful when the data isn't already in

Ex.1049 / Page 264 of 748
Intel Corporation et al