SP Parallel Programming Workshop

message passing interface (mpi)

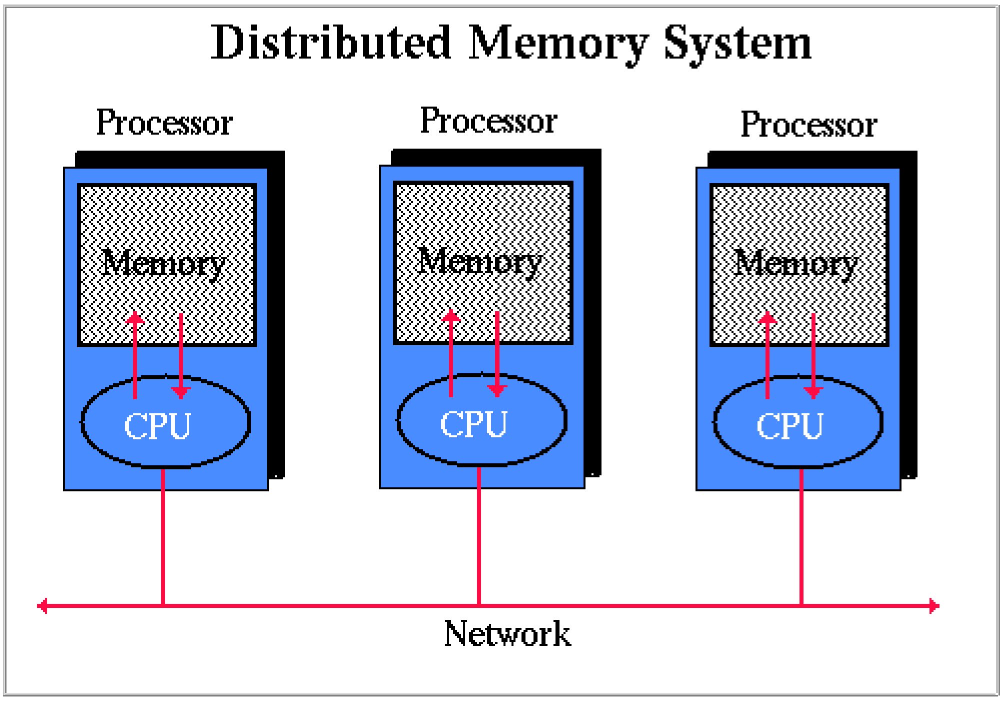
Table of Contents

The Message Passing Paradigm

Note: This section can be skipped if the reader is already familiar with message passing concepts.

Distributed Memory

Every processor has its own local memory which can be accessed directly only by its own CPU. Transfer of data from one processor to another is performed over a network. Differs from shared memory systems which permit multiple processors to directly access the same memory resource via a memory bus.



Message Passing

The method by which data from one processor’s memory is copied to the memory of another processor. In distributed memory systems, data is generally sent as packets of information over a network from one processor to another. A message may consist of one or more packets, and usually includes routing and/or other control information.

Process

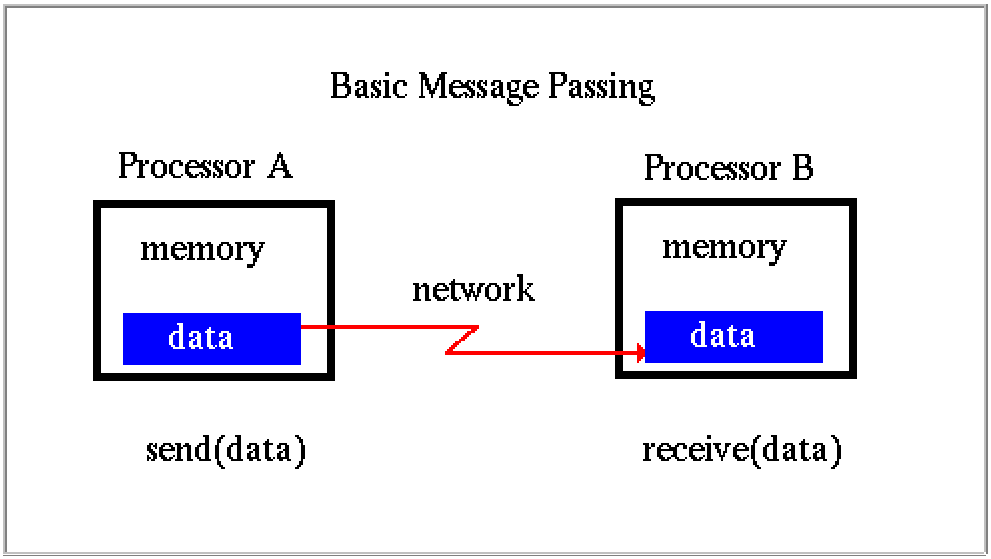
A process is a set of executable instructions (program) which runs on a processor. One or more processes may execute on a processor. In a message passing system, all processes communicate with each other by sending messages - even if they are running on the same processor. For reasons of efficiency, however, message passing systems generally associate only one process per processor.

Message Passing Library

Usually refers to a collection of routines which are imbedded in application code to accomplish send, receive and other message passing operations.

Send / Receive

Message passing involves the transfer of data from one process (send) to another process (receive). Requires the cooperation of both the sending and receiving process. Send operations usually require the sending process to specify the data's location, size, type and the destination. Receive operations should match a corresponding send operation.



Synchronous / Asynchronous

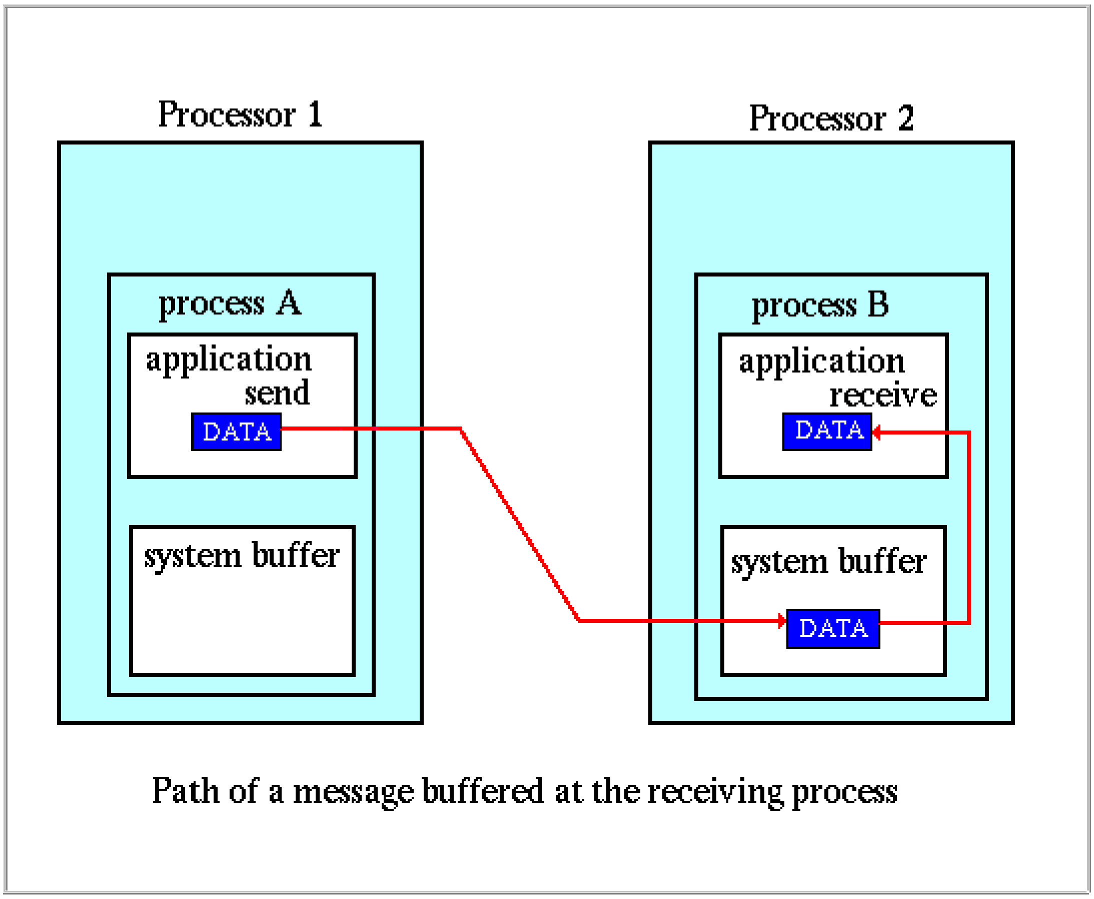
A synchronous send operation will complete only after acknowledgement that the message was safely received by the receiving process. Asynchronous send operations may "complete" even though the receiving process has not actually received the message.

