Abstract

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

The goal of this tutorial is to teach those unfamiliar with MPI how to develop and run parallel programs according to the MPI standard. The primary topics that are presented focus on those which are the most useful for new MPI programmers. The tutorial begins with an introduction, background, and basic information for getting started with MPI. This is followed by a detailed look at the MPI routines that are most useful for new MPI programmers, including MPI Environment Management, Point-to-Point Communications, and Collective Communications routines. Numerous examples in both C and Fortran are provided, as well as a lab exercise.

The tutorial materials also include more advanced topics such as Derived Data Types, Group and Communicator Management Routines, and Virtual Topologies. However, these are not actually presented during the lecture, but are meant to serve as "further reading" for those who are interested.

Level/Prerequisites: Ideal for those who are new to parallel programming with MPI. A basic understanding of parallel programming in C or Fortran is assumed. For those who are unfamiliar with Parallel Programming in general, the material covered in EC3500: Introduction To Parallel Computing would be helpful.
An Interface Specification:

- **MPI** = Message Passing Interface

- MPI is a *specification* for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

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

- Interface specifications have been defined for C/C++ and Fortran programs.

History and Evolution:

- MPI resulted from the efforts of numerous individuals and groups over the course of a 2 year period between 1992 and 1994. Some 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.


  - MPI-2 picked up where the first MPI specification left off, and addressed topics which go beyond the first MPI specification. The original MPI then became known as MPI-1. [MPI-2 is briefly covered later.](#)

  - Today, MPI implementations are a combination of MPI-1 and MPI-2. A few implementations include the full functionality of both.
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. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. For more information about MPI performance see the [MPI Performance Topics](https://computing.llnl.gov/tutorials/mpi/) tutorial.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.

Programming Model:

- MPI lends itself to virtually any distributed memory parallel programming model. In addition, MPI is commonly used to implement (behind the scenes) some shared memory models, such as Data Parallel, on distributed memory architectures.

- Hardware platforms:
  - Distributed Memory: Originally, MPI was targeted for distributed memory systems.
  - Shared Memory: As shared memory systems became more popular, particularly SMP / NUMA architectures, MPI implementations for these platforms appeared.
  - Hybrid: MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

- The number of tasks dedicated to run a parallel program is static. New tasks cannot be dynamically spawned during run time. (MPI-2 addresses this issue).

Getting Started

Header File:

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

<table>
<thead>
<tr>
<th>C include file</th>
<th>Fortran include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &quot;mpi.h&quot;</td>
<td>include 'mpif.h'</td>
</tr>
</tbody>
</table>

Format of MPI Calls:

C Binding

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = MPI_Xxxxx</td>
<td>rc = MPI_Bsend(&amp;buf,count,type,dest,tag,comm)</td>
</tr>
</tbody>
</table>

| Error code | Returned as "rc". MPI_SUCCESS if successful |

Fortran Binding

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MPI_Xxxxx parameter,..., ierr</td>
<td>CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)</td>
</tr>
</tbody>
</table>

https://computing.llnl.gov/tutorials/mpi/
Error code: Returned as "ierr" parameter. MPI_SUCCESS if successful

- C names are case sensitive; Fortran names are not.

General MPI Program Structure:

```plaintext
MPI include file

Declarations, prototypes, etc.

Program Begins
  .
  Serial code
  .

Initialize MPI environment

Parallel code begins

. . .

Do work and make message passing calls

. . .

Terminate MPI Environment

Parallel code ends

. . .

Serial code

Program Ends
```

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 that 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 "task 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

MPI environment management routines are used for an assortment of purposes, such as initializing and terminating the MPI environment, querying the environment and identity, etc. Most of the commonly used ones 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**

Returns the processor name. 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
Message Passing Interface (MPI)

solves this problem.

```c
MPI_Initialized (&flag)
MPI_INITIALIZED (flag, ierr)
```

**MPI_Wtime**

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

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

**MPI_Wtick**

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

```c
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.

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

### Examples: Environment Management Routines

#### C Language - Environment Management Routines Example

```c
#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 != MPI_SUCCESS) {
    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 *******
}
```

#### Fortran - Environment Management Routines Example

