Outline

- Introduction
- MPI Implementations and Compilers
- Environment Management Routines
- Point to Point Communication Routines
- Collective Communication Routines
- Derived Data Types
- Group and Communicator Management Routines
- Virtual Topologies

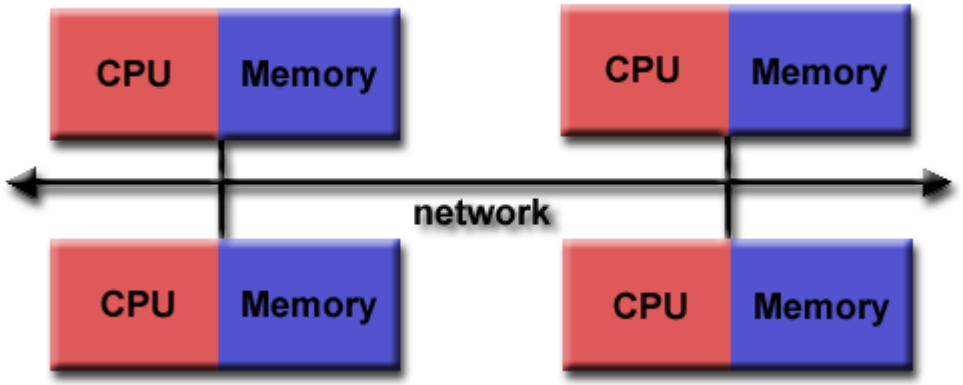
What is MPI?

- **MPI = Message Passing Interface**
- **MPI** is a *specification* for the developers and users of message passing libraries.
- MPI primarily addresses the message-passing parallel programming model:

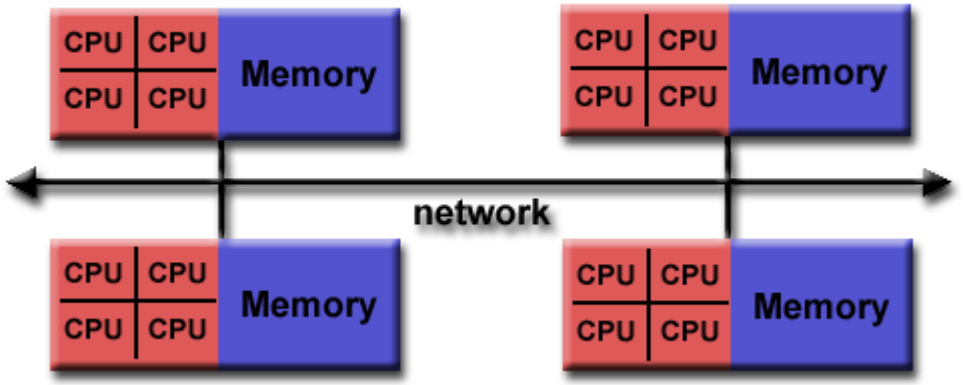
  data is moved from the address space of one *process* to that of another *process* through cooperative operations on each *process*
- The goal of MPI is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical
  - Portable
  - Efficient
  - Flexible

MPI 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.



Programming Model (cont'd)

- MPI implementers 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***, 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 which can be considered a standard. 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.
- **Availability:**
  - A variety of implementations are available, both vendor and public domain.
- **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.

References

- Documentation for all versions of the MPI standard is available at:
  http://www.mpi-forum.org/docs/
- MPI page at ANL:
  http://www.mcs.anl.gov/research/projects/mpi/index.htm
  which points to a lot of stuff about MPI, including papers, talks, and tutorials

Books

- **Using MPI: Portable Parallel Programming with the Message-Passing Interface**, 3rd Ed., *by* Gropp, Lusk, and Skjellum, MIT Press, 2014.
  http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981847
- **Using Advanced MPI: Modern Features of the Message-Passing Interface**, *by* Gropp, Hoefler, Thakur, and Lusk, MIT Press, 2014.
  http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981848
- **MPI: The Complete Reference – Vol. 1 The MPI Core**, *by* Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- **MPI: The Complete Reference – Vol. 2 The MPI-2 Extensions**, *by* Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- **Designing and Building Parallel Programs**, *by* Ian Foster, Addison-Wesley, 1995.
  http://www.mcs.anl.gov/~itf/dbpp/text/book.html

Tutorials & Examples

- MPI tutorial by Blaise Barney at LLNL:
  https://computing.llnl.gov/tutorials/mpi/
- MPI tutorials at ANL:
  http://www.mcs.anl.gov/research/projects/mpi/tutorial/index.html
- MPI tutorial by Rolf Rabenseifner at University of Stuttgart:
  https://fs.hlrs.de/projects/par/par_prog_ws/2004B/03_mpi_1_rab.pdf
- Examples:
  https://github.com/shawfdong/ams250/tree/master/examples/mpi

Confession: this presentation is built out of those tutorials!

MPI Implementations

- Many open and commercial implementations of the MPI standard are available, targeting C, C++, and Fortran programmers:
  - MPICH
  - Open MPI
  - MVAPICH
  - Intel MPI
  - Cray MPI
- Bindings are available for many other languages, including Perl, Python, R, Ruby, Java and Matlab.

MPICH

- http://www.mpich.org/
- MPICH is a high-performance and widely portable implementation of the MPI standard (MPI-1, MPI-2 and MPI-3).
- Distributed under a BSD-like license.
- Latest stable release: mpich-3.2
- The **CH** part of the name was derived from "Chameleon", which was a portable parallel programming library developed by William Gropp.
- MPICH is the foundation for the vast majority of MPI implementations, including MVAPICH, Intel MPI, Cray MPI, and many others.
- MPICH channels:
  - Nemesis – the default channel which supports multiple communication methods
  - Sock – a simple channel based on standard Unix sockets

MPI “Hello, world!” in C

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

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello, world! I am process %d of %d\n", rank, size);
  MPI_Finalize();

  return 0;
}
```

MPI “Hello, world!” in C++

```
#include "mpi.h"
#include <iostream>
using namespace std;

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

  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();
  cout << "Hello, world! I am process " << rank << " of " << size << endl;
  MPI::Finalize();

  return 0;
}
```

mpirun

- MPI start mechanism is implementation dependent
  - Most implementations, including **MPICH**, provide **mpirun**
  - MPI-2 standard defines **mpiexec**
- Learn more about **mpirun**:

```
mpirun –help
```

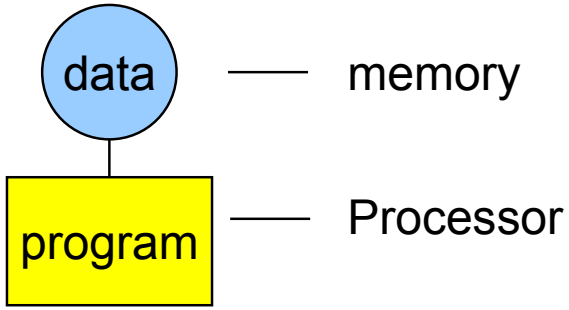
- What if you run a MPI program without using **mpirun?**

```
$ mpirun -n 2 ./hello.x
Hello, world! I am process 1 of 2
Hello, world! I am process 0 of 2
$ ./hello.x
Hello, world! I am process 0 of 1
```

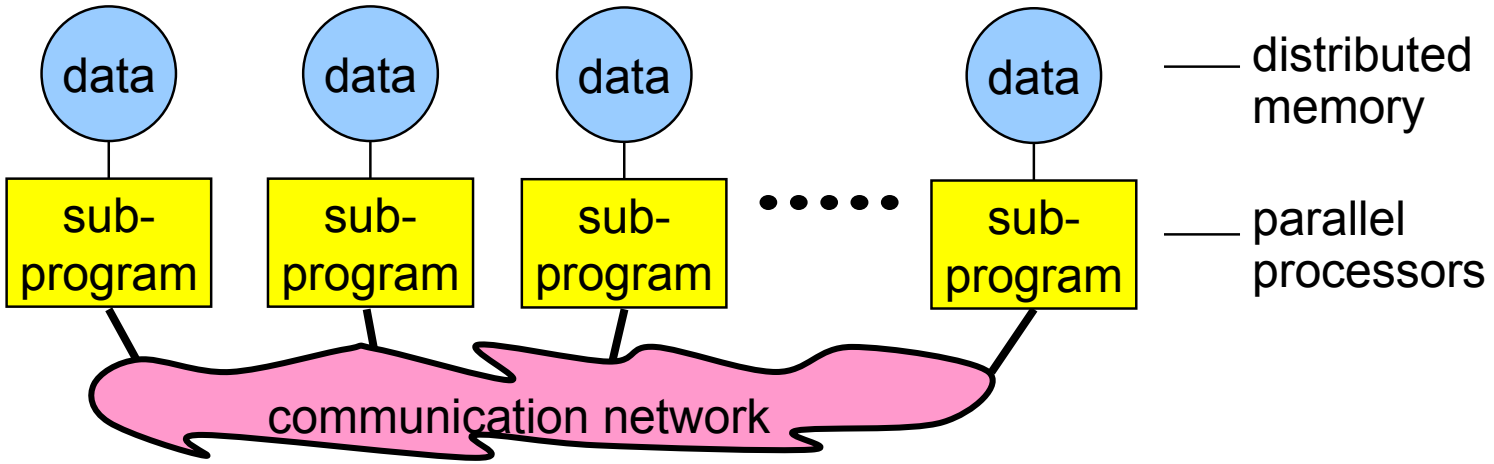
- Quiz: If there are only 2 CPU cores in your laptop, can you run a 4-processs (or 8-process) MPI job?

Programming Paradigms

- Sequential Programming Paradigm

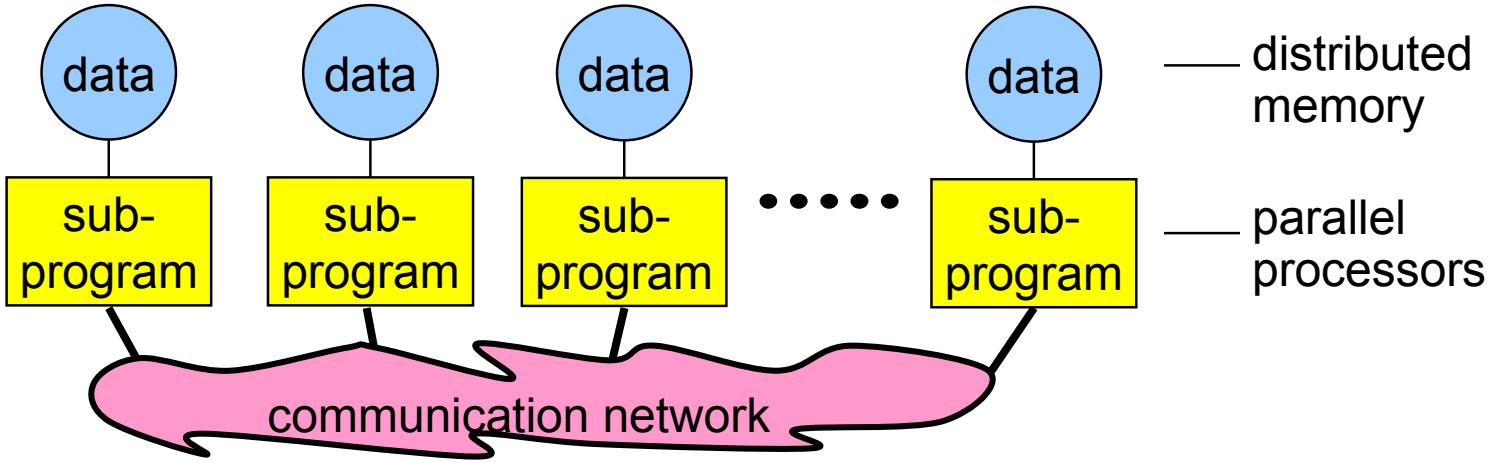


- Message-Passing Programming Paradigm



Message-Passing Programming Paradigm

- Each processor in a message passing program runs a sub-program:
  - written in a conventional sequential language, e.g., C/C++ or Fortran
  - typically the same on each processor (SPMD – **S**ingle **P**rogram **M**ultiple **D**ata):
    - the variables of each sub-program have the same names
    - but different locations (*distributed memory*) and different data!
    - i.e., all variables are private
  - but different sub-programs on different processors are allowed (MPMD)
  - sub-programs communicate via special send & receive routines (***message passing***)



SPMD vs. MPMD

- Both SPMD and MPMD are subcategories of MIMD (**M**ultiple **I**nstructions **M**ultiple **D**ata) in Flynn's taxonomy
- SPMD – **S**ingle **P**rogram **M**ultiple **D**ata, e.g.:

```
mpirun -n 32 ./mpi_hello.x
```

- MPMD – **M**ultiple **P**rogram **M**ultiple **D**ata, e.g.:

```
mpirun -n 1 pwd : -n 1 hostname
```

- MPI allows MPMD
  - But some implementations may only support SPMD, not MPMD
  - MPMD can be emulated with SPMD

Emulation of MPMD with SPMD

C:

```
int main(int argc, char **argv) {
    if (myrank < ... /* process should run the ocean model */) {
        ocean( /* arguments */ );
    }
    else {
        weather( /* arguments */ );
    }
}
```

Fortran:

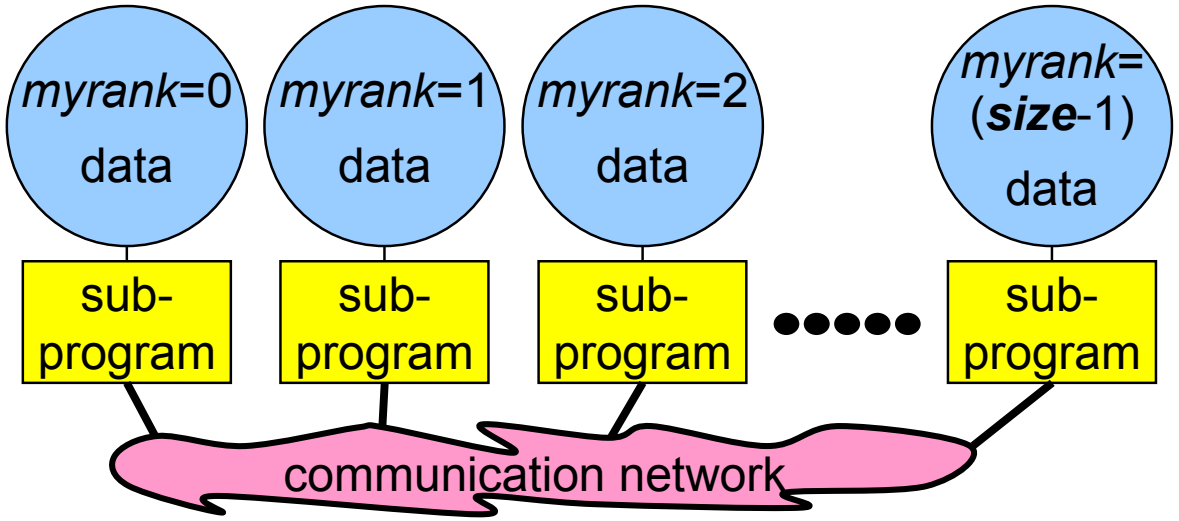
```
PROGRAM
    IF (myrank < ... ) THEN ! process should run the ocean model
        CALL  ocean ( some arguments )
    ELSE
        CALL  weather ( some arguments )
    ENDIF
END
```

Data and Work Distribution

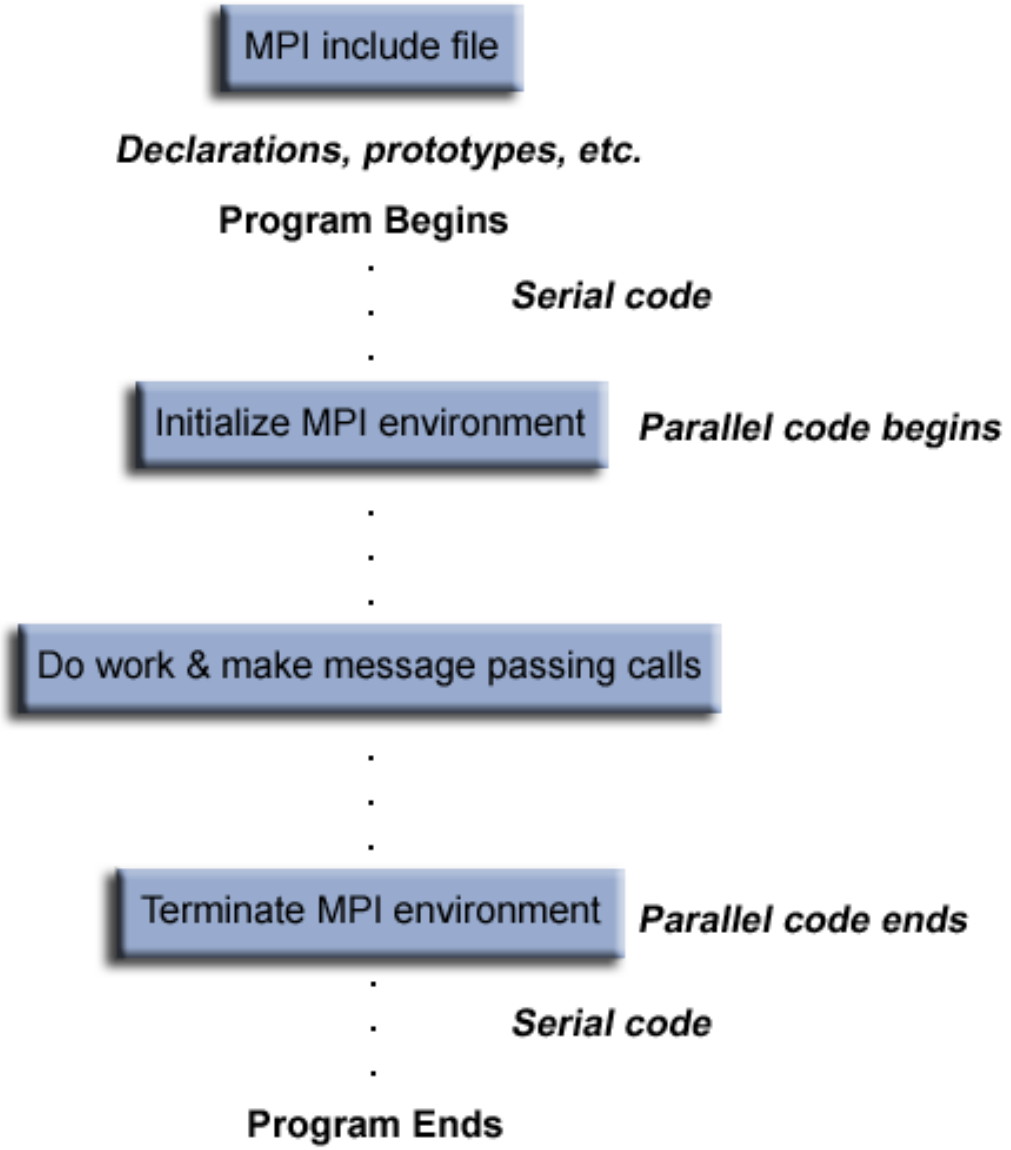
- The value of ***myrank*** is returned by a special library routine, e.g., *MPI_Comm_rank*
- The value of ***size*** (# of processes to be started) is specified as argument(s) to a special MPI initialization program (*mpirun* or *mpiexec* or *srun*), e.g:

```
mpirun -n 32 ./mpi_hello.x
```

- All distribution decisions are based on ***myrank***, e.g., which process works on which data



General MPI Program Structure




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

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  /* do some work */
  MPI_Finalize();

  return 0;
}
```

Header File

- Required for all programs that make MPI library calls.

C include file	Fortran include file	Fortran 90 module
#include "mpi.h"	include 'mpif.h'	USE mpi

- MPI-3 Fortran prefers:

```
USE mpi_f08
```

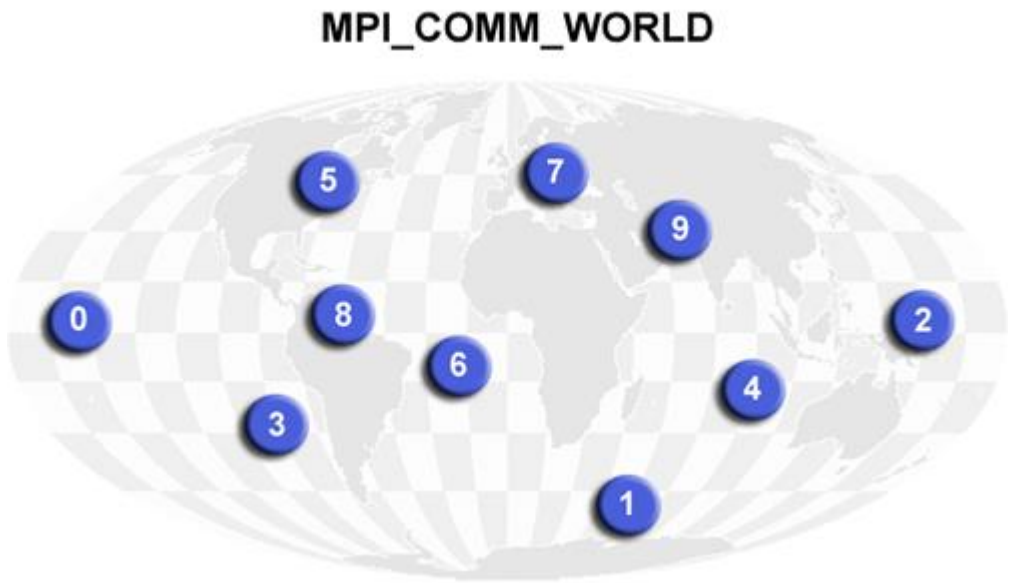
Format of MPI Calls

- **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)`
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.
- 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 at:

  https://computing.llnl.gov/tutorials/mpi/errorHandlers.pdf
- 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. The commonly used ones are:

- `MPI_Init`
- `MPI_Comm_size`
- `MPI_Comm_rank`
- `MPI_Abort`
- `MPI_Get_processor_name`
- `MPI_Get_version`
- `MPI_Initialized`
- `MPI_Wtime`
- `MPI_Wtick`
- `MPI_Finalize`

Open MPI documentation:
https://www.open-mpi.org/doc/current/

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.

C	`int MPI_Init(int *argc, char ***argv)`
Fortran	`MPI_INIT(IERROR)` `  INTEGER IERROR`

```
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ....
```

```
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_Init(ierror)
....
```

MPI_Comm_size

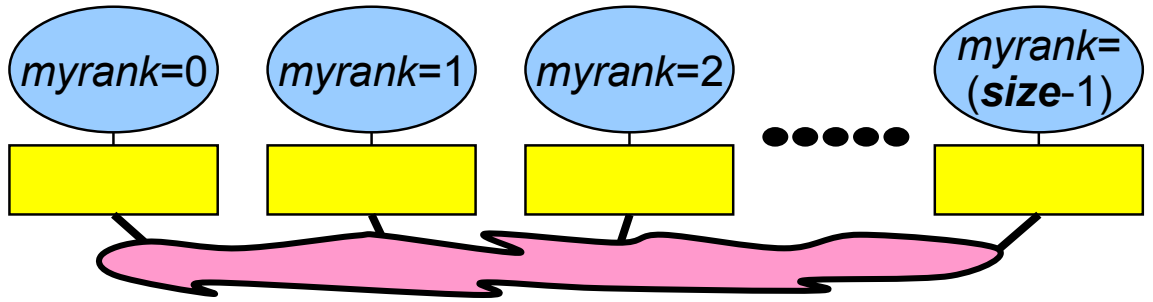
Returns the total number of MPI processes in the specified communicator, such as **MPI_COMM_WORLD**. If the communicator is **MPI_COMM_WORLD**, then it represents the number of MPI tasks available to your application.

C	`int MPI_Comm_size(MPI_Comm comm, int *size)`
Fortran	`MPI_COMM_SIZE(comm, size, ierror)` `    INTEGER comm, size, ierror`

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* (*size -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.

C	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`
Fortran	`MPI_COMM_RANK(comm, rank, ierror)` `  INTEGER comm, rank, ierror`




```
CALL MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierror)
```

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.