Application Buffer

The address space that holds the data which is to be sent or received. For example, your program uses a variable called, "inmsg". The application buffer for inmsg is the program memory location where the value of inmsg resides.

System Buffer

System space for storing messages. Depending upon the type of send/ receive operation, data in the application buffer may be required to be copied to/from system buffer space. Allows communication to be asynchronous.



Blocking Communication

A communication routine is blocking if the completion of the call is dependent on certain "events". For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use.

Non-blocking Communication

A communication routine is non-blocking if the call returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of message).

It is not safe to modify or use the application buffer after completion of a non-blocking send. It is the programmer’s responsibility to insure that the application buffer is free for reuse.

Non-blocking communications are primarily used to overlap computation with communication to effect performance gains.

What Is MPI?

- Message Passing Interface: A specification for message passing libraries, designed to be a standard for distributed memory, message passing, parallel computing.

- The goal of the Message Passing Interface simply stated is to provide a widely used standard for writing message-passing programs. The interface attempts to establish a practical, portable, efficient, and flexible standard for message passing.

- MPI resulted from the efforts of numerous individuals and groups over the course of 2 years. History:

    - 1980s - early 1990s: Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.

    - April, 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.

    - November 1992: - Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. MPIF eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.

    - November 1993: Supercomputing 93 conference - draft MPI standard presented.
    - Final version of draft released in May, 1994 - available on the WWW at: http://www.mcs.anl.gov/Projects/mpi/standard.html

- Reasons for using MPI:
    - Standardization - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms.
    - Portability - there is no need to modify your source code when you port your application to a different platform which supports MPI.
    - Performance - vendor implementations should be able to exploit native hardware features to optimize performance.
    - Functionality (over 115 routines)
    - Availability - a variety of implementations are available, both vendor and public domain.
- Target platform is a distributed memory system including massively parallel machines, SMP clusters, workstation clusters and heterogenous networks.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing the resulting algorithm using MPI constructs.
- The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time. (MPI-2 is attempting to address this issue).
- Able to be used with C and Fortran programs. C++ and Fortran 90 language bindings are being addressed by MPI-2.

Getting Started

Header File

Required for all programs/routines which make MPI library calls.

C include file	Fortran include file
`#include "mpi.h"`	`include 'mpif.h'`

Format of MPI Calls

C names are case sensitive; Fortran names are not.

C Binding	
Format:	`rc = MPI_Xxxxx(parameter, ... )`
Example:	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Error code:	Returned as "rc". MPI_SUCCESS if successful

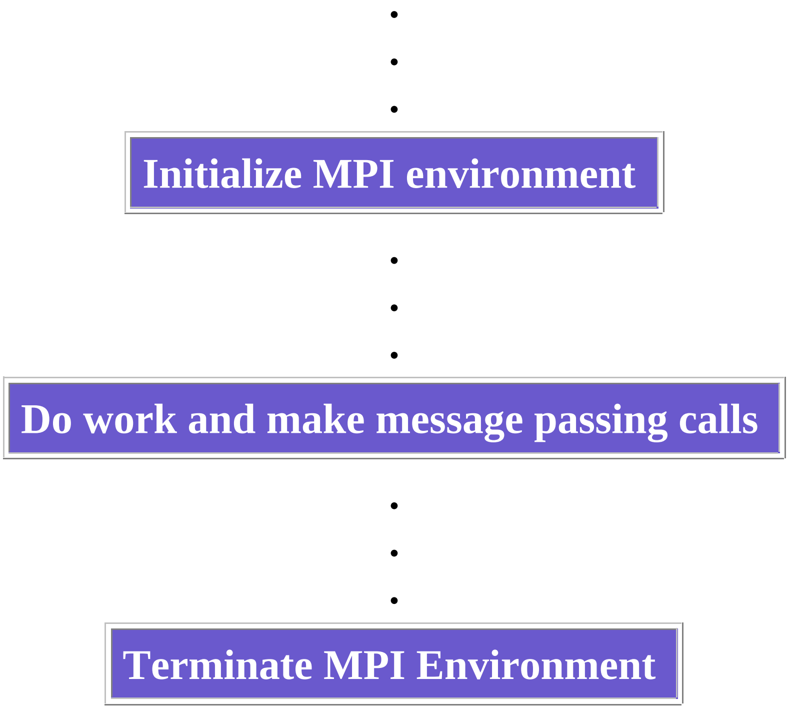
Fortran Binding	

Format:	`CALL MPI_XXXXX(parameter,..., ierr)`
	`call mpi_xxxxx(parameter,..., ierr)`

Example:	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ie`
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

General MPI Program Structure

MPI include file



Initialize MPI environment

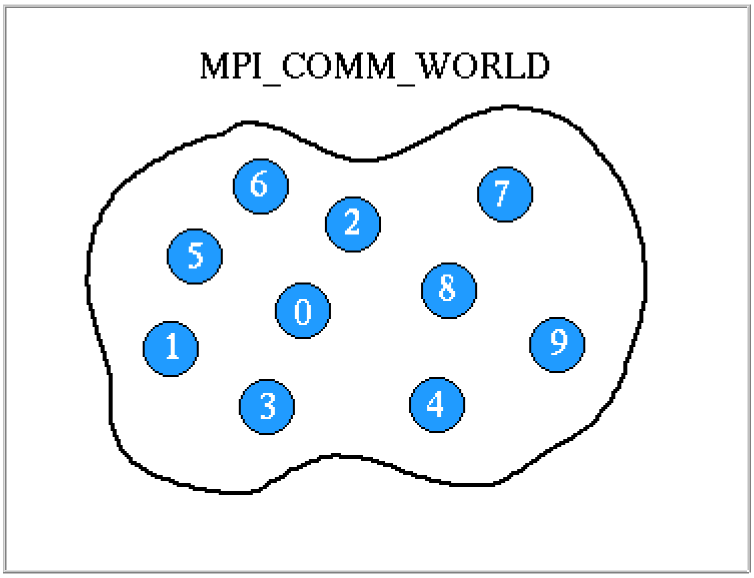
Do work and make message passing calls

Terminate MPI Environment

Communicators and Groups

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.

Communicators and groups will be covered in more detail later. For now, simply use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator which includes all of your MPI processes.



Rank

Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "process ID". Ranks are contiguous and begin at zero.

Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

Environment Management Routines

Several of the more commonly used MPI environment management routines are described below.

MPI_Init

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init (*argc,*argv)
MPI_INIT (ierr)
```

MPI_Comm_size

Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.

```
MPI_Comm_size (comm,*size)
MPI_COMM_SIZE (comm,size,ierr)
```

MPI_Comm_rank

Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank (comm,*rank)
MPI_COMM_RANK (comm,rank,ierr)
```

MPI_Abort

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

MPI_Get_processor_name

Gets the name of the processor on which the command is executed. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name (*name,*resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)
```

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized (*flag)
```

```
MPI_INITIALIZED (flag,ierr)
```

MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime ()
MPI_WTIME ()
```

MPI_Wtick

Returns the resolution in seconds (double precision) of MPI_Wtime.

```
MPI_Wtick ()
MPI_WTICK ()
```

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

Examples: Environment Management Routines

C Language

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[]; {
int    numtasks, rank, rc;

rc = MPI_Init(&argc,&argv);
if (rc != 0) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

/*******  do some work *******/

MPI_Finalize();
}
```

Fortran

```
   program simple
   include 'mpif.h'

   integer numtasks, rank, ierr, rc

   call MPI_INIT(ierr)
   if (ierr .ne. 0) then
      print *,'Error starting MPI program. Terminating.'
      call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
   end if

   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
   print *, 'Number of tasks=',numtasks,' My rank=',rank

C ****** do some work ******

   call MPI_FINALIZE(ierr)

   end
```

Point to Point Communication Routines

MPI Message Passing Routine Arguments

MPI point-to-point communication routines generally have an argument list which takes one of the following formats:

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,r`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm`

Buffer

Program (application) address space which references the data that is to be sent or or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: **`&var1`**

Data Count

Indicates the number of data elements of a particular type to be sent.