```fortran
program simple
include 'mpif.h'

integer numtasks, rank, ierr, rc

call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) 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
/
******* do some work *******
```
Point to Point Communication Routines

General Concepts

Types of Point-to-Point Operations:
- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- There are different types of send and receive routines used for different purposes. For example:
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

Buffering:
- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit. For example:
System buffer space is:
-Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous.

User managed address space (i.e. your program variables) is called the **application buffer**. MPI also provides for a user managed send buffer.

### Blocking vs. Non-blocking:

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

- **Blocking:**
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

- **Non-blocking:**
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

### Order and Fairness:

- **Order:**
  - MPI guarantees that messages will not overtake each other.
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
  - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
  - Order rules do not apply if there are multiple threads participating in the communication operations.

- **Fairness:**
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.
Point to Point Communication Routines

MPI Message Passing Routine Arguments

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Argument List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking sends</td>
<td>MPI_Send(buffer, count, type, dest, tag, comm)</td>
</tr>
<tr>
<td>Non-blocking sends</td>
<td>MPI_Isend(buffer, count, type, dest, tag, comm, request)</td>
</tr>
<tr>
<td>Blocking receive</td>
<td>MPI_Recv(buffer, count, type, source, tag, comm, request)</td>
</tr>
<tr>
<td>Non-blocking receive</td>
<td>MPI_Irecv(buffer, count, type, source, tag, comm, request)</td>
</tr>
</tbody>
</table>

Buffer

Program (application) address space that references the data that is to be sent 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 elementary data types. The table below lists those required by the standard.

<table>
<thead>
<tr>
<th>C Data Types</th>
<th>Fortran Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>real</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_DOUBLEPREC</td>
<td>double precision</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>complex</td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>double complex</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed or unpacked with MPI_Pack()/MPI_Unpack</td>
</tr>
</tbody>
</table>

Notes:

- Programmers may also create their own data types (see [Derived Data Types](https://computing.llnl.gov/tutorials/mpi/)).
- MPI_BYTE and MPI_PACKED do not correspond to standard C or Fortran types.
- The MPI standard includes the following optional data types:
  - C: MPI_LONG_LONG_INT
  - Fortran: MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_REAL2, MPI_REAL4, MPI_REAL8
- Some implementations may include additional elementary data types (MPI_LOGICAL2, MPI_COMPLEX32, etc.). Check the MPI header file.

Destination

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

Source

An argument to receive routines that 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 operation, 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 a 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_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_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)
```

**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_Buffer_attach**

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_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_Sendrecv**

https://computing.llnl.gov/tutorials/mpi/
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.

```c
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.

```c
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_ofOffsets, &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).

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

int main(argc, argv)
    int argc;
    char *argv[];
{
    int numtasks, rank, dest, source, rc, count, 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);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();
}

Fortran - Blocking Message Passing Routines Example

program ping
include 'mpif.h'
integer numtasks, rank, dest, source, count, tag, ierr
integer stat(MPI_STATUS_SIZE)
character inmsg, outmsg
outmsg = 'x'
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
  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, ierr)
  call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag, &MPI_COMM_WORLD, ierr)
endif
call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
print *, 'Task ',rank,': Received', count, 'char(s) from task', stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)
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 indicate 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)
```
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.

```c
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
MPI_Irecv (buf,count,datatype,source,tag,comm,request,ierr)
```

**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.

```c
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Issend (buf,count,datatype,dest,tag,comm,request,ierr)
```

**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.

```c
MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Ibsend (buf,count,datatype,dest,tag,comm,request,ierr)
```

**MPI_Isrsend**

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.

```c
MPI_Isrsend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Isrsend (buf,count,datatype,dest,tag,comm,request,ierr)
```

**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.

```c
MPI_Test (&request,&flag,&status)
MPI_Testany (count,&array_of_requests,&index,&flag,&status)
MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
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,ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
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).

```c
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

https://computing.llnl.gov/tutorials/mpi/
### C Language - Non-Blocking Message Passing Routines Example

```c
#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]);
    {  do some work  }
    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}
```

### Fortran - Non-Blocking Message Passing Routines Example

```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)
    do some work
    call MPI_WAITALL(4, reqs, stats, ierr);
    call MPI_FINALIZE(ierr)
end
```
Collective Communication Routines

- **All or None:**
  - Collective communication must involve all processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD.
  - It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations.

- **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.

- **Programming Considerations and Restrictions:**
  - 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 new groups and then attaching the new groups to new communicators (discussed in the Group and Communicator Management Routines section).
  - Can only be used with MPI predefined datatypes - not with MPI Derived Data 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.

