Chapter 4:

MPI

Dr. Khem Poudel

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: This tutorial is ideal for those who are new to parallel programming with MPI. A basic understanding of parallel programming in C or Fortran is required. For those who are unfamiliar with Parallel Programming in general, the material covered in EC3500: Introduction To Parallel Computing would be helpful.

What is MPI?

An Interface Specification

M P I = **M**essage **P**assing **I**nterface

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.

MPI primarily *addresses the message-passing parallel programming model*: data is moved from the address space of one process to that of another process through cooperative operations on each process.

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

The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x

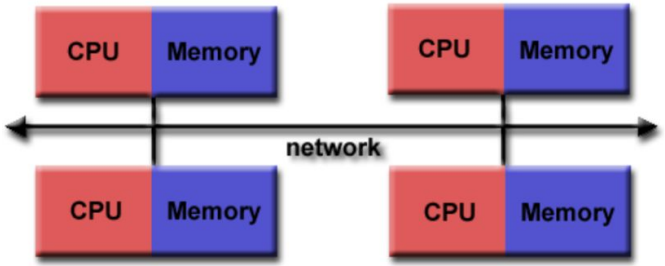
Interface specifications have been defined for C and Fortran90 language bindings:

- C++ bindings from MPI-1 are removed in MPI-3
- MPI-3 also provides support for Fortran 2003 and 2008 features

Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

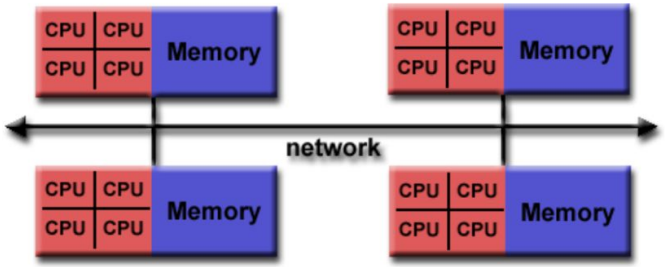
Programming Model

Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.

MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



Today, MPI runs on virtually any hardware platform:

- Distributed Memory
- Shared Memory
- Hybrid

The programming model *clearly remains a distributed memory model* however, regardless of the underlying physical architecture of the machine.