Data Type

For reasons of portability, MPI predefines its data types. Programmers may also create their own data types (derived types). Note that the MPI types MPI_BYTE and MPI_PACKED do not correspond to standard C or Fortran types.

MPI C data types		MPI Fortran data type	
MPI_CHAR	signed char	**MPI_CHARACTER**	char
MPI_SHORT	signed short int		
MPI_INT	signed int	**MPI_INTEGER**	integ
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_FLOAT	float	**MPI_REAL**	real
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**	doub prec
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	com
		MPI_LOGICAL	logic
MPI_BYTE	8 binary digits	**MPI_BYTE**	8 bin digit
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	**MPI_PACKED**	data or u with MPI MPI

Destination

An argument to send routines which indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Source

An argument to receive routines which indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

Tag

Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operations, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow much larger range than this.

Communicator

Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.

Status

For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG). In Fortran, it is an integer array of size MPI_STATUS_SIZE (ex. stat(MPI_SOURCE) stat(MPI_TAG)). Additionally, the actual number of bytes received are obtainable from Status via the MPI_Get_count routine.

Request

Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request. In Fortran, it is an integer.

Point to Point Communication Routines

Blocking Message Passing Routines

The more commonly used MPI blocking message passing routines are described below.

MPI_Send

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.

```
MPI_Send (*buf,count,datatype,dest,tag,comm)
MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Recv

Receive a message and block until the requested data is available in the application buffer in the receiving task.

```
MPI_Recv (*buf,count,datatype,source,tag,comm,*status)
MPI_RECV (buf,count,datatype,source,tag,comm,status,ierr
```

MPI_Ssend

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

```
MPI_Ssend (*buf,count,datatype,dest,tag,comm,ierr)
MPI_SSEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Bsend

Buffered blocking send: permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Insulates against the problems associated with insufficient system buffer space. Routine returns after the data has been copied from application buffer space to the allocated send buffer. Must be used with the MPI_Buffer_attach routine.

```
MPI_Bsend (*buf,count,datatype,dest,tag,comm)
MPI_BSEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Buffer_attach
MPI_Buffer_detach

Used by programmer to allocate/deallocate message buffer space to be used by the MPI_Bsend routine. The size argument is specified in actual data bytes - not a count of data elements. Only one buffer can be attached to a process at a time. Note that the IBM implementation uses MPI_BSEND_OVERHEAD bytes of the allocated buffer for overhead.

```
MPI_Buffer_attach (*buffer,size)
MPI_Buffer_detach (*buffer,size)
MPI_BUFFER_ATTACH (buffer,size,ierr)
MPI_BUFFER_DETACH (buffer,size,ierr)
```

MPI_Rsend

Blocking ready send. Should only be used if the programmer is certain that the matching receive has already been posted.

```
MPI_Rsend (*buf,count,datatype,dest,tag,comm)
MPI_RSEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Sendrecv

Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

```
MPI_Sendrecv (*sendbuf,sendcount,sendtype,dest,sendtag,
       *recvbuf,recvcount,recvtype,source,recvtag,
       comm,*status)
MPI_SENDRECV (sendbuf,sendcount,sendtype,dest,sendtag,
       recvbuf,recvcount,recvtype,source,recvtag,
       comm,status,ierr)
```

MPI_Wait
MPI_Waitany
MPI_Waitall
MPI_Waitsome

MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

```
MPI_Wait (*request,*status)
MPI_Waitany (count,*array_of_requests,*index,*status)
MPI_Waitall (count,*array_of_requests,*array_of_statuses
MPI_Waitsome (incount,*array_of_requests,*outcount,
        *array_of_offsets, *array_of_statuses)
MPI_WAIT (request,status,ierr)
MPI_WAITANY (count,array_of_requests,index,status,ierr)
MPI_WAITALL (count,array_of_requests,array_of_statuses,
        ierr)
MPI_WAITSOME (incount,array_of_requests,outcount,
        array_of_offsets, array_of_statuses,ierr)
```

MPI_Probe

Performs a blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

```
MPI_Probe (source,tag,comm,*status)
MPI_PROBE (source,tag,comm,status,ierr)
```

Examples: Blocking Message Passing

Routines

Task 0 pings task 1 and awaits return ping

C Language

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Fortran

```
  program ping
  include 'mpif.h'

  integer numtasks, rank, dest, source, tag, ierr
  integer stat(MPI_STATUS_SIZE)
  character inmsg, outmsg
  tag = 1

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

  if (rank .eq. 0) then
     dest = 1
     source = 1
     outmsg = 'x'
     call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
&            MPI_COMM_WORLD, ierr)
     call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
&            MPI_COMM_WORLD, stat, ierr)

  else if (rank .eq. 1) then
     dest = 0
     source = 0
     call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
&       MPI_COMM_WORLD, stat, err)
     call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
&       MPI_COMM_WORLD, err)
  endif

  call MPI_FINALIZE(ierr)

  end
```

Point to Point Communication Routines

Non-Blocking Message Passing Routines

The more commonly used MPI non-blocking message passing routines are described below.

MPI_Isend

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicates that the non-blocking send has completed.

```
MPI_Isend (*buf,count,datatype,dest,tag,comm,*request)
MPI_ISEND (buf,count,datatype,dest,tag,comm,request,ierr
```

MPI_Irecv

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

```
MPI_Irecv (*buf,count,datatype,source,tag,comm,*request)
MPI_IRECV (buf,count,datatype,source,tag,comm,request,ie
```

MPI_Issend

Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

```
MPI_Issend (*buf,count,datatype,dest,tag,comm,*request)
MPI_ISSEND (buf,count,datatype,dest,tag,comm,request,ier
```

MPI_Ibsend

Non-blocking buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.

```
MPI_Ibsend (*buf,count,datatype,dest,tag,comm,*request)
MPI_IBSEND (buf,count,datatype,dest,tag,comm,request,ier
```

MPI_Irsend

Non-blocking ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Should only be used if the programmer is certain that the matching receive has already been posted.

```
MPI_Irsend (*buf,count,datatype,dest,tag,comm,*request)
MPI_IRSEND (buf,count,datatype,dest,tag,comm,request,ier
```

MPI_Test
MPI_Testany
MPI_Testall
MPI_Testsome

MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

```
MPI_Test (*request,*flag,*status)
MPI_Testany (count,*array_of_requests,*index,*flag,*stat
MPI_Testall (count,*array_of_requests,*flag,*array_of_st
MPI_Testsome (incount,*array_of_requests,*outcount,
       *array_of_offsets, *array_of_statuses)
MPI_TEST (request,flag,status,ierr)
MPI_TESTANY (count,array_of_requests,index,flag,status,i
MPI_TESTALL (count,array_of_requests,flag,array_of_statu
MPI_TESTSOME (incount,array_of_requests,outcount,
       array_of_offsets, array_of_statuses,ierr)
```

MPI_Iprobe

Performs a non-blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. The integer "flag" parameter is returned logical true (1) if a message has arrived, and logical false (0) if not. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

```
MPI_Iprobe (source,tag,comm,*flag,*status)
MPI_IPROBE (source,tag,comm,flag,status,ierr)
```

Examples: Non-Blocking Message Passing Routines

Nearest neighbor exchange in ring topology

C Language

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

Fortran

```
      program ringtopo
      include 'mpif.h'

      integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr
      integer stats(MPI_STATUS_SIZE,4), reqs(4)
      tag1 = 1
      tag2 = 2

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      prev = rank - 1
      next = rank + 1
      if (rank .eq. 0) then
         prev = numtasks - 1
      endif
      if (rank .eq. numtasks - 1) then
         next = 0
      endif

      call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1,
    &     MPI_COMM_WORLD, reqs(1), ierr)
      call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2,
    &     MPI_COMM_WORLD, reqs(2), ierr)

      call MPI_ISEND(rank, 1, MPI_INTEGER, prev, tag2,
    &     MPI_COMM_WORLD, reqs(3), ierr)
      call MPI_ISEND(rank, 1, MPI_INTEGER, next, tag1,
    &     MPI_COMM_WORLD, reqs(4), ierr)

      call MPI_WAITALL(4, reqs, stats, ierr);

      call MPI_FINALIZE(ierr)

      end
```

Collective Communication Routines

- Collective communication involves all processes in the scope of the communicator. All processes are by default, members in the communicator MPI_COMM_WORLD.