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

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

MPI_Get_version

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

C	`int MPI_Get_version(int *version, int *subversion)`
Fortran	`MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)` `    INTEGER    VERSION, SUBVERSION, IERROR`

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.

C	`int MPI_Initialized(int *flag)`

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

MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision), *since some time in the past*, on the calling processor.

C	`double MPI_Wtime()`
Fortran	`DOUBLE PRECISION MPI_WTIME()`

MPI_Wtick

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

C	`double MPI_Wtick()`
Fortran	`DOUBLE PRECISION 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	`int MPI_Finalize()`
Fortran	`MPI_FINALIZE(IERROR)` `  INTEGER  IERROR`

Handles

- Handles identify MPI objects
- For the programmer, handles are:
  - predefined constants in *mpi.h* or *mpif.h*
    - example: **MPI_COMM_WORLD**
    - predefined values exist only after **MPI_Init** was called
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - in Fortran: **INTEGER**
    - in C: special MPI typedefs, e.g., **MPI_Comm**
- Handles refer to internal MPI data structures

Environment Management Routines Example in C

(mpi_hello.c)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int  size, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    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, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf ("Number of tasks= %d My rank= %d Running on %s\n", size, rank,
            hostname);
    /*******  do some work *******/
    MPI_Finalize();
}
```

Environment Management Routines Example in Fortran

(mpi_hello.f)

```
      program simple
      include 'mpif.h'
      integer size, rank, len, ierr
      character(MPI_MAX_PROCESSOR_NAME) hostname
      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, size, ierr)
      call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
      print *, 'Number of tasks=', size, ' My rank=', rank,
     &         ' Running on=', hostname

C ****** do some work ******
      call MPI_FINALIZE(ierr)
      end
```

Point-to-Point Communication

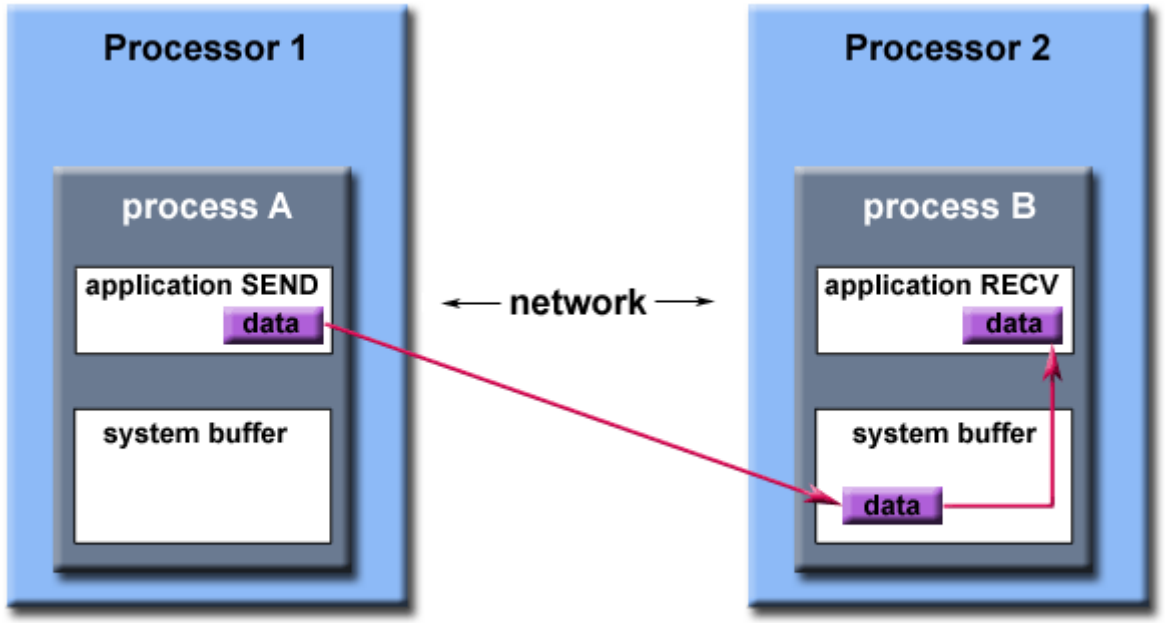
- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks/processes. One task is performing a send operation and the other task is performing a *matching* receive operation.
- There are different types of send and receive routines used for different 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.

Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. 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.

Buffering (cont'd)

- 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*.



Path of a message buffered at the receiving process

- 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

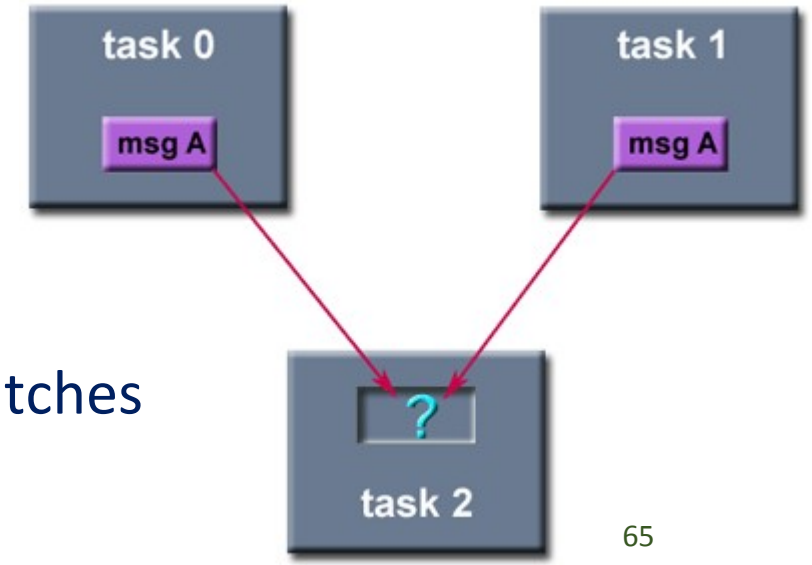
- A blocking send routine will only "return" after it is safe to modify the *application buffer* (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a *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.



MPI Message Passing Routine Arguments

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

Blocking send	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking send	`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.

MPI Message Passing Routine Arguments (cont'd)

Data Type

For reasons of portability, MPI predefines its elementary data types. Programmers may also create their own data types.

- C Data Types:

  MPI_CHAR, MPI_WCHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, MPI_C_BOOL, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_UINT64_T, MPI_BYTE, MPI_PACKED

- Fortran Data Types:

  MPI_CHARACTER, MPI_INTEGER, MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_REAL, MPI_REAL2, MPI_REAL4, MPI_REAL8, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_DOUBLE_COMPLEX, MPI_LOGICAL, MPI_BYTE, MPI_PACKED

MPI Message Passing Routine Arguments (cont'd)

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.

MPI Message Passing Routine Arguments (cont'd)

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**. In Fortran, it is an integer array of size **MPI_STATUS_SIZE**. Additionally, the actual number of bytes received is 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.

MPI_Get_count

MPI_Get_count gets the number of top-level elements received.

C	`int MPI_Get_count(const MPI_Status *status,`
	`  MPI_Datatype datatype,`
	`  int *count)`

Fortran	`MPI_MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)` `  INTEGER  STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR`

Blocking Point-to-Point Communication Routines

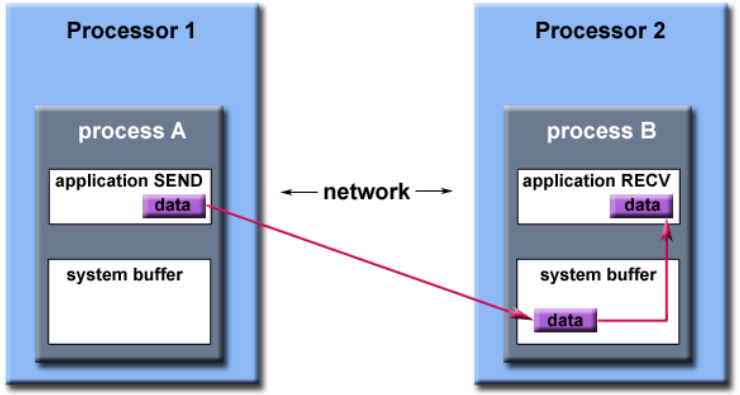
The more commonly used MPI blocking message passing routines are:

- `MPI_Send`
- `MPI_Recv`
- `MPI_Ssend`
- `MPI_Bsend`
- `MPI_Buffer_attach`
- `MPI_Buffer_detach`
- `MPI_Rsend`
- `MPI_Sendrecv`
- `MPI_Wait`
- `MPI_Waitany`
- `MPI_Waitall`
- `MPI_Waitsome`
- `MPI_Probe`

Open MPI documentation:
https://www.open-mpi.org/doc/current/

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 shortly) to implement the basic blocking send.



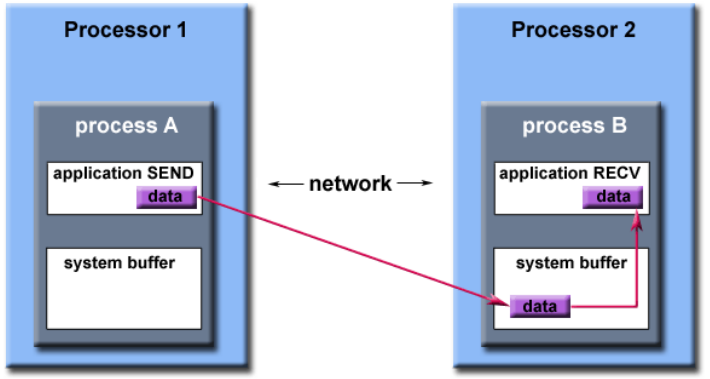
Path of a message buffered at the receiving process

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

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

MPI_Recv

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



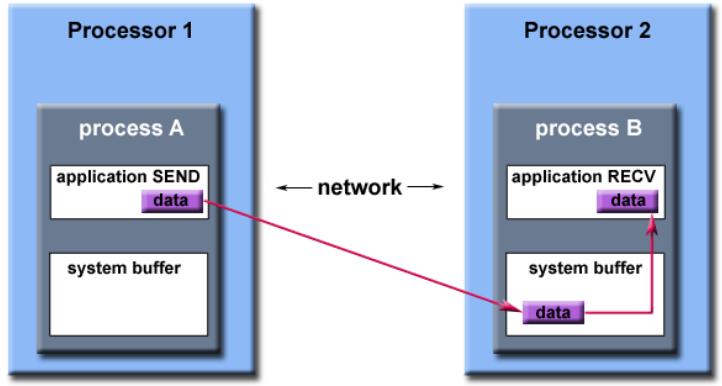
Path of a message buffered at the receiving process

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

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

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.



Path of a message buffered at the receiving process

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

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

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.

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

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

```
MPI_Buffer_attach( b, n*sizeof(double) + MPI_BSEND_OVERHEAD );
for (i=0; i<m; i++) {
    MPI_Bsend( buf, n, MPI_DOUBLE, ... );
}
MPI_Buffer_detach( b, &size );
```

MPI_Buffer_attach & MPI_Buffer_detach

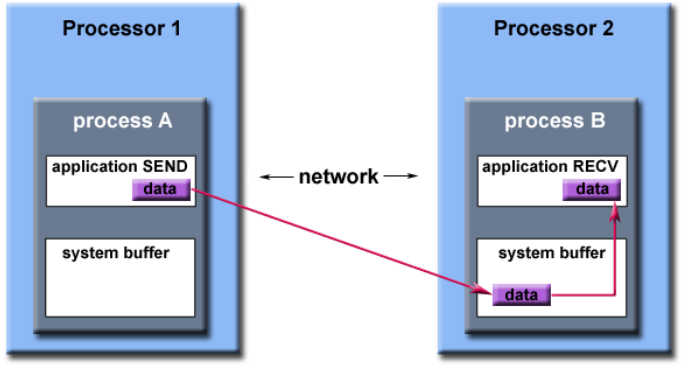
Used by programmer to allocate/deallocate message buffer space to be used by the **MPI_Bsend** routine. The size argument is specified in actual data bytes - not a count of data elements. Only one buffer can be attached to a process at a time.

C	`int MPI_Buffer_attach(void *buf, int size)`
	
	`int MPI_Buffer_detach(void *buf, int *size)`

Fortran	`MPI_BUFFER_ATTACH(BUF, SIZE, IERROR)`
	`    <type>    BUF(*)`
	`    INTEGER    SIZE, IERROR`
	
	`MPI_BUFFER_DETACH(BUF, SIZE, IERROR)`
	`    <type>    BUF(*)`
	`    INTEGER    SIZE, IERROR`

MPI_Rsend

Blocking ready send. Should only be used if the programmer is certain that the matching receive has already been posted. It is an error if the receive is not posted before the ready send is called. Often simply implemented as an **MPI_Send** routine.



Path of a message buffered at the receiving process

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

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

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.

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

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

MPI_Wait

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	`int MPI_Wait(MPI_Request *request, MPI_Status *status)`
	`int MPI_Waitany(int count, MPI_Request array_of_requests[],`
	`  int *index, MPI_Status *status)`
	`int MPI_Waitall(int count, MPI_Request array_of_requests[],`
	`  MPI_Status *array_of_statuses)`
	`int MPI_Waitsome(int incount, MPI_Request array_of_requests[],`
	`  int *outcount, int array_of_indices[],`
	`  MPI_Status array_of_statuses[])`

Fortran	`MPI_WAIT(REQUEST, STATUS, IERROR)`
	`MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)`
	`MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)`
	`MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,`
	`  ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)`

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)*.

It allows checking of incoming messages, without actual receipt of them. The user can then decide how to receive them, based on the information returned by the probe in the status variable.

C	`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
Fortran	`MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)` `  INTEGER  SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR`

Blocking Message Passing Routines Example in C

(mpi_ping.c)

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

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

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  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();
}
```