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

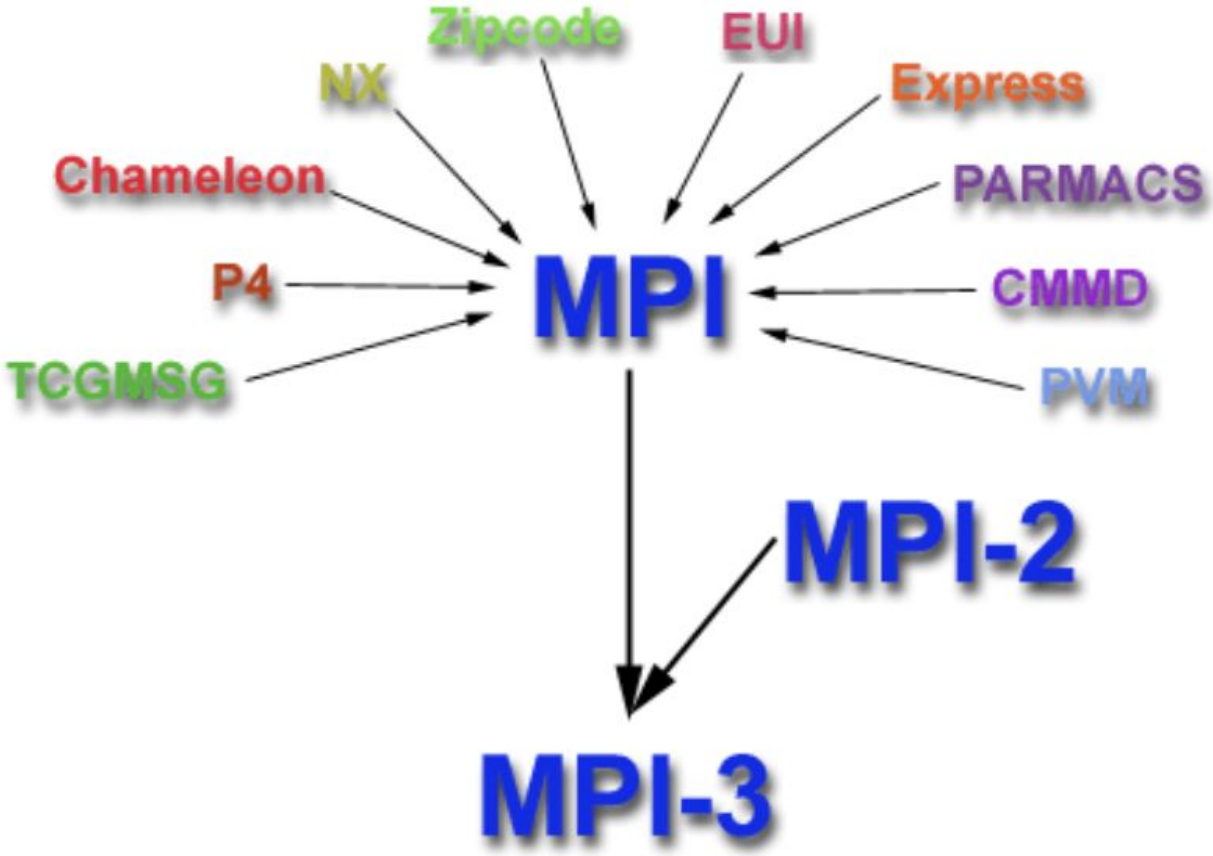
Reasons for Using MPI

- **Standardization** - MPI is the only message passing library that 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 little or 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. Any implementation is free to develop optimized algorithms.
- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
    - NOTE: Most MPI programs can be written using a dozen or less routines
- **Availability** - A variety of implementations are available, both vendor and public domain.

History and Evolution: (for those interested)

MPI has resulted from the efforts of numerous individuals and groups that began in 1992. 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.
- **Apr 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.
- **Nov 1992**: Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. It eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.
- **Nov 1993**: Supercomputing 93 conference - draft MPI standard presented.
- **May 1994**: Final version of MPI-1.0 released
    - MPI-1.1 (Jun 1995)
    - MPI-1.2 (Jul 1997)
    - MPI-1.3 (May 2008).
- **1998**: MPI-2 picked up where the first MPI specification left off, and addressed topics which went far beyond the MPI-1 specification.
    - MPI-2.1 (Sep 2008)
    - MPI-2.2 (Sep 2009)
- **Sep 2012**: The MPI-3.0 standard was approved.
    - MPI-3.1 (Jun 2015)
- **Current**: The MPI-4.0 standard is under development.


Zipcode
EUI
NX
Express
Chameleon
PARMACS
P4
MPI
CMMD
TCGMSG
PVM
MPI-2
MPI-3

LLNL MPI Implementations and Compilers

Multiple Implementations

Although the MPI programming interface has been standardized, actual library implementations will differ.

For example, just a few considerations of many:

- Which version of the MPI standard is supported?
- Are all of the features in a particular MPI version supported?
- Have any new features been added?
- What network interfaces are supported?
- How are MPI applications compiled?
- How are MPI jobs launched?
- Runtime environment variable controls?

MPI library implementations on LC systems vary, as do the compilers they are built for. These are summarized in the table below:

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

Each MPI library is briefly discussed in the following sections, including links to additional detailed information.

Selecting Your MPI Library and Compiler

LC provides a default MPI library for each cluster.

LC also provides default compilers for each cluster.

Typically, there are multiple versions of MPI libraries and compilers on each cluster.

Modules are used to select a specific MPI library or compiler: More info HERE.

For example, using modules:

- List which modules are currently loaded
- Show all available modules
- Load a different MPI module
- Load a different compiler module
- Confirm newly loaded modules

See examples here

MVAPICH

General Info

MVAPICH MPI is developed and supported by the Network-Based Computing Lab at Ohio State University.