- Three types of collective operations

  - Synchronization - processes wait until all members of the group have reached the synchronization point
  - Data movement - broadcast, scatter/gather, all to all
  - Collective computation (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data

- Collective operations are blocking
- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into a new groups and then attaching the new groups to new communicators (discussed later).
- Work with MPI defined datatypes - not with derived types.

Collective Communication Routines

MPI_Barrier

Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.

```
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)
```

MPI_Bcast

Broadcasts (sends) a message from the process with rank "root" to all other processes in the group. Diagram here.

```
MPI_Bcast (*buffer,count,datatype,root,comm)
MPI_BCAST (buffer,count,datatype,root,comm,ierr)
```

MPI_Scatter

Distributes distinct messages from a single source task to each task in the group. Diagram here.

```
MPI_Scatter (*sendbuf,sendcnt,sendtype,*recvbuf,
       recvcnt,recvtype,root,comm)
MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,
       recvcnt,recvtype,root,comm,ierr)
```

MPI_Gather

Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter. Diagram here.

```
MPI_Gather (*sendbuf,sendcnt,sendtype,*recvbuf,
       recvcount,recvtype,root,comm)
MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,
       recvcount,recvtype,root,comm,ierr)
```

MPI_Allgather

Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group. Diagram here.

```
MPI_Allgather (*sendbuf,sendcount,sendtype,*recvbuf,
       recvcount,recvtype,comm)
MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,
       recvcount,recvtype,comm,info)
```

MPI_Reduce

Applies a reduction operation on all tasks in the group and places the result in one task. Diagram here.

```
MPI_Reduce (*sendbuf,*recvbuf,count,datatype,op,root,com
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,
```

The predefined MPI reduction operations appear below. Users can also define their own reduction functions by using the MPI_Op_create routine.

MPI Reduction Operation		C data types	Fortran data types
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

MPI_Allreduce

Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast. Diagram here.

```
MPI_Allreduce (*sendbuf,*recvbuf,count,datatype,op,comm)
MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op,comm,ie
```

MPI_Reduce_scatter

First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation. Diagram here.

```
MPI_Reduce_scatter (*sendbuf,*recvbuf,recvcount,datatype
       op,comm)
MPI_REDUCE_SCATTER (sendbuf,recvbuf,recvcount,datatype,
       op,comm,ierr)
```

MPI_Alltoall

Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index. Diagram here.

```
MPI_Alltoall (*sendbuf,sendcount,sendtype,*recvbuf,
       recvcnt,recvtype,comm)
MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf,
       recvcnt,recvtype,comm,ierr)
```

MPI_Scan

Performs a scan operation with respect to a reduction operation across a task group. Diagram here.

```
MPI_Scan (*sendbuf,*recvbuf,count,datatype,op,comm)
MPI_SCAN (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

Examples: Collective Communications

Perform a scatter operation on the rows of an array

C Language

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
             MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran

```
   program scatter
   include 'mpif.h'

   integer SIZE
   parameter(SIZE=4)
   integer numtasks, rank, sendcount, recvcount, source, ierr
   real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)

C  Fortran stores this array in column major order, so the
C  scatter will actually scatter columns, not rows.
   data sendbuf /1.0, 2.0, 3.0, 4.0,
 &         5.0, 6.0, 7.0, 8.0,
 &         9.0, 10.0, 11.0, 12.0,
 &         13.0, 14.0, 15.0, 16.0 /

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   if (numtasks .eq. SIZE) then
      source = 1
      sendcount = SIZE
      recvcount = SIZE
      call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf,
 &   recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
      print *, 'rank= ',rank,' Results: ',recvbuf
   else
      print *, 'Must specify',SIZE,' processors.  Terminating.'
   endif

   call MPI_FINALIZE(ierr)

   end
```

Sample program output:

```
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
```

Derived Data Types

- MPI predefines its primitive data types:

MPI C data types	MPI Fortran data types
MPI_CHAR	MPI_CHARACTER
MPI_SHORT	
MPI_INT	MPI_INTEGER
MPI_LONG	
MPI_UNSIGNED_CHAR	
MPI_UNSIGNED_SHORT	
MPI_UNSIGNED	
MPI_UNSIGNED_LONG	
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_LONG_DOUBLE	
	MPI_COMPLEX
	MPI_LOGICAL
MPI_BYTE	MPI_BYTE
MPI_PACKED	MPI_PACKED

- MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.

- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.

- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Derived Data Type Routines

MPI_Type_contiguous

The simplest constructor. Produces a new data type by making count copies of an existing data type.

```
MPI_Type_contiguous (count,oldtype,*newtype)
MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)
```

MPI_Type_vector
MPI_Type_hvector

Similar to contiguous, but allows for regular gaps (stride) in the displacements. MPI_Type_hvector is identical to MPI_Type_vector except that stride is specified in bytes.

```
MPI_Type_vector (count,blocklength,stride,oldtype,*newty
MPI_TYPE_VECTOR (count,blocklength,stride,oldtype,newtyp
```

MPI_Type_indexed
MPI_Type_hindexed

An array of displacements of the input data type is provided as the map for the new data type. MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are specified in bytes.

```
MPI_Type_indexed (count,blocklens[],offsets[],old_type,*
MPI_TYPE_INDEXED (count,blocklens(),offsets(),old_type,n
```

MPI_Type_struct

The new data type is formed according to completely defined map of the component data types.

```
MPI_Type_struct (count,blocklens[],offsets[],old_types,*
MPI_TYPE_STRUCT (count,blocklens(),offsets(),old_types,n
```

MPI_Type_extent

Returns the size in bytes of the specified data type. Useful for the MPI subroutines which require specification of offsets in bytes.

```
MPI_Type_extent (datatype,*extent)
MPI_TYPE_EXTENT (datatype,extent,ierr)
```

MPI_Type_commit

Commits new datatype to the system. Required for all user constructed (derived) datatypes.

```
MPI_Type_commit (datatype)
MPI_TYPE_COMMIT (datatype,ierr)
```

Examples: Contiguous Derived Data Type

Create a data type representing a row of an array and distribute a different row to all processes. Diagram here.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
   13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
     }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
         rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran

```
   program contiguous
   include ’mpif.h’

   integer SIZE
   parameter(SIZE=4)
   integer numtasks, rank, source, dest, tag, i,  ierr
   real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
   integer stat(MPI_STATUS_SIZE), columntype

C  Fortran stores this array in column major order
   data a  /1.0, 2.0, 3.0, 4.0,
  &         5.0, 6.0, 7.0, 8.0,
  &         9.0, 10.0, 11.0, 12.0,
  &         13.0, 14.0, 15.0, 16.0 /

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   call MPI_TYPE_CONTIGUOUS(SIZE, MPI_REAL, columntype, ierr)
   call MPI_TYPE_COMMIT(columntype, ierr)

   tag = 1
   if (numtasks .eq. SIZE) then
      if (rank .eq. 0) then
         do 10 i=0, numtasks-1
         call MPI_SEND(a(0,i), 1, columntype, i, tag,
  &                       MPI_COMM_WORLD,ierr)
 10      continue
      endif

      source = 0
      call MPI_RECV(b, SIZE, MPI_REAL, source, tag,
  &                   MPI_COMM_WORLD, stat, ierr)
         print *, ’rank= ’,rank,’ b= ’,b

   else
      print *, ’Must specify’,SIZE,’ processors.  Terminating.’
   endif
   call MPI_FINALIZE(ierr)

   end
```

Sample program output:

```
rank= 0  b= 1.0 2.0 3.0 4.0
rank= 1  b= 5.0 6.0 7.0 8.0
rank= 2  b= 9.0 10.0 11.0 12.0
rank= 3  b= 13.0 14.0 15.0 16.0
```

Examples: Vector Derived Data Type

Create a data type representing a column of an array and distribute different columns to all processes. Diagram here.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
  13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
```

```
    for (i=0; i<numtasks; i++)
       MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
        }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
        rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran

```
      program vector
      include 'mpif.h'

      integer SIZE
      parameter(SIZE=4)
      integer numtasks, rank, source, dest, tag, i,  ierr
      real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
      integer stat(MPI_STATUS_SIZE), rowtype

C     Fortran stores this array in column major order
      data a  /1.0, 2.0, 3.0, 4.0,
     &         5.0, 6.0, 7.0, 8.0,
     &         9.0, 10.0, 11.0, 12.0,
     &         13.0, 14.0, 15.0, 16.0 /

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      call MPI_TYPE_VECTOR(SIZE, 1, SIZE, MPI_REAL, rowtype, ierr)
      call MPI_TYPE_COMMIT(rowtype, ierr)

      tag = 1
      if (numtasks .eq. SIZE) then
         if (rank .eq. 0) then
            do 10 i=0, numtasks-1
            call MPI_SEND(a(i,0), 1, rowtype, i, tag,
     &                    MPI_COMM_WORLD, ierr)
 10         continue
         endif

         source = 0
         call MPI_RECV(b, SIZE, MPI_REAL, source, tag,
     &                 MPI_COMM_WORLD, stat, ierr)
         print *, ’rank= ’,rank,’ b= ’,b

      else
         print *, ’Must specify’,SIZE,’ processors.  Terminating.’
      endif
```

```
   call MPI_FINALIZE(ierr)

   end
```

Sample program output:

```
rank= 0  b= 1.0 5.0 9.0 13.0
rank= 1  b= 2.0 6.0 10.0 14.0
rank= 2  b= 3.0 7.0 11.0 15.0
rank= 3  b= 4.0 8.0 12.0 16.0
```

Examples: Indexed Derived Data Type

Create a datatype by extracting variable portions of an array and distribute to all tasks. Diagram here.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;

MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype)
MPI_Type_commit(&indextype);
if (rank == 0) {
  for (i=0; i<numtasks; i++)
     MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
     rank,b[0],b[1],b[2],b[3],b[4],b[5]);

MPI_Finalize();
}
```

Fortran

```
   program indexed
   include 'mpif.h'

   integer NELEMENTS
   parameter(NELEMENTS=6)
   integer numtasks, rank, source, dest, tag, i,  ierr
   integer blocklengths(0:1), displacements(0:1)
   real*4 a(0:15), b(0:NELEMENTS-1)
   integer stat(MPI_STATUS_SIZE), indextype

   data a  /1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  &         9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   blocklengths(0) = 4
   blocklengths(1) = 2
   displacements(0) = 5
   displacements(1) = 12

   call MPI_TYPE_INDEXED(2, blocklengths, displacements, MPI_REAL,
  &                      indextype, ierr)
   call MPI_TYPE_COMMIT(indextype, ierr)

   tag = 1
   if (rank .eq. 0) then
      do 10 i=0, numtasks-1
      call MPI_SEND(a, 1, indextype, i, tag, MPI_COMM_WORLD, ierr)
 10   continue
   endif

   source = 0
   call MPI_RECV(b, NELEMENTS, MPI_REAL, source, tag, MPI_COMM_WORLD,
  &              stat, ierr)
   print *, 'rank= ',rank,' b= ',b

   call MPI_FINALIZE(ierr)
   end
```

Sample program output:

```
rank= 0  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 1  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 2  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3  b= 6.0 7.0 8.0 9.0 13.0 14.0
```

Examples: Struct Derived Data Type

Create a data type which represents a particle and distribute an array of such particles to all processes. Diagram here.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;

typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
  }          Particle;
Particle     p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];
int          blockcounts[2];

/* MPI_Aint type used to be consistent with syntax of */
/* MPI_Type_extent routine */
MPI_Aint    offsets[2], extent;

MPI_Status stat;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

/* Setup description of the 4 MPI_FLOAT fields x, y, z, velocity */
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;

/* Setup description of the 2 MPI_INT fields n, type */
/* Need to first figure offset by getting size of MPI_FLOAT */
MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;

/* Now define structured type and commit it */
MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
MPI_Type_commit(&particletype);

/* Initialize the particle array and then send it to each task */
if (rank == 0) {
  for (i=0; i<NELEM; i++) {
     particles[i].x = i * 1.0;
     particles[i].y = i * -1.0;
     particles[i].z = i * 1.0;
     particles[i].velocity = 0.25;
     particles[i].n = i;
     particles[i].type = i % 2;
     }
  for (i=0; i<numtasks; i++)
     MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);

/* Print a sample of what was received */
printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank,p[3].x,
     p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

MPI_Finalize();
}
```

Fortran

```
      program struct
      include 'mpif.h'

      integer NELEM
      parameter(NELEM=25)
      integer numtasks, rank, source, dest, tag, i,  ierr
      integer stat(MPI_STATUS_SIZE)

      type Particle
      sequence
      real*4 x, y, z, velocity
      integer n, type
      end type Particle

      type (Particle) p(NELEM), particles(NELEM)
      integer particletype, oldtypes(0:1), blockcounts(0:1),
     &        offsets(0:1), extent

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   C  Setup description of the 4 MPI_REAL fields x, y, z, velocity
      offsets(0) = 0
      oldtypes(0) = MPI_REAL
      blockcounts(0) = 4

   C  Setup description of the 2 MPI_INTEGER fields n, type
   C  Need to first figure offset by getting size of MPI_REAL
      call MPI_TYPE_EXTENT(MPI_REAL, extent, ierr)
      offsets(1) = 4 * extent
      oldtypes(1) = MPI_INTEGER
      blockcounts(1) = 2

   C  Now define structured type and commit it
      call MPI_TYPE_STRUCT(2, blockcounts, offsets, oldtypes,
     &                     particletype, ierr)
      call MPI_TYPE_COMMIT(particletype, ierr)

   C  Initialize the particle array and then send it to each task
      tag = 1
      if (rank .eq. 0) then
         do 10 i=0, NELEM-1
         particles(i) = Particle ( 1.0*i, -1.0*i, 1.0*i,
     &                  0.25, i, mod(i,2) )
    10   continue

         do 20 i=0, numtasks-1
         call MPI_SEND(particles, NELEM, particletype, i, tag,
     &                 MPI_COMM_WORLD, ierr)
    20   continue
      endif

      source = 0
      call MPI_RECV(p, NELEM, particletype, source, tag,
     &              MPI_COMM_WORLD, stat, ierr)

      print *, 'rank= ',rank,' p(3)= ',p(3)
      call MPI_FINALIZE(ierr)
      end
```

Sample program output:

```
rank= 0   3.00 -3.00 3.00 0.25 3 1
rank= 2   3.00 -3.00 3.00 0.25 3 1
rank= 1   3.00 -3.00 3.00 0.25 3 1
rank= 3   3.00 -3.00 3.00 0.25 3 1
```

Group and Communicator Management Routines

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator which comprises all tasks is MPI_COMM_WORLD.

- From the programmer’s perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

- Primary purposes of group and communicator objects:

  1. Allow you to organize tasks, based upon function, into task groups.

  2. Enable Collective Communications operations across a subset of related tasks.

  3. Provide basis for implementing user defined virtual topologies

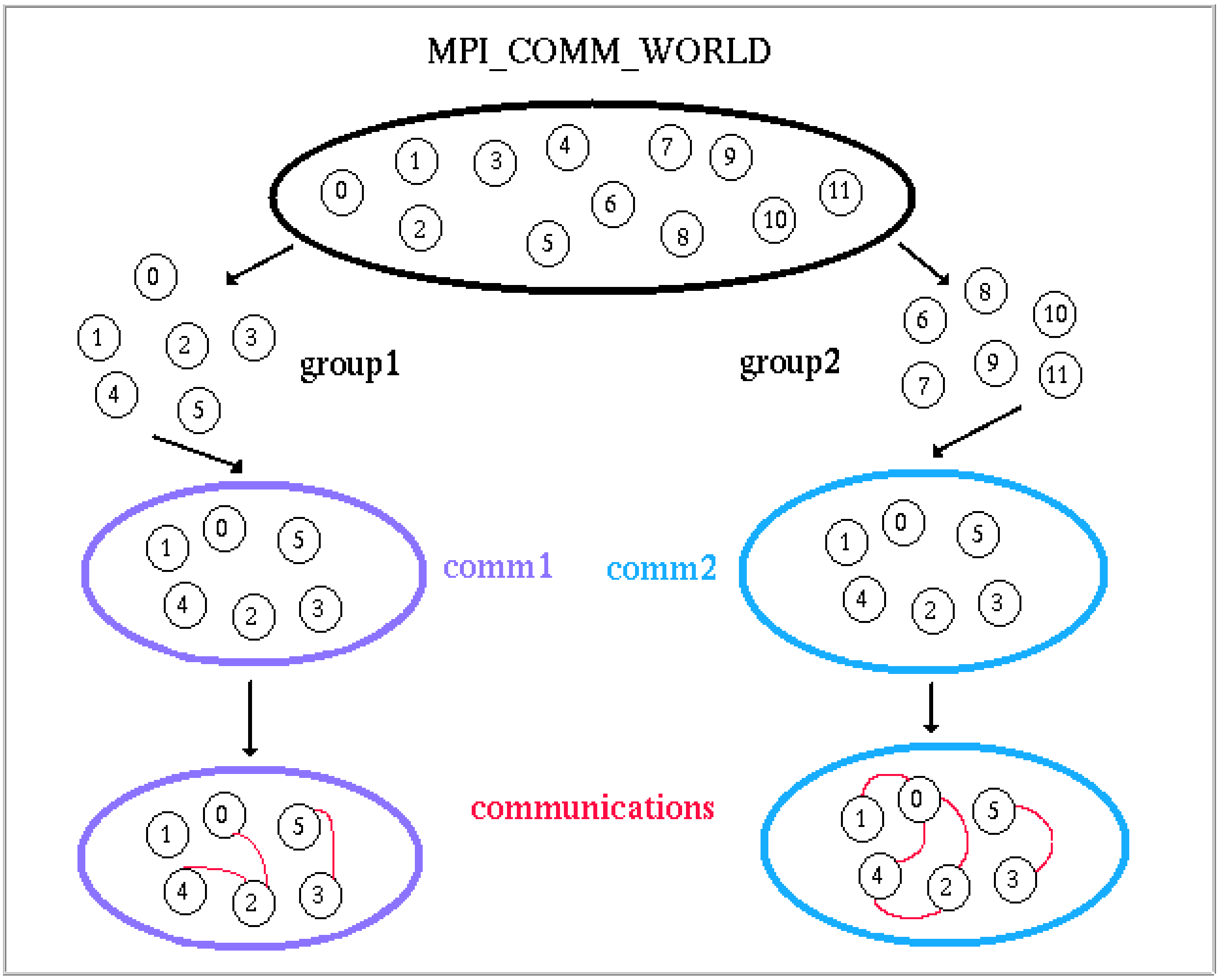
  4. Provide for safe communications

- Groups/communicators are dynamic - they can be created and destroyed during program execution.

- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.

- MPI provides over 40 routines related to groups, communicators, and virtual topologies.

- Typical usage:
  1. Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  2. Form new group as a subset of global group using MPI_Group_incl
  3. Create new communicator for new group using MPI_Comm_create
  4. Determine new rank in new communicator using MPI_Comm_rank
  5. Conduct communications using any MPI message passing routine
  6. When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free



Group and Communicator Management

Routines

MPI_Comm_group

Determines the group associated with the given communicator.

```
MPI_Comm_group (comm,*group)
MPI_COMM_GROUP (comm,group,ierr)
```

MPI_Group_rank

Returns the rank of this process in the given group or MPI_UNDEFINED if the process is not a member.

```
MPI_Group_rank (group,*rank)
MPI_GROUP_RANK (group,rank,ierr)
```

MPI_Group_size

Returns the size of a group - number of processes in the group.

```
MPI_Group_size (group,*size)
MPI_GROUP_SIZE (group,size,ierr)
```

MPI_Group_excl

Produces a group by reordering an existing group and taking only unlisted members.

```
MPI_Group_excl (group,n,*ranks,*newgroup)
MPI_GROUP_EXCL (group,n,ranks,newgroup,ierr)
```

MPI_Group_incl

Produces a group by reordering an existing group and taking only listed members.

```
MPI_Group_incl (group,n,*ranks,*newgroup)
MPI_GROUP_INCL (group,n,ranks,newgroup,ierr)
```

MPI_Group_intersection

Produces a group as the intersection of two existing groups.

```
MPI_Group_intersection (group1,group2,*newgroup)
MPI_GROUP_INTERSECTION (group1,group2,newgroup,ierr)
```

MPI_Group_union

Produces a group by combining two groups.

```
MPI_Group_union (group1,group2,*newgroup)
MPI_GROUP_UNION (group1,group2,newgroup,ierr)
```

MPI_Group_difference

Creates a group from the difference of two groups.

```
MPI_Group_difference (group1,group2,*newgroup)
MPI_GROUP_DIFFERENCE (group1,group2,newgroup,ierr)
```

MPI_Group_compare

Compares two groups and returns an integer result which is MPI_IDENT if the order and members of the two groups are the same, MPI_SIMILAR if only the members are the same, and MPI_UNEQUAL otherwise.

```
MPI_Group_compare (group1,group2,*result)
MPI_GROUP_COMPARE (group1,group2,result,ierr)
```

MPI_Group_free

Frees a group

```
MPI_Group_free (group)
MPI_GROUP_FREE (group,ierr)
```

MPI_Comm_create

Creates a new communicator from the old communicator and the new group.

```
MPI_Comm_create (comm,group,*newcomm)
MPI_COMM_CREATE (comm,group,newcomm,ierr)
```

MPI_Comm_dup

Duplicates an existing communicator with all its associated information.

```
MPI_Comm_dup (comm,*newcomm)
MPI_COMM_DUP (comm,newcomm,ierr)
```

MPI_Comm_compare

Compares two communicators and returns integer result which is MPI_IDENT if the contexts and groups are the same, MPI_CONGRUENT if different contexts but identical groups, MPI_SIMILAR if different contexts but similar groups, and MPI_UNEQUAL otherwise.

```
MPI_Comm_compare (comm1,comm2,*result)
MPI_COMM_COMPARE (comm1,comm2,result,ierr)
```

MPI_Comm_free

Marks the communicator object for deallocation.

```
MPI_Comm_free (*comm)
MPI_COMM_FREE (comm,ierr)
```

Examples: Group and Communicator Routines

Create two different process groups for separate collective communications exchange. Requires creating new communicators also.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(argc,argv)
int argc;
char *argv[];  {
int        rank, new_rank, sendbuf, recvbuf, numtasks,
           ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group  orig_group, new_group;
MPI_Comm   new_comm;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks != NPROCS) {
  printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;

/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Create new new communicator and then perform collective communicatio
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

Fortran

```
   program group
   include 'mpif.h'

   integer NPROCS
   parameter(NPROCS=8)
   integer rank, new_rank, sendbuf, recvbuf, numtasks
   integer ranks1(4), ranks2(4), ierr
   integer orig_group, new_group, new_comm
   data ranks1 /0, 1, 2, 3/, ranks2 /4, 5, 6, 7/

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

   if (numtasks .ne. NPROCS) then
     print *, 'Must specify MPROCS= ',NPROCS,' Terminating.'
     call MPI_FINALIZE(ierr)
     stop
   endif

   sendbuf = rank

C  Extract the original group handle
   call MPI_COMM_GROUP(MPI_COMM_WORLD, orig_group, ierr)

C  Divide tasks into two distinct groups based upon rank
   if (rank .lt. NPROCS/2) then
      call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks1,
 &                    new_group, ierr)
   else
      call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks2,
 &                    new_group, ierr)
   endif

   call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group,
 &                    new_comm, ierr)
   call MPI_ALLREDUCE(sendbuf, recvbuf, 1, MPI_INTEGER,
 &                    MPI_SUM, new_comm, ierr)

   call MPI_GROUP_RANK(new_group, new_rank, ierr)
   print *, 'rank= ',rank,' newrank= ',new_rank,' recvbuf= ',
 &     recvbuf

   call MPI_FINALIZE(ierr)
   end
```

Sample program output:

```
rank= 7 newrank= 3 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
```

```
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
```

Virtual Topologies

▲ ▼

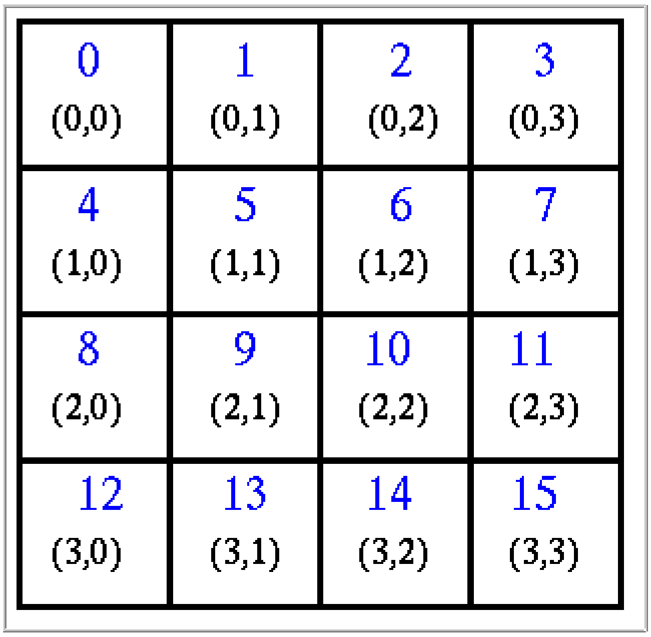
- MPI provides a means for ordering the processes of a group into topologies. MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.

- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.

- Virtual topologies may be useful for applications with specific communication patterns - patterns that match the topology structure. For example, a Cartesian topology might prove convenient for an application which requires 4-way nearest neighbor communications for grid based data.

- Virtual topologies are built upon MPI communicators and groups.

- The mapping of processes into a Cartesian topology is dependent upon the MPI implementation. A particular implementation may optimize the mapping based upon the physical characteristics of a given parallel machine. A simplified mapping of processes to coordinates appears below:



0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Virtual Topology Routines

MPI_Cart_coords

Determines process coordinates in Cartesian topology given rank in group.

```
MPI_Cart_coords (comm,rank,maxdims,*coords[])
MPI_CART_COORDS (comm,rank,maxdims,coords(),ierr)
```

MPI_Cart_create

Creates a new communicator to which Cartesian topology information has been attached.

```
MPI_Cart_create (comm_old,ndims,*dims[],*periods,
       reorder,*comm_cart)
MPI_CART_CREATE (comm_old,ndims,dims(),periods,
       reorder,comm_cart,ierr)
```

MPI_Cart_get

Retrieves the number of dimensions, coordinates and periodicity setting for the calling process in a Cartesian topology.

```
MPI_Cart_get (comm,maxdims,*dims,*periods,*coords[])
MPI_CART_GET (comm,maxdims,dims,periods,coords(),ierr)
```

MPI_Cart_map

Maps process to Cartesian topology information

```
MPI_Cart_map (comm_old,ndims,*dims[],*periods[],*newrank
MPI_CART_MAP (comm_old,ndims,dims(),periods(),newrank,ie
```

MPI_Cart_rank

Determines process rank in communicator given the Cartesian coordinate location.

```
MPI_Cart_rank (comm,*coords[],*rank)
MPI_CART_RANK (comm,coords(),rank,ierr)
```

MPI_Cart_shift

Returns the shifted source and destination ranks for the calling process in a Cartesian topology. Calling process specifies the shift direction and amount.

```
MPI_Cart_shift (comm,direction,displ,*source,*dest)
MPI_CART_SHIFT (comm,direction,displ,source,dest,ierr)
```

MPI_Cart_sub

Partitions a communicator into subgroups which form lower-dimensional Cartesian subgrids

```
MPI_Cart_sub (comm,*remain_dims[],*comm_new)
MPI_CART_SUB (comm,remain_dims(),comm_new,ierr)
```

MPI_Cartdim_get

Retrieves the number of dimensions associated with a Cartesian topology communicator.

```
MPI_Cartdim_get (comm,*ndims)
MPI_CARTDIM_GET (comm,ndims,ierr)
```

MPI_Dims_create

Creates a division of processors in a Cartesian grid.

```
MPI_Dims_create (nnodes,ndims,*dims[])
MPI_DIMS_CREATE (nnodes,ndims,dims(),ierr)
```

MPI_Graph_create

Makes a new communicator to which topology information has been attached.

```
MPI_Graph_create (comm_old,nnodes,*index[],*edges[],
       reorder,*comm_graph)
MPI_GRAPH_CREATE (comm_old,nnodes,index(),edges(),
       reorder,comm_graph,ierr)
```

MPI_Graph_get

Retrieves graph topology information associated with a communicator.

```
MPI_Graph_get (comm,maxindex,maxedges,*index[],*edges[])
MPI_GRAPH_GET (comm,maxindex,maxedges,index(),edges(),ie
```

MPI_Graph_map

Maps process to graph topology information.

```
MPI_Graph_map (comm_old,nnodes,*index[],*edges[],*newran
MPI_GRAPH_MAP (comm_old,nnodes,index(),edges(),newrank,i
```

MPI_Graph_neighbors

Returns the neighbors of a node associated with a graph topology.

```
MPI_Graph_neighbors (comm,rank,maxneighbors,*neighbors[]
MPI_GRAPH_NEIGHBORS (comm,rank,maxneighbors,neighbors(),
```

MPI_Graphdims_get

Retrieves graph topology information (number of nodes and number of edges) associated with a communicator

```
MPI_Graphdims_get (comm,*nnodes,*nedges)
MPI_GRAPHDIMS_GET (comm,nnodes,nedges,ierr)
```

MPI_Topo_test

Determines the type of topology (if any) associated with a communicator.

```
MPI_Topo_test (comm,*top_type)
MPI_TOPO_TEST (comm,top_type,ierr)
```

Examples: Cartesian Virtual Topology Example

Create a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbors.

C Language

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm)
  MPI_Comm_rank(cartcomm, &rank);
  MPI_Cart_coords(cartcomm, rank, 2, coords);
  MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
  MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
  outbuf = rank;

  for (i=0; i<4; i++) {
     dest = nbrs[i];
     source = nbrs[i];
     MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
               MPI_COMM_WORLD, &reqs[i]);
     MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
               MPI_COMM_WORLD, &reqs[i+4]);
     }

  MPI_Waitall(8, reqs, stats);

  printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
        rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
        nbrs[RIGHT]);
  printf("rank= %d                 inbuf(u,d,l,r)= %d %d %d %d\n",
        rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran

```
      program cartesian
      include 'mpif.h'

      integer SIZE, UP, DOWN, LEFT, RIGHT
      parameter(SIZE=16)
      parameter(UP=1)
      parameter(DOWN=2)
      parameter(LEFT=3)
      parameter(RIGHT=4)
      integer numtasks, rank, source, dest, outbuf, i, tag, ierr,
     &         inbuf(4), nbrs(4), dims(2), coords(2),
     &         stats(MPI_STATUS_SIZE, 8), reqs(8), cartcomm,
     &         periods(2), reorder
      data inbuf /MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,
     &      MPI_PROC_NULL/,  dims /4,4/, tag /1/,
     &      periods /0,0/, reorder /0/