Try it out

```
$ mpiicc ping.c -o ping.x
$ mpirun -n 2 ./ping.x
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1
```

Non-Blocking Point-to-Point Communication Routines

The more commonly used MPI non-blocking message passing routines are:

- `MPI_Isend`
- `MPI_Irecv`
- `MPI_Issend`
- `MPI_Ibsend`
- `MPI_Irsend`
- `MPI_Test`
- `MPI_Testany`
- `MPI_Testall`
- `MPI_Testsome`
- `MPI_Iprobe`

Open MPI documentation:
https://www.open-mpi.org/doc/current/

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.

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

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

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

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

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

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

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

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

MPI_Irsend

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

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

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

Recall: MPI_Wait

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	`int MPI_Wait(MPI_Request *request, MPI_Status *status)`
	`int MPI_Waitany(int count, MPI_Request array_of_requests[],`
	`  int *index, MPI_Status *status)`
	`int MPI_Waitall(int count, MPI_Request array_of_requests[],`
	`  MPI_Status *array_of_statuses)`
	`int MPI_Waitsome(int incount, MPI_Request array_of_requests[],`
	`  int *outcount, int array_of_indices[],`
	`  MPI_Status array_of_statuses[])`

Fortran	`MPI_WAIT(REQUEST, STATUS, IERROR)`
	`MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)`
	`MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)`
	`MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,`
	`  ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)`

MPI_Test

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	`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
	`int MPI_Testany(int count, MPI_Request array_of_requests[],`
	`  int *index, int *flag, MPI_Status *status)`
	`int MPI_Testall(int count, MPI_Request array_of_requests[],`
	`  int *flag, MPI_Status array_of_statuses[])`
	`int MPI_Testsome(int incount, MPI_Request array_of_requests[],`
	`  int *outcount, int array_of_indices[],`
	`  MPI_Status array_of_statuses[])`

Fortran	`MPI_TEST(REQUEST, FLAG, STATUS, IERROR)`
	`MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)`
	`MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)`
	`MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)`
	`MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,`
	`  ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)`

Recall: 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)*.

It allows checking of incoming messages, without actual receipt of them. The user can then decide how to receive them, based on the information returned by the probe in the status variable.

C	`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
Fortran	`MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)` `INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR`

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)*.

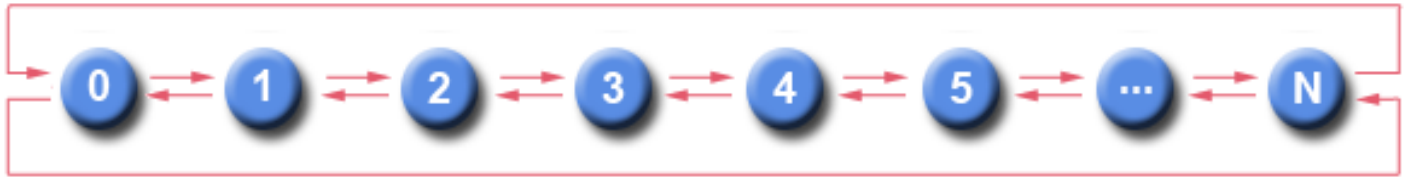
C	`int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,` `  MPI_Status *status)`

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

Non-Blocking Message Passing Routines Example in C

(mpi_ringtop.c)

Nearest neighbor exchange in a ring topology




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

int main(int argc, char *argv[])  {
  int size, 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, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

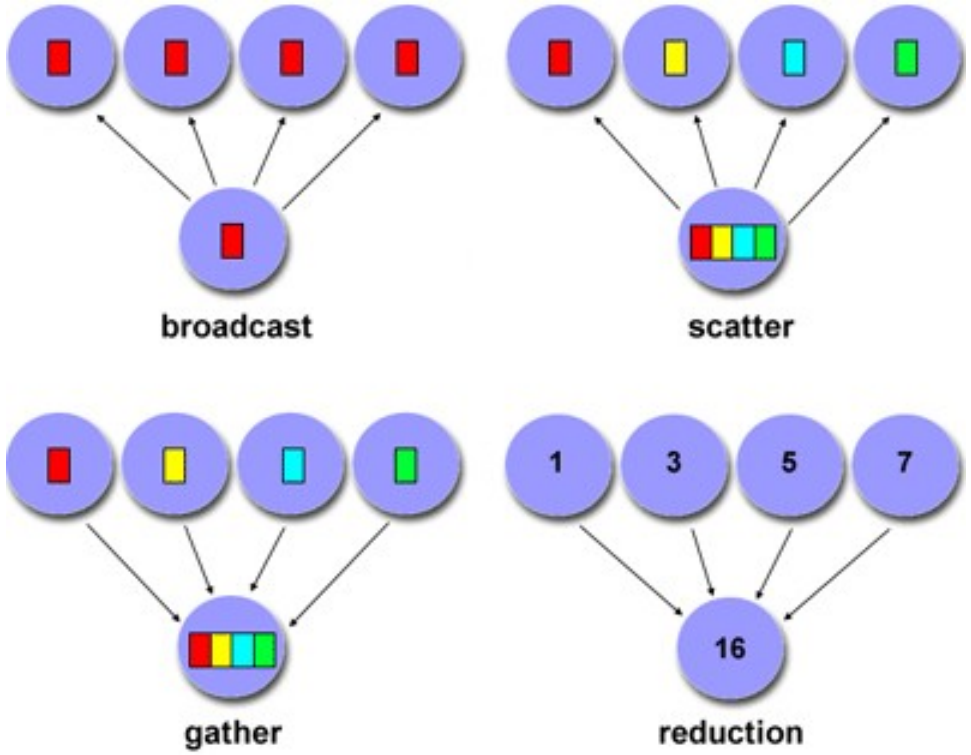
```
  prev = rank-1;
  next = rank+1;
  if (rank == 0)  prev = size - 1;
  if (rank == (size - 1))  next = 0;

  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();
}
```

Collective Communication Routines

- 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.
- 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.

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 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.
- 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.
- With MPI-3, collective operations can be blocking or non-blocking.

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.

C	`int MPI_Barrier(MPI_Comm comm)`
Fortran	`MPI_BARRIER(COMM, IERROR)` `INTEGER COMM, IERROR`

MPI_Bcast

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

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

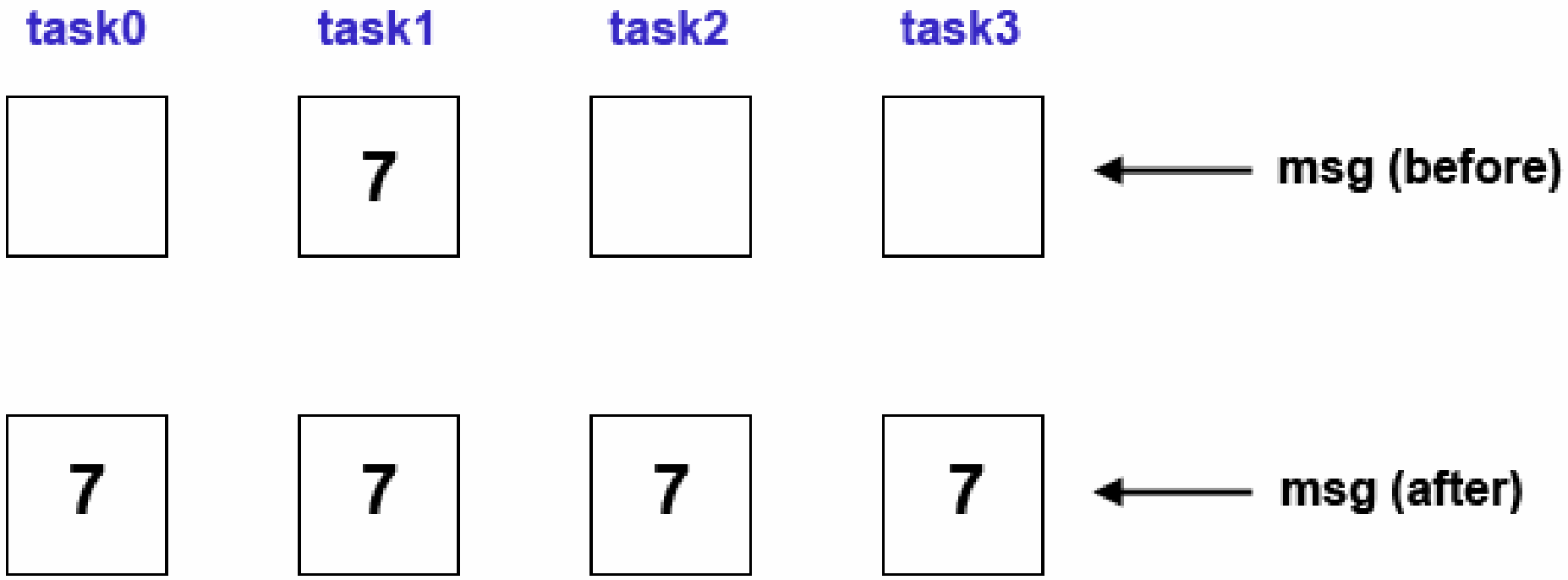
Fortran	`MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)`
	`<type> BUFFER(*)`
	`INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR`