```
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.

```
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.

https://computing.llnl.gov/tutorials/mpi/
### Message Passing Interface (MPI)

#### MPI_Gather

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

```
MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, 
              recvcount, recvtype, root, comm)
```

#### MPI_Allgather

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

```
MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, 
                recvcount, recvtype, comm)
```

#### MPI_Reduce

```
MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)
```

#### 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.

```
MPI_Reduce_scatter (&sendbuf, &recvbuf, count, datatype, op, comm)
```
Message Passing Interface (MPI)

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.

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.

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 - Collective Communications Example**

```c
#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
",rank,recvbuf[0], recvbuf[1],recvbuf[2],recvbuf[3]);
} else
    printf("Must specify %d processors. Terminating.
",SIZE);
MPI_Finalize();
}
```

**Fortran - Collective Communications Example**

```fortran
program scatter
include 'mpif.h'
p
```
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

- As previously mentioned, MPI predefines its primitive data types:

<table>
<thead>
<tr>
<th>C Data Types</th>
<th>Fortran Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>MPI_COMPLEX</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>MPI_DOUBLE_COMPLEX</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>MPI_PACKED</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

- 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.

```c
MPI_Type_contiguous (count, oldtype, &newtype)
MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierr)
```

**MPI_Type_vector**

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.

```c
MPI_Type_vector (count, blocklength, stride, oldtype, &newtype)
MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype, ierr)
```

**MPI_Type_indexed**

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

```c
MPI_Type_indexed (count, blocklens[], offsets[], old_type, &newtype)
MPI_TYPE_INDEXED (count, blocklens(), offsets(), old_type, newtype, ierr)
```

**MPI_Type_struct**

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

```c
MPI_Type_struct (count, blocklens[], offsets[], old_types, &newtype)
MPI_TYPE_STRUCT (count, blocklens(), offsets(), old_types, newtype, ierr)
```

**MPI_Type_extent**

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

```c
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.

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

**MPI_Type_free**

Deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.

```c
MPI_Type_free (&datatype)
MPI_TYPE_FREE (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.

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

https://computing.llnl.gov/tutorials/mpi/
Message Passing Interface (MPI)

```c
#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_Type_free(&rowtype);
    MPI_Finalize();
}
```

Fortran - Contiguous Derived Data Type Example

```fortran
program contiguous
    include 'mpif.h'

    integer SIZE
    parameter(SIZE=4)
    integer numtasks, rank, source, dest, tag, 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)
    endif
```

https://computing.llnl.gov/tutorials/mpi/
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.

[C Language - Vector Derived Data Type Example](#)

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

int main(argc,argv)

```
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.

```
#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;
```
Fortran - Indexed Derived Data Type Example

```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_TYPE_FREE(indextype, ierr)
  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
```
Examples: Struct Derived Data Type

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

C Language - Struct Derived Data Type Example

```c
#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);
```
Fortran - Struct Derived Data Type Example

```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
  endif

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

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

  print *, 'rank= ',rank, p(3)= ',p(3)
  call MPI_TYPE_FREE(particletype, ierr)
  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

Groups vs. Communicators:

- 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 that 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

Programming Considerations and Restrictions:

- 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

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

C Language - Group and Communicator Routines Example

```c
#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 communications */
```

https://computing.llnl.gov/tutorials/mpi/
Fortran - Group and Communicator Routines Example

```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
What Are They?

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.

Why Use Them?

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Example:

A simplified mapping of processes into a Cartesian virtual topology appears below:

<table>
<thead>
<tr>
<th></th>
<th>0 (0,0)</th>
<th>1 (0,1)</th>
<th>2 (0,2)</th>
<th>3 (0,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(1,0)</td>
<td>5 (1,1)</td>
<td>6 (1,2)</td>
<td>7 (1,3)</td>
</tr>
<tr>
<td>8</td>
<td>(2,0)</td>
<td>9 (2,1)</td>
<td>10 (2,2)</td>
<td>11 (2,3)</td>
</tr>
<tr>
<td>12</td>
<td>(3,0)</td>
<td>13 (3,1)</td>
<td>14 (3,2)</td>
<td>15 (3,3)</td>
</tr>
</tbody>
</table>