Available on all of LC's Linux clusters.

MVAPICH2

- Default MPI implementation
- Multiple versions available
- MPI-2 and MPI-3 implementations based on MPICH MPI library from Argonne National Laboratory. Versions 1.9 and later implement MPI-3 according to the developer's documentation.
- Thread-safe

To see what versions are available, and/or to select an alternate version, use Modules commands. For example:

```
module avail mvapich         (list available modules)
module load mvapich2/2.3     (use the module of interest)
```

Compiling

See the MPI Build Scripts table below.

Running

MPI executables are launched using the SLURM `srun` command with the appropriate options. For example, to launch an 8-process MPI job split across two different nodes in the `pdebug` pool:

```
srun -N2 -n8 -ppdebug a.out
```

The `srun` command is discussed in detail in the Running Jobs section of the Linux Clusters Overview tutorial.

Open MPI

General Information

Open MPI is a thread-safe, open source MPI implementation developed and supported by a consortium of academic, research, and industry partners.

Available on all LC Linux clusters. However, you'll need to load the desired module first. For example:

```
module avail                (list available modules)
module load openmpi/3.0.1   (use the module of interest)
```

This ensures that LC's MPI wrapper scripts point to the desired version of Open MPI.

Compiling

See the MPI Build Scripts table below.

Running

Be sure to load the same Open MPI module that you used to build your executable. If you are running a batch job, you will need to load the module in your batch script.

Launching an Open MPI job can be done using the following commands. For example, to run a 48 process MPI job:

```
mpirun -np 48 a.out
mpiexec -np 48 a.out
srun -n 48 a.out
```

Documentation

Open MPI home page: http://www.open-mpi.org/

Intel MPI

- Available on LC's Linux clusters.
- Based on MPICH3. Supports MPI-3 functionality.
- Thread-safe
- Compiling and running Intel MPI programs: see the LC documentation at: https://lc.llnl.gov/confluence/pages/viewpage.action?pageId=137725526

CORAL Early Access and Sierra Clusters:

- The IBM Spectrum MPI library is the only supported implementation on these clusters.
- Based on Open MPI. Includes MPI-3 functionality.
- Thread-safe
- NVIDIA GPU support
- Compiling and running IBM Spectrum MPI programs: see the Sierra Tutorial at https://hpc.llnl.gov/training/tutorials/using-lcs-sierra-system

MPI Build Scripts

LC developed MPI compiler wrapper scripts are used to compile MPI programs on all LC systems.

Automatically perform some error checks, include the appropriate MPI #include files, link to the necessary MPI libraries, and pass options to the underlying compiler.

- Note: you may need to load a module for the desired MPI implementation, as discussed previously. Failing to do this will result in getting the default implementation.

The table below lists the primary MPI compiler wrapper scripts for LC's Linux clusters. For CORAL EA / Sierra systems, see the links provided above.

MPI Build Scripts - Linux Clusters			
Implementation	**Language**	**Script Name**	**Underlying Compiler**
MVAPCH2	**C**	**mpicc**	C compiler for loaded compiler package
	C++	**mpicxx** **mpic++**	C++ compiler for loaded compiler package
	Fortran	**mpif77**	Fortran77 compiler for loaded compiler package. Points to mpifort.
		mpif90	Fortran90 compiler for loaded compiler package. Points to mpifort.
		mpifort	Fortran 77/90 compiler for loaded compiler package.
Open MPI	**C**	**mpicc**	C compiler for loaded compiler package
	C++	**mpiCC** **mpic++** **mpicxx**	C++ compiler for loaded compiler package
	Fortran	**mpif77**	Fortran77 compiler for loaded compiler package. Points to mpifort.
		mpif90	Fortran90 compiler for loaded compiler package. Points to mpifort.
		mpifort	Fortran 77/90 compiler for loaded compiler package.