MPI_Bcast

Broadcasts a message from one task to all other tasks in communicator

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

task1 contains the message to be broadcast



MPI_Scatter

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

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

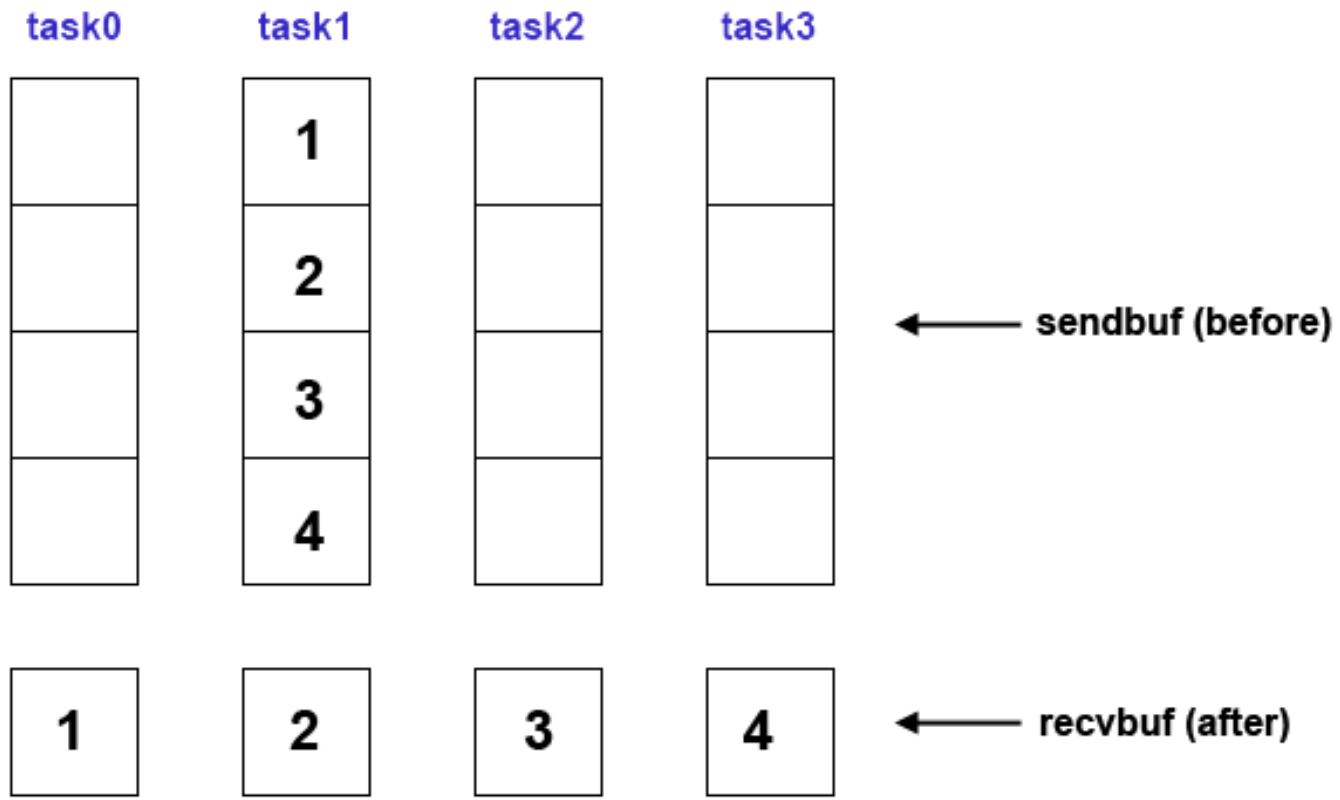
Fortran	`MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,`
	`  RECVTYPE, ROOT, COMM, IERROR)`
	`  <type>  SENDBUF(*), RECVBUF(*)`
	`  INTEGER  SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT`
	`  INTEGER  COMM, IERROR`

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered



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**.

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

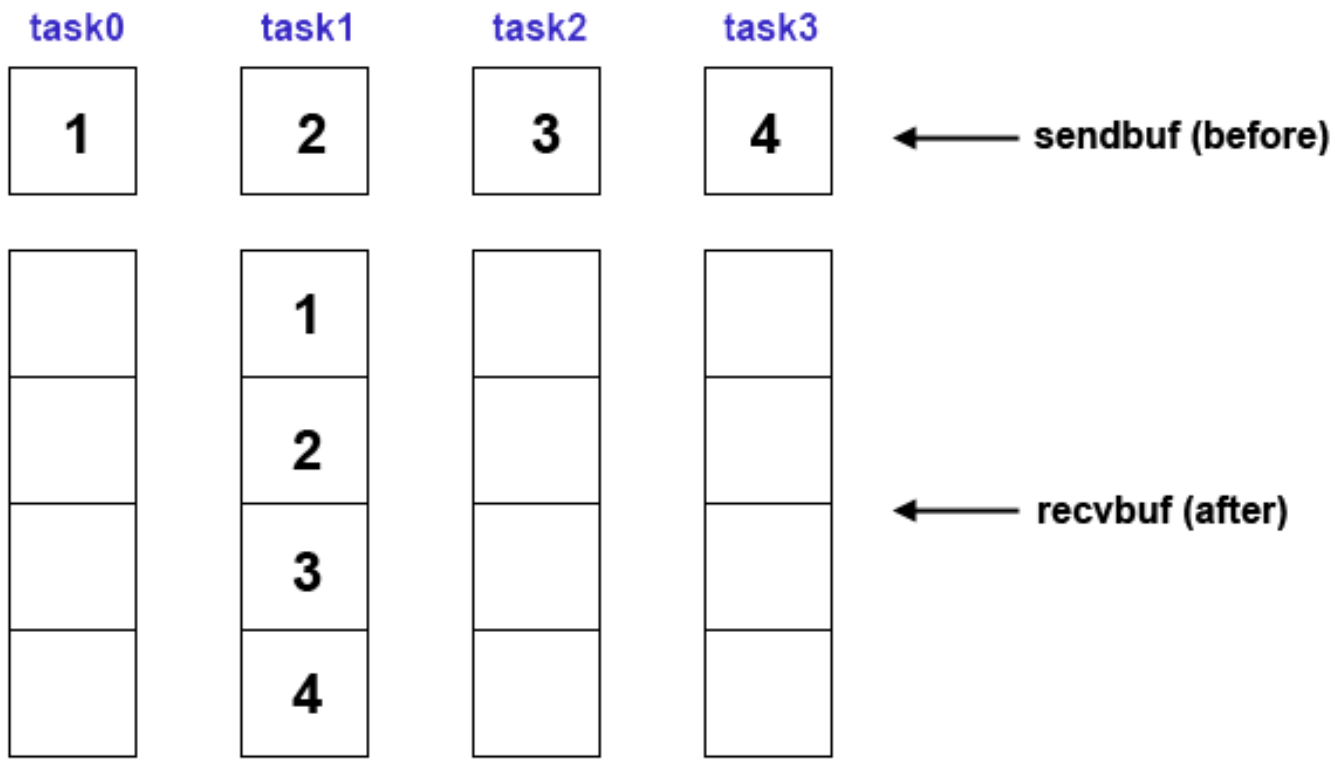
Fortran	`MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,`
	`RECVTYPE, ROOT, COMM, IERROR)`
	`<type> SENDBUF(*), RECVBUF(*)`
	`INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT`
	`INTEGER COMM, IERROR`

MPI_Gather

Gathers data from all tasks in communicator to a single task

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

message will be gathered into task1



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.

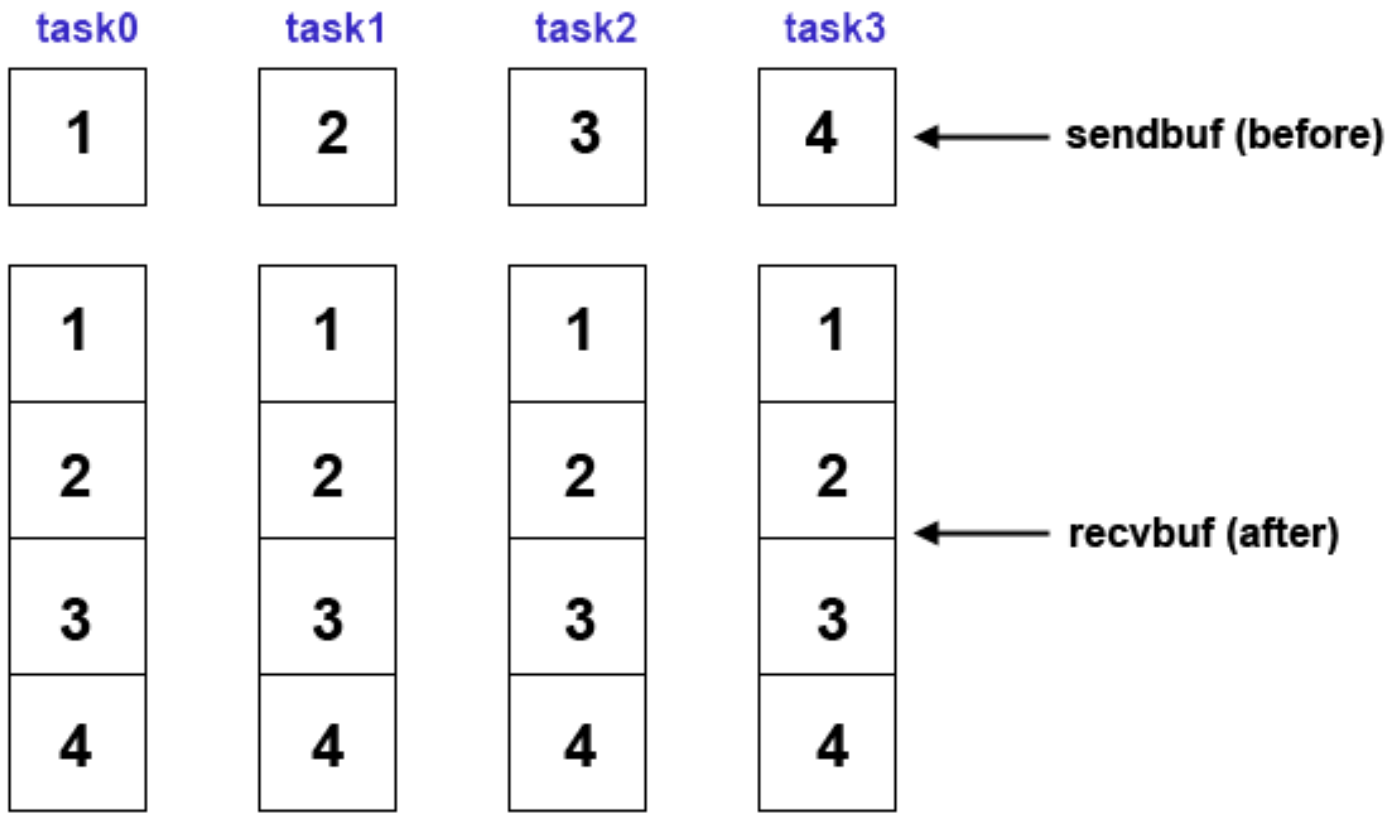
C	`int MPI_Allgather(const void *sendbuf, int  sendcount,`
	`  MPI_Datatype sendtype, void *recvbuf, int recvcount,`
	`  MPI_Datatype recvtype, MPI_Comm comm)`

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

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```



MPI_Reduce

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

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

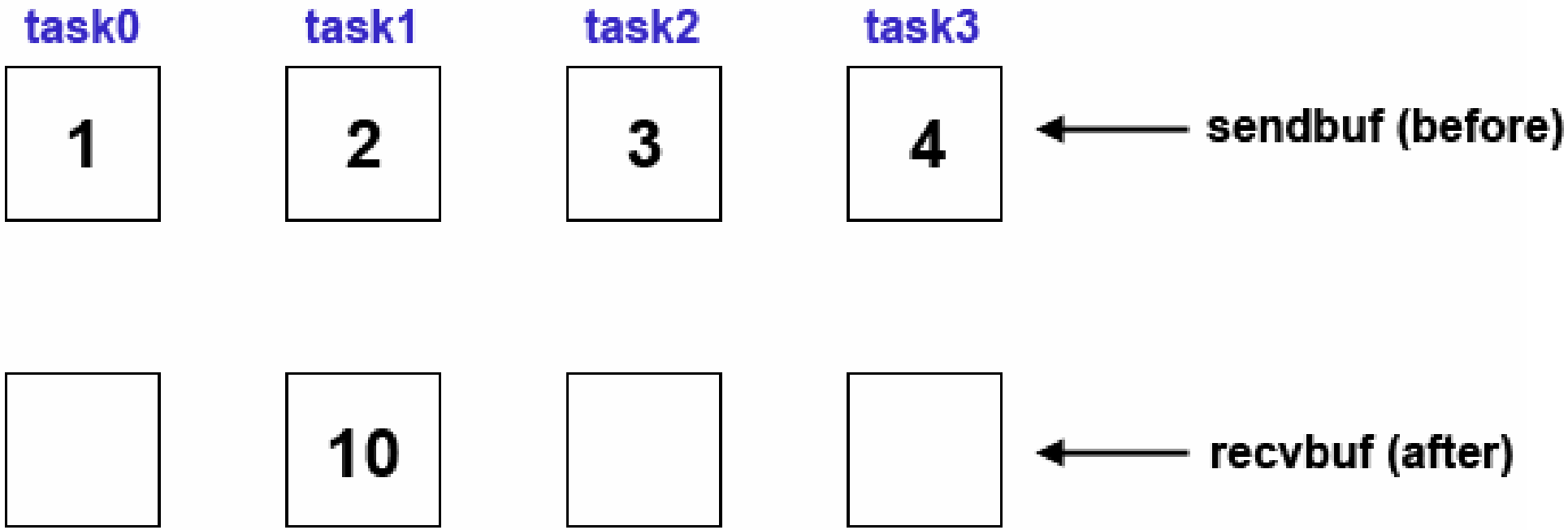
Fortran	`MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)`
	`  <type>  SENDBUF(*), RECVBUF(*)`
	`  INTEGER  COUNT, DATATYPE, OP, ROOT, COMM, IERROR`

MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result



Predefined MPI Reduction Operations

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

Users can also define their own reduction functions by using the **MPI_Op_create** routine.