      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      if (numtasks .eq. SIZE) then
         call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods, reorder,
     &                        cartcomm, ierr)
         call MPI_COMM_RANK(cartcomm, rank, ierr)
         call MPI_CART_COORDS(cartcomm, rank, 2, coords, ierr)
         print *,'rank= ',rank,'coords= ',coords
         call MPI_CART_SHIFT(cartcomm, 0, 1, nbrs(UP), nbrs(DOWN), ierr)
         call MPI_CART_SHIFT(cartcomm, 1, 1, nbrs(LEFT), nbrs(RIGHT),
         call MPI_CART_SHIFT(cartcomm, 1, 1, nbrs(LEFT), nbrs(RIGHT),
     &                       ierr)

         outbuf = rank
         do i=1,4
            dest = nbrs(i)
            source = nbrs(i)
            call MPI_ISEND(outbuf, 1, MPI_INTEGER, dest, tag,
     &                    MPI_COMM_WORLD, reqs(i), ierr)
            call MPI_IRECV(inbuf(i), 1, MPI_INTEGER, source, tag,
     &                    MPI_COMM_WORLD, reqs(i+4), ierr)
         enddo

         call MPI_WAITALL(8, reqs, stats, ierr)

         print *,’rank= ’,rank,’ coords= ’,coords,
     &           ’ neighbors(u,d,l,r)= ’,nbrs
         print *,’rank= ’,rank,’                  ’,
     &           ’ inbuf(u,d,l,r)= ’,inbuf

      else
        print *, ’Must specify’,SIZE,’ processors.  Terminating.’
      endif
      call MPI_FINALIZE(ierr)
      end
```

Sample program output: (partial)

```
rank= 0 coords= 0 0  neighbors(u,d,l,r)= -3 4 -3 1
rank= 0                inbuf(u,d,l,r)= -3 4 -3 1
rank= 1 coords= 0 1  neighbors(u,d,l,r)= -3 5 0 2
rank= 1                inbuf(u,d,l,r)= -3 5 0 2
rank= 2 coords= 0 2  neighbors(u,d,l,r)= -3 6 1 3
rank= 2                inbuf(u,d,l,r)= -3 6 1 3
        . . . . .

rank= 14 coords= 3 2  neighbors(u,d,l,r)= 10 -3 13 15
rank= 14                inbuf(u,d,l,r)= 10 -3 13 15
rank= 15 coords= 3 3  neighbors(u,d,l,r)= 11 -3 14 -3
rank= 15                inbuf(u,d,l,r)= 11 -3 14 -3
```

A Look Into The Future: MPI-2

- In March 1995, the MPI Forum began discussing enhancements to the MPI standard. Following this:

  - December 1995: Supercomputing 95 conference - Birds of a Feather meeting to discuss MPI-2 extensions.

  - November 1996: Supercomputing 96 conference - MPI-2 draft made available. Public comments solicited. Meeting to discuss MPI 2 extensions.

  These enhancements have now become part of the MPI-2 standard.

- Key areas of new functionality include:

  - Dynamic processes: extensions which remove the static process model of MPI. Provides routines to create new processes.

  - One sided communications: provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.

  - Extended collective operations: allows for non-blocking collective operations and application of collective operations to inter-communicators

  - External interfaces: defines routines which allow developers to layer on top of MPI, such as for debuggers and profilers.

  - Additional language bindings: describes C++ bindings and discusses Fortran-90 issues.

  - Parallel I/O: discusses MPI support for parallel I/O.

- A draft version of MPI-2 was made available for public comment at the Supercomputing '96 conference. The completed document is now available on the WWW at:
  WWW.ERC.MsState.Edu/mpi/mpi2.html

- More information about MPI 2 is available on the WWW at:
  WWW.ERC.MsState.Edu/mpi/mpi2.html
  www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
  www.epm.ornl.gov/~walker/mpi/mpi2.html

References and More Information

- "Using MPI", Gropp, Lusk and Skjellum. MIT Press, 1994.

- "MPI: From Fundamentals To Applications", WWW documents by David W. Walker. Mathematical Sciences Section, Oak Ridge National Laboratory.
  www.epm.ornl.gov/~walker/mpi/SLIDES/mpi-tutorial.html

- "Message Passing Interface", WWW documents from the Mississippi State University server.
  www.erc.msstate.edu/mpi

- IBM Parallel Environment Manuals
  www.rs6000.ibm.com/resource/aix_resource/sp_books/pe

- "RS/6000 SP: Practical MPI Programming", Yukiya Aoyama and Jun Nakano, RS/6000 Technical Support Center, IBM Japan. Available from IBM's Redbooks server at*www.redbooks.ibm.com*.

- *"MPI (Message-Passing Interface)" WWW documents by Shennon Shen and John Zollweg from the Cornell Theory Center server:*
  www.tc.cornell.edu/Edu/Talks/MPI/toc.html

- *"A User's Guide to MPI", Peter S. Pacheco. Department of Mathematics, University of San Francisco. PostScript draft available on the WWW at*
  ftp://math.usfca.edu/pub/MPI/mpi.guide.ps

- *Argonne National Laboratory*
  www.mcs.anl.gov/mpi

- *Cornell Theory Center*
  www.tc.cornell.edu/Edu

- *Maui High Performance Computing Center*
  www.mhpcc.edu/doc/mpi/mpi.html

- *Mississippi State University*
  WWW.ERC.MsState.Edu/mpi

- *Oak Ridge National Laboratory*
  www.epm.ornl.gov/~walker/mpi

- *Ohio Supercomputing Center*
  www.osc.edu/lam.html#MPI

Appendix A: MPI Routine Index

The following table provides a comprehensive list of all MPI routines, each linked to its corresponding man page. Man pages were derived from the MPICH implementation of MPI and may differ from man pages of other implementations.

Environment Management Routines		
MPI_Abort	MPI_Errhandler_create	MPI_Errhand
MPI_Errhandler_get	MPI_Errhandler_set	MPI_Error_cl
MPI_Error_string	MPI_Finalize	MPI_Get_pro
MPI_Init	MPI_Initialized	MPI_Wtick
MPI_Wtime		
Point-to-Point Communication Routines		
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_a
MPI_Buffer_detach	MPI_Cancel	MPI_Get_cou
MPI_Get_elements	MPI_Ibsend	MPI_Iprobe
MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv
MPI_Recv_init	MPI_Request_free	MPI_Rsend
MPI_Rsend_init	MPI_Send	MPI_Send_in
MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend

MPI_Ssend_init	MPI_Start	MPI_Startall
MPI_Test	MPI_Test_cancelled	MPI_Testall
MPI_Testany	MPI_Testsome	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsom
Collective Communication Routines		
MPI_Allgather	MPI_Allgatherv	MPI_Allreduc
MPI_Alltoall	MPI_Alltoallv	MPI_Barrier
MPI_Bcast	MPI_Gather	MPI_Gatherv
MPI_Op_create	MPI_Op_free	MPI_Reduce
MPI_Reduce_scatter	MPI_Scan	MPI_Scatter
MPI_Scatterv		
Process Group Routines		
MPI_Group_compare	MPI_Group_difference	MPI_Group_e
MPI_Group_free	MPI_Group_incl	MPI_Group_i
MPI_Group_range_excl	MPI_Group_range_incl	MPI_Group_r
MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_u
Communicators Routines		
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_
MPI_Comm_free	MPI_Comm_group	MPI_Comm_
MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_
MPI_Comm_split	MPI_Comm_test_inter	MPI_Intercom
MPI_Intercomm_merge		
Derived Types Routines		
MPI_Type_commit	MPI_Type_contiguous	MPI_Type_co
MPI_Type_extent	MPI_Type_free	MPI_Type_hi
MPI_Type_hvector	MPI_Type_indexed	MPI_Type_lb
MPI_Type_size	MPI_Type_struct	MPI_Type_ub
MPI_Type_vector		
Virtual Topology Routines		
MPI_Cart_coords	MPI_Cart_create	MPI_Cart_get
MPI_Cart_map	MPI_Cart_rank	MPI_Cart_shi

MPI_Cart_sub	MPI_Cartdim_get	MPI_Dims_cr
MPI_Graph_create	MPI_Graph_get	MPI_Graph_m
MPI_Graph_neighbors	MPI_Graph_neighbors_count	MPI_Graphdi
MPI_Topo_test		
Miscellaneous Routines		
MPI_Address	MPI_Attr_delete	MPI_Attr_get
MPI_Attr_put	MPI_DUP_FN	MPI_Keyval_
MPI_Keyval_free	MPI_NULL_COPY_FN	MPI_NULL_D
MPI_Pack	MPI_Pack_size	MPI_Pcontrol
MPI_Unpack		

http://www.mhpcc.edu/training/workshop/mpi/MAIN.html
Last Modified: Tuesday, September 28, 1999 10:18:00 blaise@mhpcc.edu