For additional information:

- See the man page (if it exists)
- Issue the script name with the -help option
- View the script yourself directly

Level of Thread Support

MPI libraries vary in their level of thread support:

- **MPI_THREAD_SINGLE** - Level 0: Only one thread will execute.
- **MPI_THREAD_FUNNELED** - Level 1: The process may be multi-threaded, but only the main thread will make MPI calls - all MPI calls are funneled to the main thread.
- **MPI_THREAD_SERIALIZED** - Level 2: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time. That is, calls are not made concurrently from two distinct threads as all MPI calls are serialized.
- **MPI_THREAD_MULTIPLE** - Level 3: Multiple threads may call MPI with no restrictions.

Consult the MPI_Init_thread() man page for details.

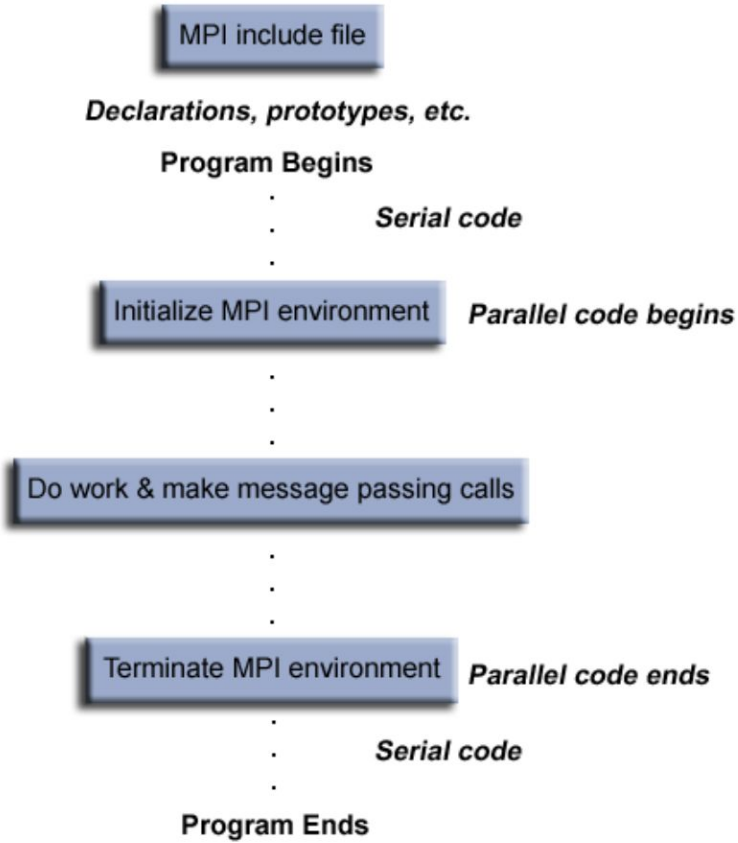
A simple C language example for determining thread level support is shown below.

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

int main( int argc, char *argv[] )
{
    int provided, claimed;

/*** Select one of the following
    MPI_Init_thread( 0, 0, MPI_THREAD_SINGLE, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_FUNNELED, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_SERIALIZED, &provided );
    MPI_Init_thread( 0, 0, MPI_THREAD_MULTIPLE, &provided );
***/

    MPI_Init_thread(0, 0, MPI_THREAD_MULTIPLE, &provided );
    MPI_Query_thread( &claimed );
        printf( "Query thread level= %d  Init_thread level= %d\n", claimed, provided );