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**.

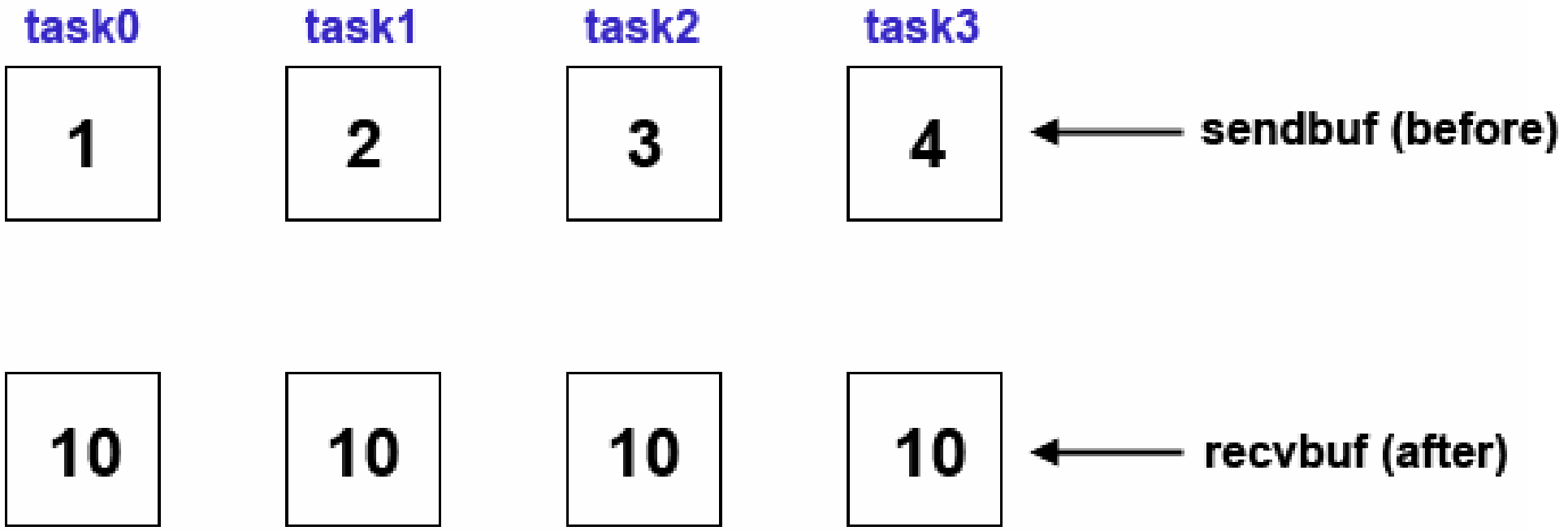
C	`int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,` `  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

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

MPI_Allreduce

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```



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.

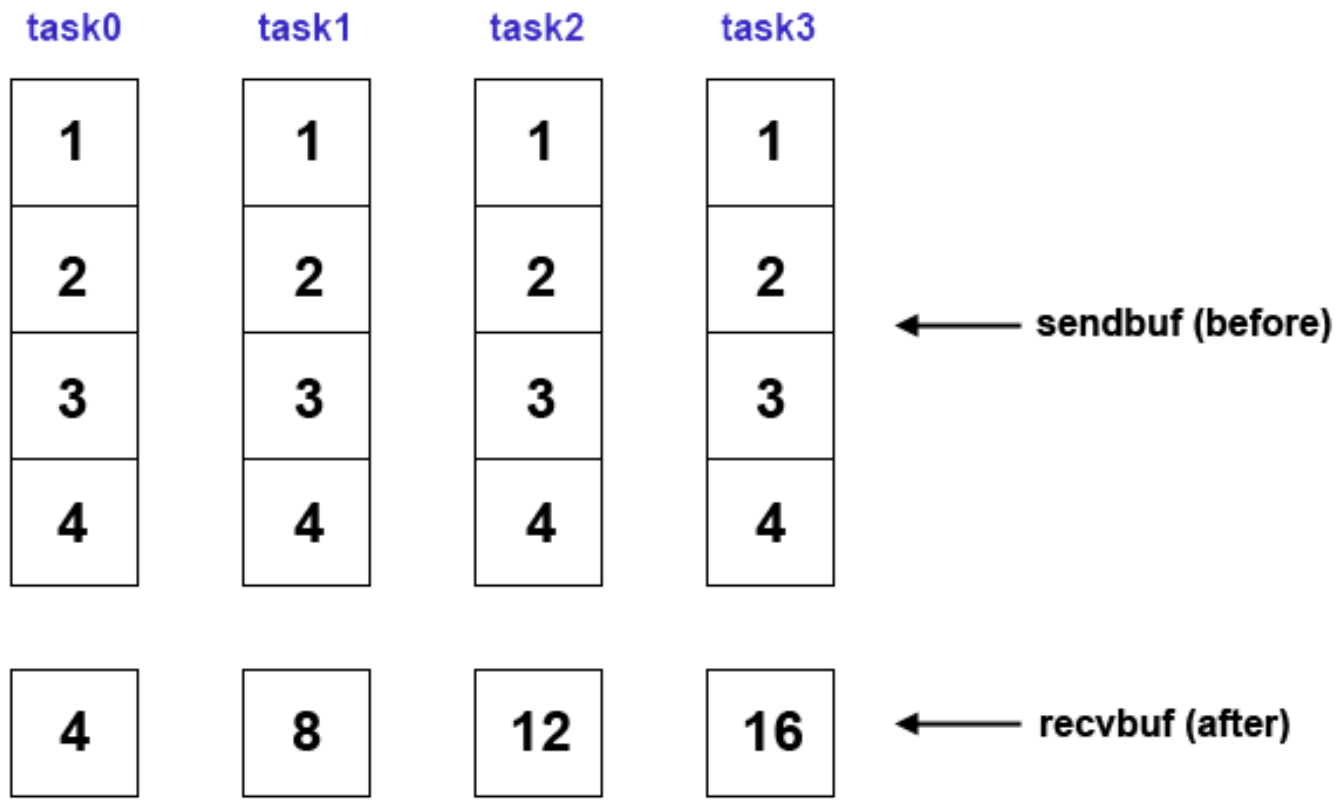
C	`int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

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

MPI_Reduce_scatter

Perform reduction on vector elements and distribute segments of result vector across all tasks in communicator

```
recvcnt = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount,
                   MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



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.

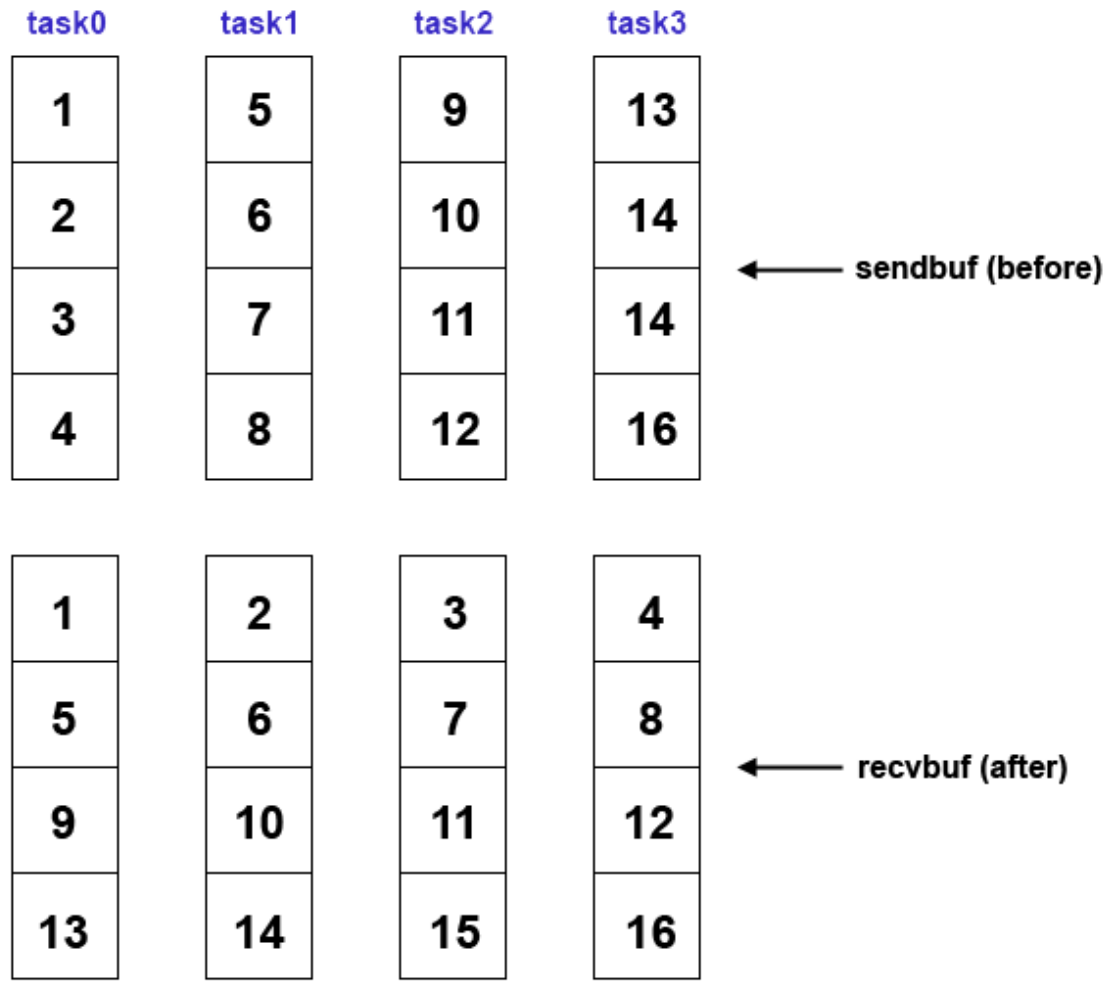
C	`int MPI_Alltoall(const void *sendbuf, int sendcount,`
	`  MPI_Datatype sendtype, void *recvbuf, int recvcount,`
	`  MPI_Datatype recvtype, MPI_Comm comm)`

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

MPI_Alltoall

Scatter data from all tasks to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT
             recvbuf, recvcnt, MPI_INT
             MPI_COMM_WORLD);
```