Virtual Topology Routines

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

C Language - Cartesian Virtual Topology Example

```c
#include "mpi.h"
#include <stdio.h>
define SIZE 16
#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3
```
```c
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
", 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
", rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
    } else
        printf("Must specify %d processors. Terminating.\n",SIZE);
    MPI_Finalize();
}
```

### Fortran - Cartesian Virtual Topology Example

```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]),
```
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 Brief Word on MPI-2

History:

- Intentionally, the MPI specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2.
- In March 1995, following the release of the initial MPI specification, the MPI Forum began discussing enhancements to the MPI standard. Following this:
  - December 1995: Supercomputing '95 conference - Birds of a Feather meeting to discuss proposed extensions to MPI.
  - The draft presented at Supercomputing '96 shortly thereafter became the MPI-2 standard.
- Not all MPI libraries provide a full implementation of MPI-2.

Key Areas of New Functionality:

- Dynamic Processes - extensions that 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 that 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** - describes MPI support for parallel I/O.

### More Information on MPI-2:
- The Argonne National Lab MPI web pages have MPI-2 information. See the **References** section for links.

### LLNL Specific Information and Recommendations

Although the MPI programming interface has been standardized, implementations will differ, as will the way MPI programs are compiled and run on different platforms. A summary of LC's MPI environment is provided here, however users will definitely want to consult the tutorials mentioned below for all of the details.

#### IBM AIX Clusters:
- IBM's MPI library is the only supported library on these platforms
- Full MPI-2 except for Dynamic Processes
- Thread-safe
- C, C++, Fortran77/90/95 are supported
- Compiling and running MPI programs, see: [IBM POWER Systems Overview](https://asc.llnl.gov/computing_resources/bluegenel/basics/)

#### Opteron Linux Clusters:
- The MVAPICH MPI library is the only supported library on these platforms. Open MPI and generic MPICH may also be available, if really needed.
- This is an MPI-1 implementation, not MPI-2, but does include MPI-I/O support
- Not thread-safe
- C, C++, Fortran77/90/95 are supported
- Compiling and running MPI programs, see: [Linux Clusters Overview](https://asc.llnl.gov/computing_resources/bluegenel/basics/)

#### IBM BG/L Clusters:
- The IBM BG/L MPI library is the only supported library on these platforms.
- This is an IBM implementation based on MPICH2. Includes MPI-2 functionality minus Dynamic Processes.
- Thread-safe
- C, C++, Fortran77/90/95 are supported
- Compiling and running MPI programs, see: [asc.llnl.gov/computing_resources/bluegenel/basics/](https://asc.llnl.gov/computing_resources/bluegenel/basics/)

---

This completes the tutorial.
Please complete the online evaluation form - unless you are doing the exercise, in which case please complete it at the end of the exercise.

Where would you like to go now?
- Exercise
- Agenda
- Back to the top

References and More Information

- Author: Blaise Barney, Livermore Computing.
- MPI web pages at Argonne National Laboratory
  http://www-unix.mcs.anl.gov/mpi
- Livermore Computing specific information:
  - MPI at LLNL
    computing.llnl.gov/mpi
  - IBM POWER Systems Overview tutorial
    computing.llnl.gov/tutorials/ibm_sp
  - Linux Clusters Overview tutorial
    computing.llnl.gov/tutorials/linux_clusters
- IBM Parallel Environment Manuals
- IBM Compiler Documentation:
  Fortran: www-4.ibm.com/software/ad/fortran
  C/C++: www-4.ibm.com/software/ad/caix

Appendix A: MPI-1 Routine Index

These man pages were derived from an IBM implementation of MPI and may differ from the man pages of other implementations.