    MPI_Finalize();
}
```

```
# Sample output:
Query thread level= 3  Init_thread level= 3
```

Getting Started

General MPI Program Structure:



Header File:

Required for all programs that make MPI library calls.

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

With MPI-3 Fortran, the `USE mpi_f08` module is preferred over using the include file shown above.

Format of MPI Calls:

C names are case sensitive; Fortran names are not.

Programs must not declare variables or functions with names beginning with the prefix `MPI_` or `PMPI_` (profiling interface).

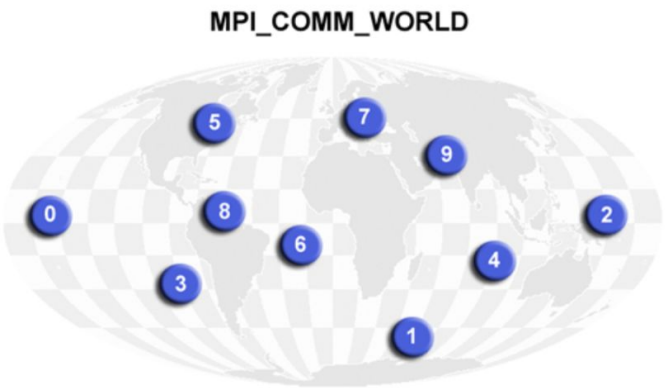
C Binding	
Format:	rc = MPI_Xxxxx(parameter, ...)
Example:	rc = MPI_Bsend(&buf,count,type,dest,tag,comm)
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	CALL MPI_XXXXX(parameter,..., ierr) call mpi_xxxxx(parameter,..., ierr)
Example:	CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

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).

Error Handling:

Most MPI routines include a return/error code parameter, as described in the "Format of MPI Calls" section above.

However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).

The standard does provide a means to override this default error handler. A discussion on how to do this is available HERE. You can also consult the error handling section of the relevant MPI Standard documentation located at http://www.mpi-forum.org/docs/.

The types of errors displayed to the user are implementation dependent.

Environment Management Routines

This group of routines is used for interrogating and setting the MPI execution environment, and covers an assortment of purposes, such as initializing and terminating the MPI environment, querying a rank's identity, querying the MPI library's version, 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

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

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

MPI Comm rank

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 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 Get version

Returns the version and subversion of the MPI standard that's implemented by the library.

```
MPI_Get_version (&version,&subversion)
MPI_GET_VERSION (version,subversion,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

C Language - Environment Management Routines

```
// required MPI include file
#include "mpi.h"
#include <stdio.h>

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

// initialize MPI
MPI_Init(&argc,&argv);

// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);


    // do some work with message passing


// done with MPI
MPI_Finalize();
}
```

Point to Point Communication Routines: General Concepts

First, a Simple Example:

The value of PI can be calculated in various ways. Consider the Monte Carlo method of approximating PI:

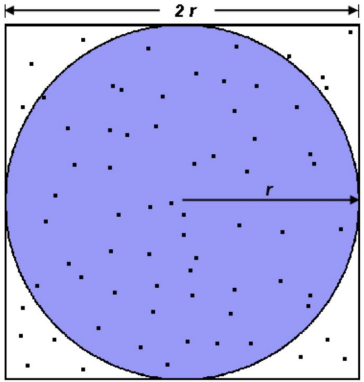
- Inscribe a circle with radius *r* in a square with side length of *2r*
- The area of the circle is Πr^2 and the area of the square is $4r^2$
- The ratio of the area of the circle to the area of the square is: $\frac{\Pi r^2}{4r^2} = \frac{\Pi}{4}$
- If you randomly generate *N* points inside the square, approximately $\frac{N * \Pi}{4}$ *of those points (M) should fall inside the circle.*
- Π is then approximated as:

$$\frac{N * \Pi}{4} = M$$

$$\frac{\Pi}{4} = \frac{M}{N}$$

$$\Pi = \frac{4 * M}{N}$$

Note that increasing the number of points generated improves the approximation.