MPI_Scan

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

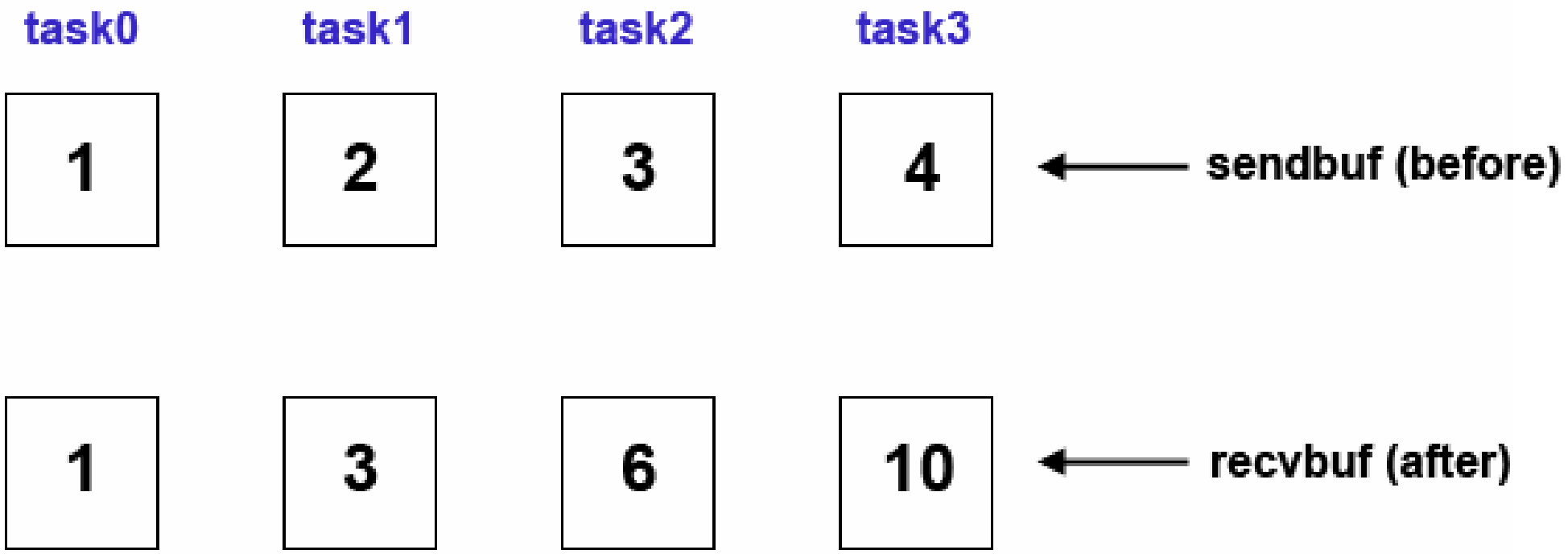
C	`int MPI_Scan(const void *sendbuf, void *recvbuf, int count,` `  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

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

MPI_Scan

Computes the scan (partial reductions) across all tasks in communicator

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT,
         MPI_SUM, MPI_COMM_WORLD);
```



Collective Communications Example in C

(mpi_scatter.c)

Perform a scatter operation on the rows of an array

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

int 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_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

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

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

  MPI_Finalize();
}
```

Try it out

```
$ mpiicc scatter.c -o scatter.x

$ mpirun -n 4 ./scatter.x
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
```

Derived Data Types

- As previously mentioned, MPI predefines its primitive data types.
- 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
- MPI_Type_vector
- MPI_Type_hvector
- MPI_Type_indexed
- MPI_Type_hindexed
- MPI_Type_struct
- MPI_Type_extent
- MPI_Type_commit
- MPI_Type_free

Open MPI documentation:
https://www.open-mpi.org/doc/current/

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.

NOTE: This function is deprecated in MPI-2.0 and replaced by **MPI_Type_get_extent** in MPI-3.0

C	`int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)`
Fortran	`MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)` `    INTEGER    DATATYPE, EXTENT, IERROR`

MPI_Type_commit

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

C	`int MPI_Type_commit(MPI_Datatype *datatype)`
Fortran	`MPI_TYPE_COMMIT(DATATYPE, IERROR)` `    INTEGER    DATATYPE, IERROR`

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	`int MPI_Type_free(MPI_Datatype *datatype)`
Fortran	`MPI_TYPE_FREE(DATATYPE, IERROR)` `    INTEGER    DATATYPE, IERROR`

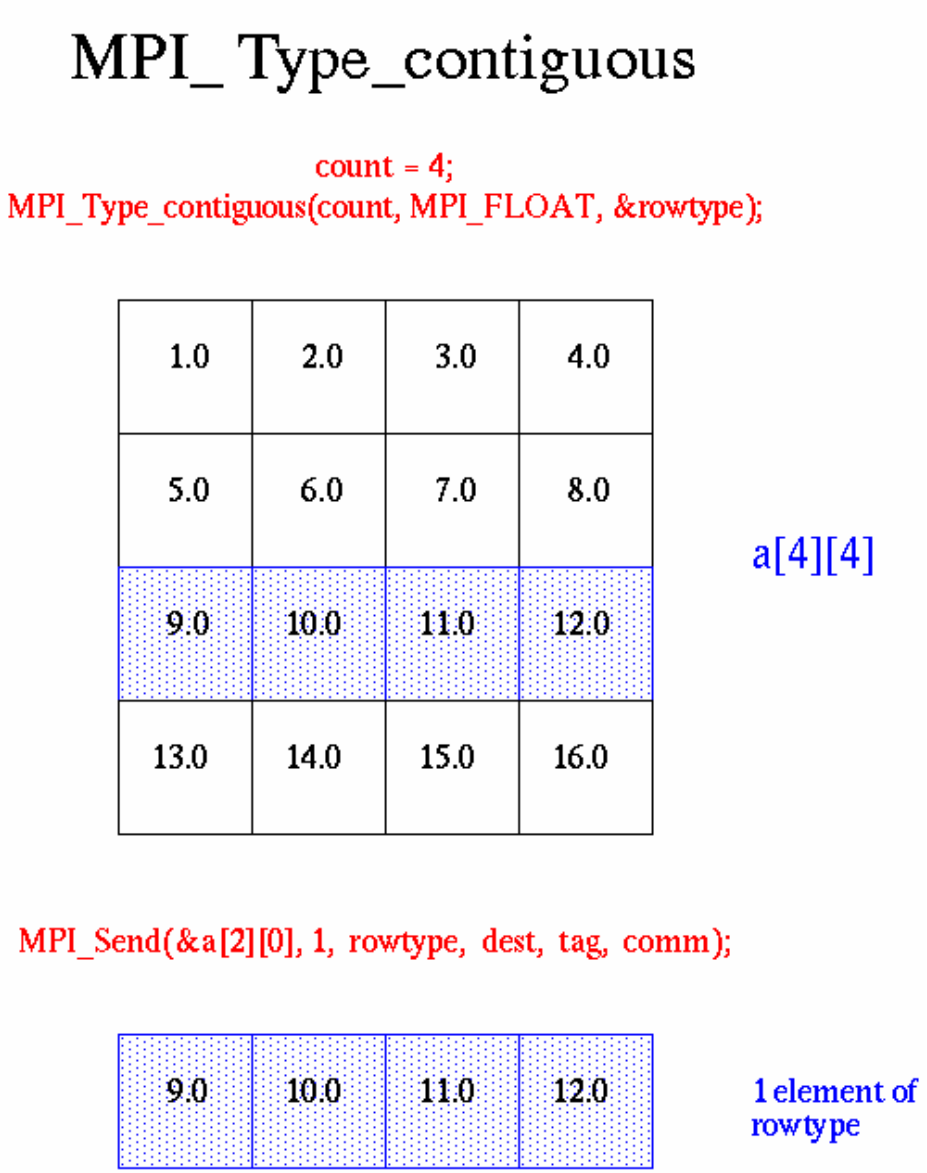
MPI_Type_contiguous

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

C	`int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`
Fortran	`MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)` `INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR`

Contiguous Derived Data Type Example

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



Contiguous Derived Data Type Example in C

(mpi_contig.c)

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

int main(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();
}
```

Try it out

```
$ mpiicc contiguous.c -o contiguous.x

$ mpirun -n 4 ./contiguous.x
Fatal error in MPI_Send: Other MPI error, error stack:
MPI_Send(186): MPI_Send(buf=0x7fff5160f580, count=1,
dtype=USER<contig>, dest=0, tag=1, MPI_COMM_WORLD) failed
MPID_Send(53): DEADLOCK: attempting to send a message to the
local process without a prior matching receive
```

DEADLOCK in the example codes in LLNL MPI tutorial:
https://computing.llnl.gov/tutorials/mpi/#Derived_Data_Types

Fixed Contiguous Derived Data Type Example in C

(mpi_contig_fixed.c)

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

int main(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_Request req; /* fix */
  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) {
    /* fix */
    MPI_Irecv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &req);
    if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
    }
    MPI_Wait(&req, &stat); /* fix */
    /* 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();
}
```

Try it out

```
$ mpiicc contiguous_fixed.c -o contiguous_fixed.x

$ mpirun -n 4 ./contiguous_fixed.x
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
```

MPI_Type_vector & MPI_Type_hvector

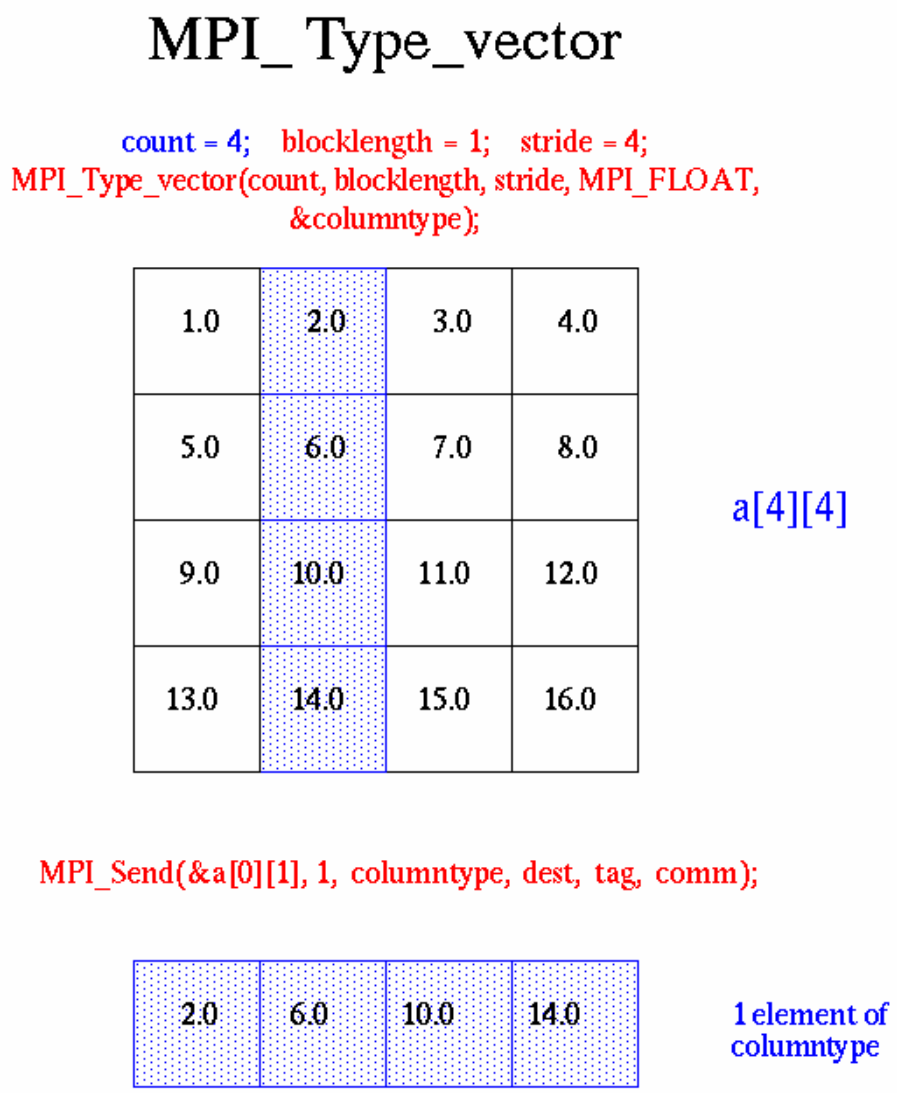
Similar to contiguous, but allows for regular gaps (stride) in the displacements. **MPI_Type_hvector** is identical to **MPI_Type_vector** except that stride is specified in bytes.