<table>
<thead>
<tr>
<th>Environment Management Routines</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Abort</td>
<td>MPI_Errhandler_create</td>
<td>MPI_Errhandler_free</td>
</tr>
<tr>
<td>MPI_Errhandler_set</td>
<td>MPI_Errhandler_set</td>
<td>MPI_Error_class</td>
</tr>
<tr>
<td>MPI_Error_string</td>
<td>MPI_Finalize</td>
<td>MPI_Get_processor_name</td>
</tr>
<tr>
<td>MPI_Init</td>
<td>MPI_INITIALIZED</td>
<td>MPI_Wtick</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Point-to-Point Communication Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bsend</td>
<td></td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td>MPI_Buffer_detach</td>
<td></td>
<td>MPI_Buffer_detach</td>
</tr>
<tr>
<td>MPI_Get_elements</td>
<td></td>
<td>MPI_Get_elements</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td></td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td></td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>MPI_Irecv_init</td>
<td></td>
<td>MPI_Irecv_init</td>
</tr>
<tr>
<td>MPI_Isend_init</td>
<td></td>
<td>MPI_Isend_init</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td></td>
<td>MPI_Sendrecv</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td></td>
<td>MPI_Ssend</td>
</tr>
<tr>
<td>MPI_Ssend_init</td>
<td></td>
<td>MPI_Ssend_init</td>
</tr>
<tr>
<td>MPI_Test</td>
<td></td>
<td>MPI_Test</td>
</tr>
<tr>
<td>MPI_Testany</td>
<td></td>
<td>MPI_Testany</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td></td>
<td>MPI_Waitall</td>
</tr>
</tbody>
</table>

### Collective Communication Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgather</td>
<td></td>
<td>MPI_Allgather</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td></td>
<td>MPI_Alltoall</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td></td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td></td>
<td>MPI_Barrier</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td></td>
<td>MPI_Reduce</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td></td>
<td>MPI_Scatter</td>
</tr>
</tbody>
</table>

### Process Group Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Group_compare</td>
<td></td>
<td>MPI_Group_compare</td>
</tr>
<tr>
<td>MPI_Group_free</td>
<td></td>
<td>MPI_Group_free</td>
</tr>
<tr>
<td>MPI_Group_range_excl</td>
<td></td>
<td>MPI_Group_range_excl</td>
</tr>
<tr>
<td>MPI_Group_size</td>
<td></td>
<td>MPI_Group_size</td>
</tr>
</tbody>
</table>

### Communicators Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_compare</td>
<td></td>
<td>MPI_Comm_compare</td>
</tr>
<tr>
<td>MPI_Comm_free</td>
<td></td>
<td>MPI_Comm_free</td>
</tr>
<tr>
<td>MPI_Comm_remote_group</td>
<td></td>
<td>MPI_Comm_remote_group</td>
</tr>
<tr>
<td>MPI_Comm_split</td>
<td></td>
<td>MPI_Comm_split</td>
</tr>
</tbody>
</table>

### Derived Types Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_commit</td>
<td></td>
<td>MPI_Type_commit</td>
</tr>
<tr>
<td>MPI_Type_extent</td>
<td></td>
<td>MPI_Type_extent</td>
</tr>
<tr>
<td>MPI_Type_hvector</td>
<td></td>
<td>MPI_Type_hvector</td>
</tr>
<tr>
<td>MPI_Type_size</td>
<td></td>
<td>MPI_Type_size</td>
</tr>
<tr>
<td>MPI_Type_vector</td>
<td></td>
<td>MPI_Type_vector</td>
</tr>
</tbody>
</table>

### Virtual Topology Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Cart_coords</td>
<td></td>
<td>MPI_Cart_coords</td>
</tr>
<tr>
<td>MPI_Cart_map</td>
<td></td>
<td>MPI_Cart_map</td>
</tr>
<tr>
<td>MPI_Cart_sub</td>
<td></td>
<td>MPI_Cart_sub</td>
</tr>
<tr>
<td>MPI_Graph_create</td>
<td></td>
<td>MPI_Graph_create</td>
</tr>
<tr>
<td>MPI_Graph_neighbors</td>
<td></td>
<td>MPI_Graph_neighbors</td>
</tr>
<tr>
<td>MPI_Topo_test</td>
<td></td>
<td>MPI_Topo_test</td>
</tr>
</tbody>
</table>

### Miscellaneous Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Brief</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Address</td>
<td></td>
<td>MPI_Address</td>
</tr>
<tr>
<td>MPI_Attr_delete</td>
<td></td>
<td>MPI_Attr_delete</td>
</tr>
<tr>
<td>MPI_Attr_get</td>
<td></td>
<td>MPI_Attr_get</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>MPI_Attr_put</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_DUP_FN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Keyval_create</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Keyval_free</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_NULL_COPY_FN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_NULL_DELETE_FN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Pack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Pack_size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Pcontrol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Unpack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://computing.llnl.gov/tutorials/mpi/
Last Modified: Mon, 01 Mar 2010 21:40:15 GMT  blaiseb@llnl.gov
UCRL-MI-133316