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

Serial pseudo code for this procedure

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

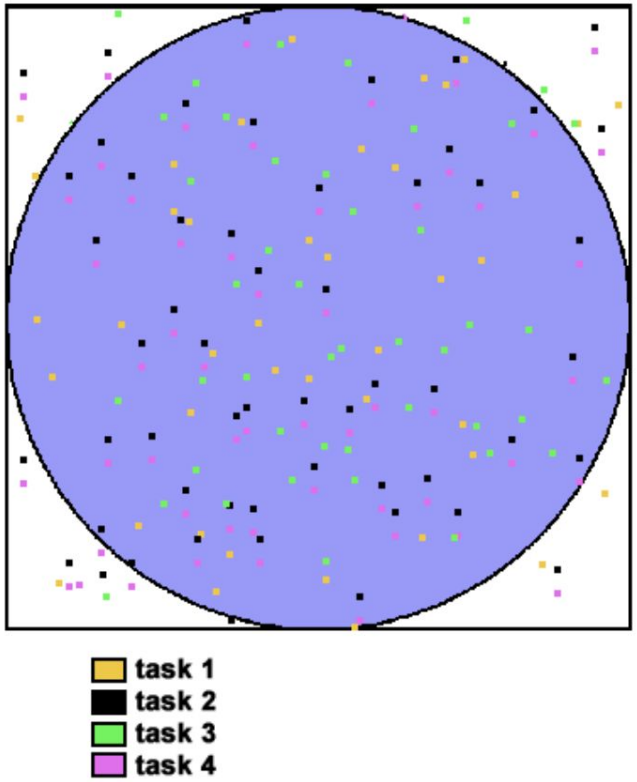
leads to an “embarrassingly parallel” solution:

- It breaks the loop iterations into chunks that can be executed by different tasks simultaneously.
- Each task executes its portion of the loop a number of times.
- Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
- Master task receives results from other tasks **using send/receive point-to-point operations**.

Pseudo code solution: **bold** highlights changes for parallelism.

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do


if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```

Key Concept: Divide work between available tasks which communicate data via point-to-point message passing calls.


task 1
task 2
task 3
task 4

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.

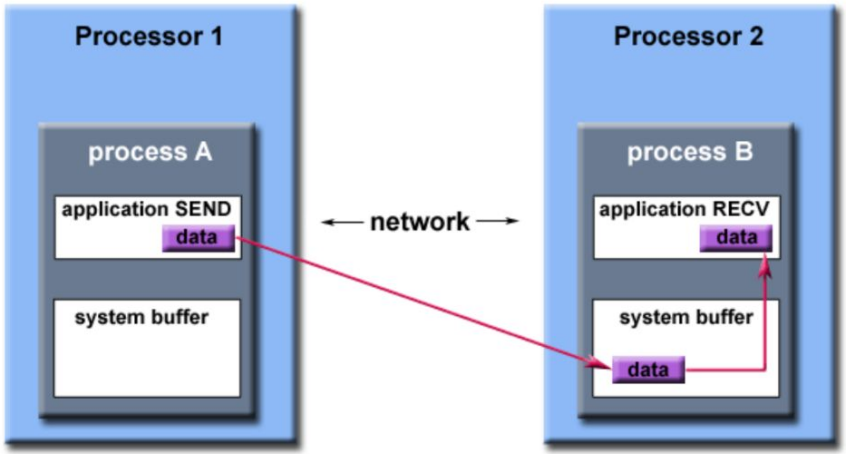
Buffering

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:



Path of a message buffered at the receiving process

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.

Blocking Send

```
myvar = 0;

for (i=1; i<ntasks; i++) {
   task = i;
   MPI_Send (&myvar ... ... task ...);
   myvar = myvar + 2

   /* do some work */

   }
```

Safe. Why?

Non-blocking Send

```
myvar = 0;

for (i=1; i<ntasks; i++) {
   task = i;
   MPI_Isend (&myvar ... ... task ...);
   myvar = myvar + 2;

   /* do some work */

   MPI_Wait (...);
   }
```

Unsafe. Why?

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.

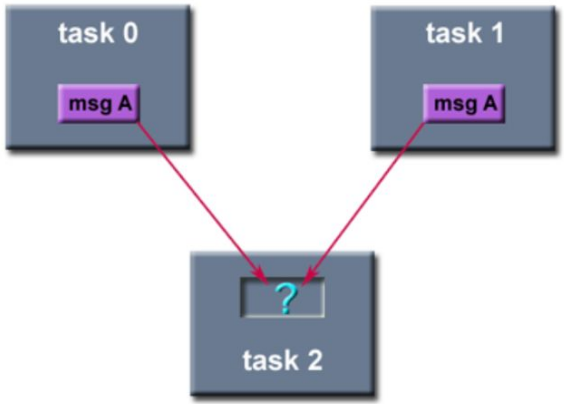
If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.