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

Fortran	`MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)`
	`  INTEGER  COUNT, BLOCKLENGTH, STRIDE, OLDTYPE`
	`  INTEGER  NEWTYPE, IERROR`
	
	`MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)`
	`  INTEGER  COUNT, BLOCKLENGTH, STRIDE, OLDTYPE`
	`  INTEGER  NEWTYPE, IERROR`

Vector Derived Data Type Example

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



Vector Derived Data Type Example in C

(mpi_vector_fixed.c)

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

int main(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_Request req;
  MPI_Status stat;
  MPI_Datatype columntype;

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

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

  if (numtasks == SIZE) {
    MPI_Irecv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &req);
    if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
    }
    MPI_Wait(&req, &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(&columntype);
  MPI_Finalize();
}
```

Try it out

```
$ mpiicc vector.c -o vector.x
$ mpirun -n 4 ./vector.x
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
rank= 0  b= 1.0 5.0 9.0 13.0
```

Note: same **DEADLOCK** bug in all "Derived Data Types" examples in the LLNL MPI tutorial:
https://computing.llnl.gov/tutorials/mpi/#Derived_Data_Types

I've applied the simple **IRECV** fix in this presentation!

MPI_Type_indexed & MPI_Type_hindexed

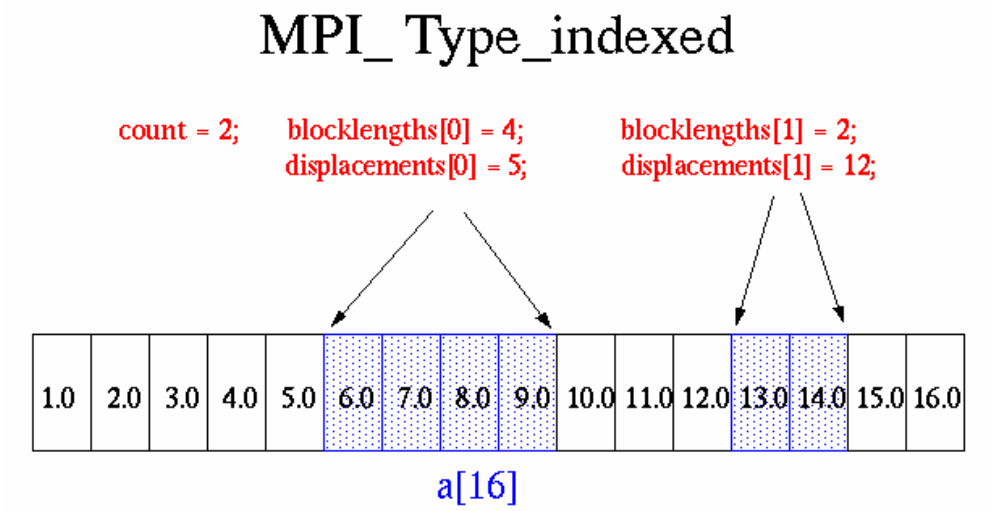
An array of displacements of the input data type is provided as the map for the new data type. **MPI_Type_hindexed** is identical to **MPI_Type_indexed** except that offsets are specified in bytes.

C	`int MPI_Type_indexed(int count, const int array_of_blocklengths[],`
	`  const int array_of_displacements[], MPI_Datatype oldtype,`
	`  MPI_Datatype *newtype)`
	`int MPI_Type_hindexed(int count, int *array_of_blocklengths,`
	`  MPI_Aint *array_of_displacements, MPI_Datatype oldtype,`
	`  MPI_Datatype *newtype)`

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

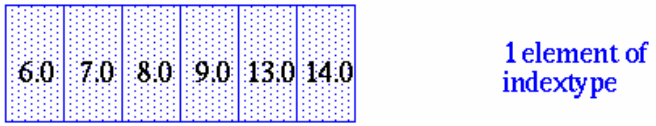
Indexed Derived Data Type Example

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




```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



Indexed Derived Data Type Example in C

(mpi_indexed_fixed.c)

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

int main(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_Request req;
  MPI_Status stat;
  MPI_Datatype indextype;

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

```
  blocklengths[0] = 4;
  blocklengths[1] = 2;
  displacements[0] = 5;
  displacements[1] = 12;
  MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
  MPI_Type_commit(&indextype);

  MPI_Irecv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &req);
  if (rank == 0) {
    for (i=0; i<numtasks; i++)
      MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

  MPI_Wait(&req, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
         rank, b[0], b[1], b[2], b[3], b[4], b[5]);

  MPI_Type_free(&indextype);
  MPI_Finalize();
}
```

Try it out

```
$ mpiicc indexed.c -o indexed.x
$ mpirun -n 4 ./indexed.x
rank= 0  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 1  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 2  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3  b= 6.0 7.0 8.0 9.0 13.0 14.0
```

MPI_Type_struct

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

NOTE: This function is deprecated in MPI-2.0 and replaced by **MPI_Type_create_struct** in MPI-3.0

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

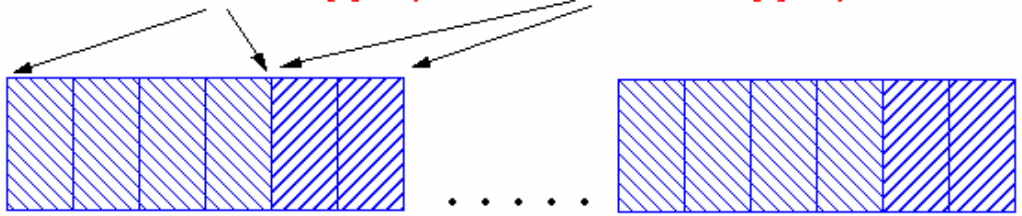
Fortran	`MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,`
	`  ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,`
	`  NEWTYPE, IERROR)`
	`  INTEGER  COUNT, ARRAY_OF_BLOCKLENGTHS(*)`
	`  INTEGER  ARRAY_OF_DISPLACEMENTS(*)`
	`  INTEGER  ARRAY_OF_TYPES(*), NEWTYPE, IERROR`

Struct Derived Data Type Example

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

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

Struct Derived Data Type Example in C

(mpi_struct_fixed.c)

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25
int main(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_Request req;
  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);
```

```
  MPI_Irecv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &req);
  /* 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_Wait(&req, &stat);
  /* Print a sample of what was received */
  printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank, p[3].x,
         p[3].y, p[3].z, p[3].velocity, p[3].n, p[3].type);
  MPI_Type_free(&particletype);
  MPI_Finalize();
}
```

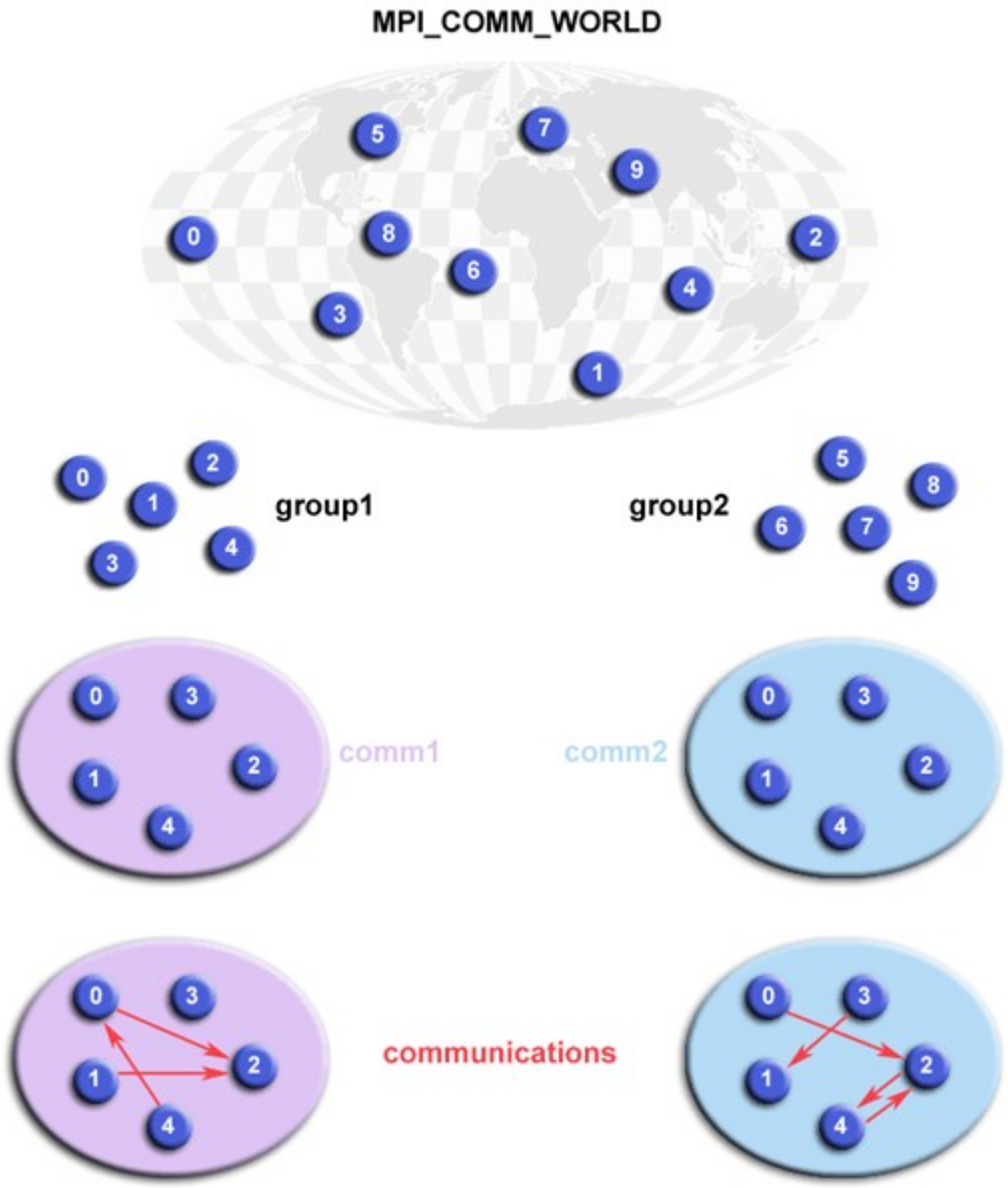
Try it out

```
$ mpiicc struct.c -o struct.x
$ mpirun -n 4 ./struct.x
rank= 0   3.00 -3.00 3.00 0.25 3 1
rank= 1   3.00 -3.00 3.00 0.25 3 1
rank= 2   3.00 -3.00 3.00 0.25 3 1
rank= 3   3.00 -3.00 3.00 0.25 3 1
```

Groups and 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 Routines Example in C

(mpi_group.c)

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

int main(int argc, char *argv[]) {
  int rank, new_rank, sendbuf, recvbuf, numtasks;
  int 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 communicator and then perform collective communications */
  MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
  MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

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

Try it out

```
$ mpiicc group.c -o group.x
$ mpirun -n 8 ./group.x
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 6 newrank= 2 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 5 newrank= 1 recvbuf= 22
rank= 7 newrank= 3 recvbuf= 22
```

Virtual Topologies

- 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 Virtual Topologies?

- 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.

Virtual Topology Example

Mapping of processes into a Cartesian virtual topology

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

Cartesian Virtual Topology Example in C

(mpi_cartesian.c)

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

Try it out

```
$ mpiicc cartesian.c -o cartesian.x
$ mpirun -n 16 ./cartesian.x
rank= 1 coords= 0 1  neighbors(u,d,l,r)= -1 5 0 2
rank= 4 coords= 1 0  neighbors(u,d,l,r)= 0 8 -1 5
rank= 3 coords= 0 3  neighbors(u,d,l,r)= -1 7 2 -1
rank= 9 coords= 2 1  neighbors(u,d,l,r)= 5 13 8 10
...
rank= 1                    inbuf(u,d,l,r)= -1 5 0 2
rank= 2                    inbuf(u,d,l,r)= -1 6 1 3
rank= 4                    inbuf(u,d,l,r)= 0 8 -1 5
rank= 13                    inbuf(u,d,l,r)= 9 -1 12 14
...
```

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

Try it out

```
$ mpiifort cartesian.f -o cartesian.x
$ mpirun -n 16 ./cartesian.x
 0     0     0    -1     4    -1     1
 1     0     1    -1     5     0     2
 2     0     2    -1     6     1     3
 8     2     0     4    12    -1     9
...
 1    -1     5     0     2
 8     4    12    -1     9
 2    -1     6     1     3
 0    -1     4    -1     1
...
```

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

A Brief Note on MPI-2

- Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1997.
- MPI-2 was a major revision to MPI-1 adding new functionality and corrections.
- Key areas of new functionality in MPI-2:
  - **Dynamic Processes** - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
  - **One-Sided Communications** - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
  - **Extended Collective Operations** - allows for the 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.

A Brief Note on MPI-3

- The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:
  - **Non-blocking Collective Operations** - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
  - **New One-sided Communication Operations** - to better handle different memory models.
  - **Neighborhood Collectives** - Extends the distributed graph and Cartesian process topologies with additional communication power.
  - **Fortran 2008 Bindings** - expanded from Fortran90 bindings
  - **MPIT Tool Interface** - This new tool interface allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
  - **Matched Probe** - Fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.