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:

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

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.

For datatype see here

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 is obtainable from Status via the MPI_Get_count routine. The constants MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE can be substituted if a message's source, tag or size will be be queried later.

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) MPI_SSEND (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 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)
```

MPI Get count

Returns the source, tag and number of elements of datatype received. Can be used with both blocking and non-blocking receive operations. 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_Get_count (&status,datatype,&count)
MPI_GET_COUNT (status,datatype,count,ierr)
```

Examples: Blocking Message Passing Routines

Task 0 pings task 1 and awaits return ping

C Language - Blocking Message Passing Example

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

main(int argc, char *argv[])  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;   // required variable for receive routines

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

// task 0 sends to task 1 and waits to receive a return message
if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    }

// task 1 waits for task 0 message then returns a message
else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }

// query receive Stat variable and print message details
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();
}
```

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)
```

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,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.

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

MPI_Test

MPI_Testany

MPI_Testall

MPI_Testsome

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)
```

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,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.

```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISSEND (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.

```
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).

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

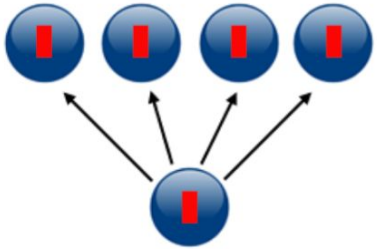
Examples: Non-blocking Message Passing Routines

Nearest neighbor exchange in a ring topology

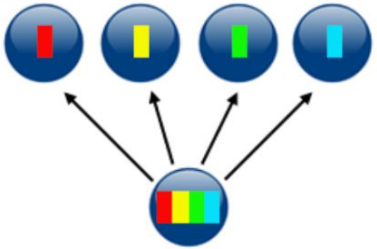
C Language - Non-blocking Message Passing Example

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

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];   // required variable for non-blocking calls
MPI_Status stats[4];   // required variable for Waitall routine

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

// determine left and right neighbors
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

// post non-blocking receives and sends for neighbors
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 while sends/receives progress in background

// wait for all non-blocking operations to complete
MPI_Waitall(4, reqs, stats);

    // continue - do more work

MPI_Finalize();
}
```

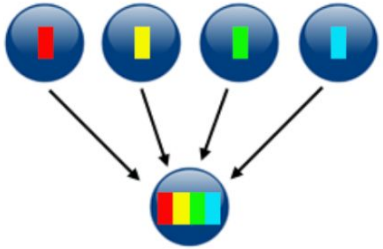
Collective Communication Routines



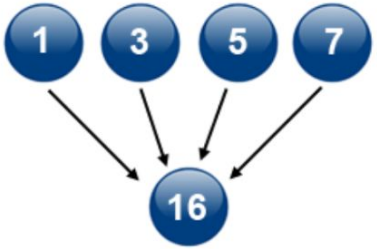
broadcast



scatter



gather



reduction

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.

Scope:

Collective communication routines must involve all processes within the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD. Additional communicators can be defined by the programmer. See the Group and Communicator Management Routines section for details.

Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.

It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Programming Considerations and Restrictions:

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.

MPI-2 extended most collective operations to allow data movement between intercommunicators (not covered here).

With MPI-3, collective operations can be blocking or non-blocking. Only blocking operations are covered in this tutorial.

Collective Communication Routines

MPI_Barrier

Synchronization operation. 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. Then all tasks are free to proceed.

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

MPI_Bcast

Data movement operation. 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

Data movement operation. 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

Data movement operation. 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

Data movement operation. 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

Collective computation operation. 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,comm)
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

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 Type
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

Collective computation operation + data movement. 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,ierr)
```

MPI Reduce scatter

Collective computation operation + data movement. 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

Data movement operation. 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

C Language - Collective Communications Example

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

main(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) {
    // define source task and elements to send/receive, then perform collective scatter
    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();
}
```

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
```