Message Passing Interface (MPI)

Zhengxiong Hou

Fall, 2022

Topics Overview

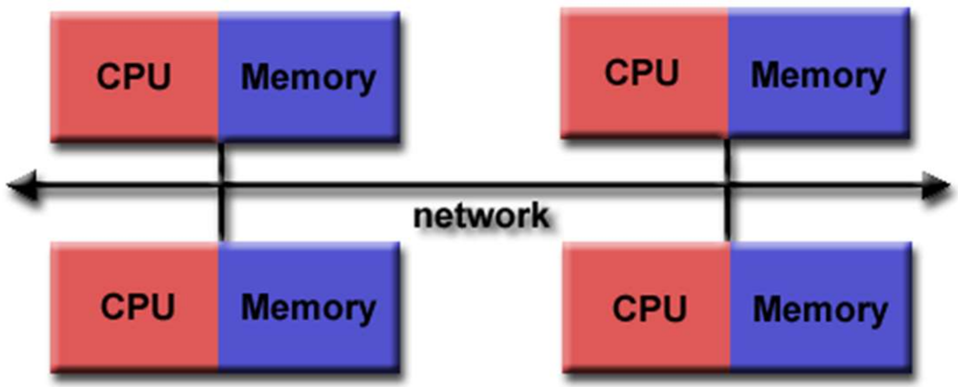
- What is MPI (MPI Basics)?
- Why MPI?
- Compiling and Running MPI programs
- MPI Routines-C and Fortran
- Examples

What is MPI (Message Passing Interface) ?

- **MPI *defines a standard library of functions* for message-passing that can be used to develop portable message-passing programs using either C/C++ or Fortran.**
  - **It is not a new programming language**
  - 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.
- **Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.**

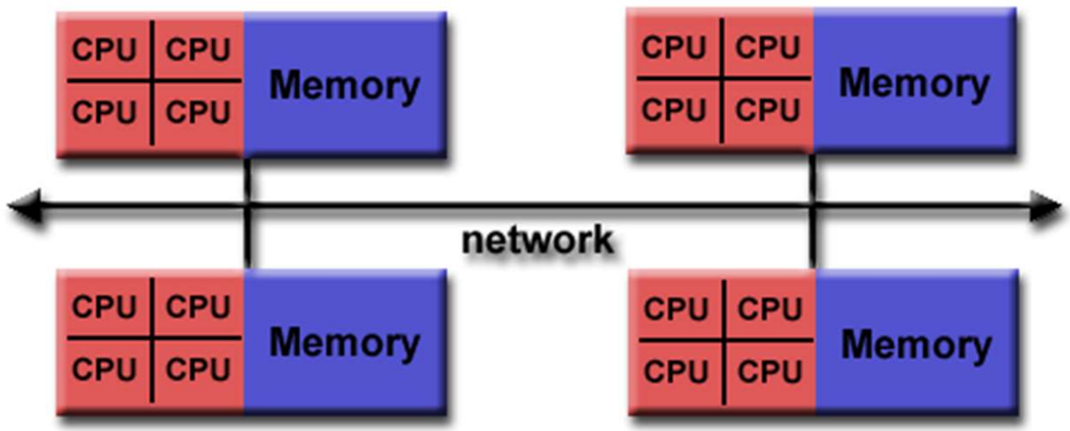
Programming Model:

- 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.



History and Evolution

- MPI has resulted from the efforts of numerous individuals and groups that began in 1992.

  -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)

June 9, 2021: MPI 4.0 Standard

MPI-1.0

- Point-to-point communication
- Collective communication
- Data types
- Non-blocking communication

A Brief Word on MPI-2 and MPI-3

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 1998.
- 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.

MPI-3:

- The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:
    - **Nonblocking 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** - 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.

More Information on MPI-2 and MPI-3:

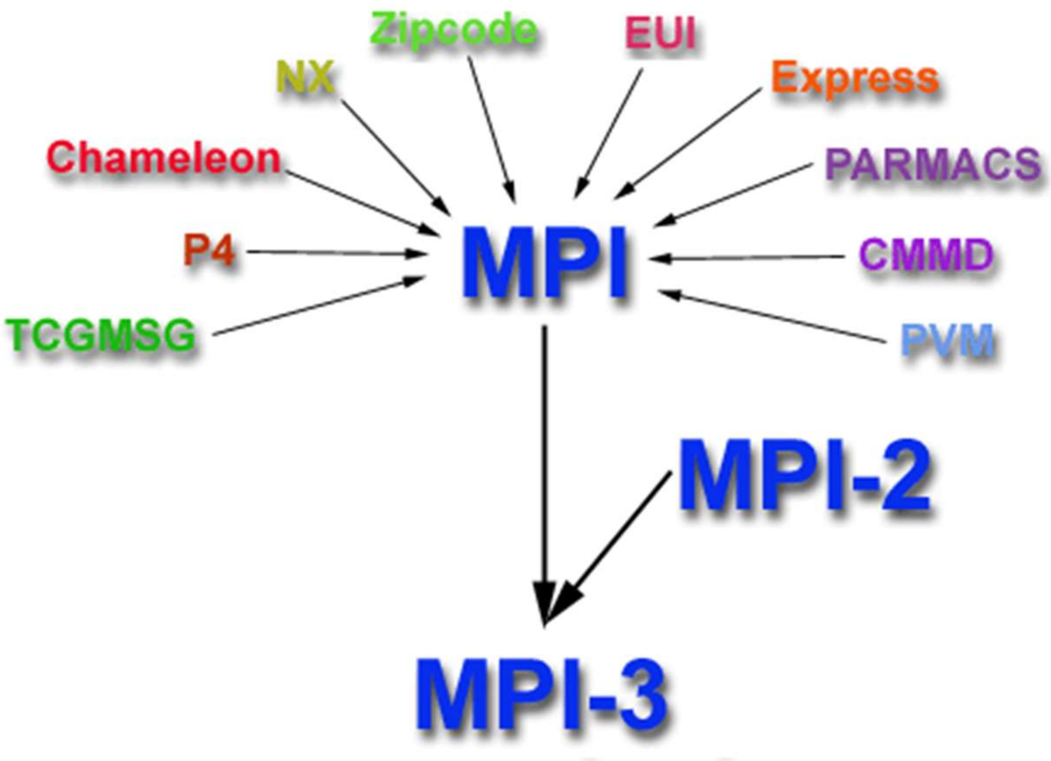
- MPI Standard documents: http://www.mpi-forum.org/docs/

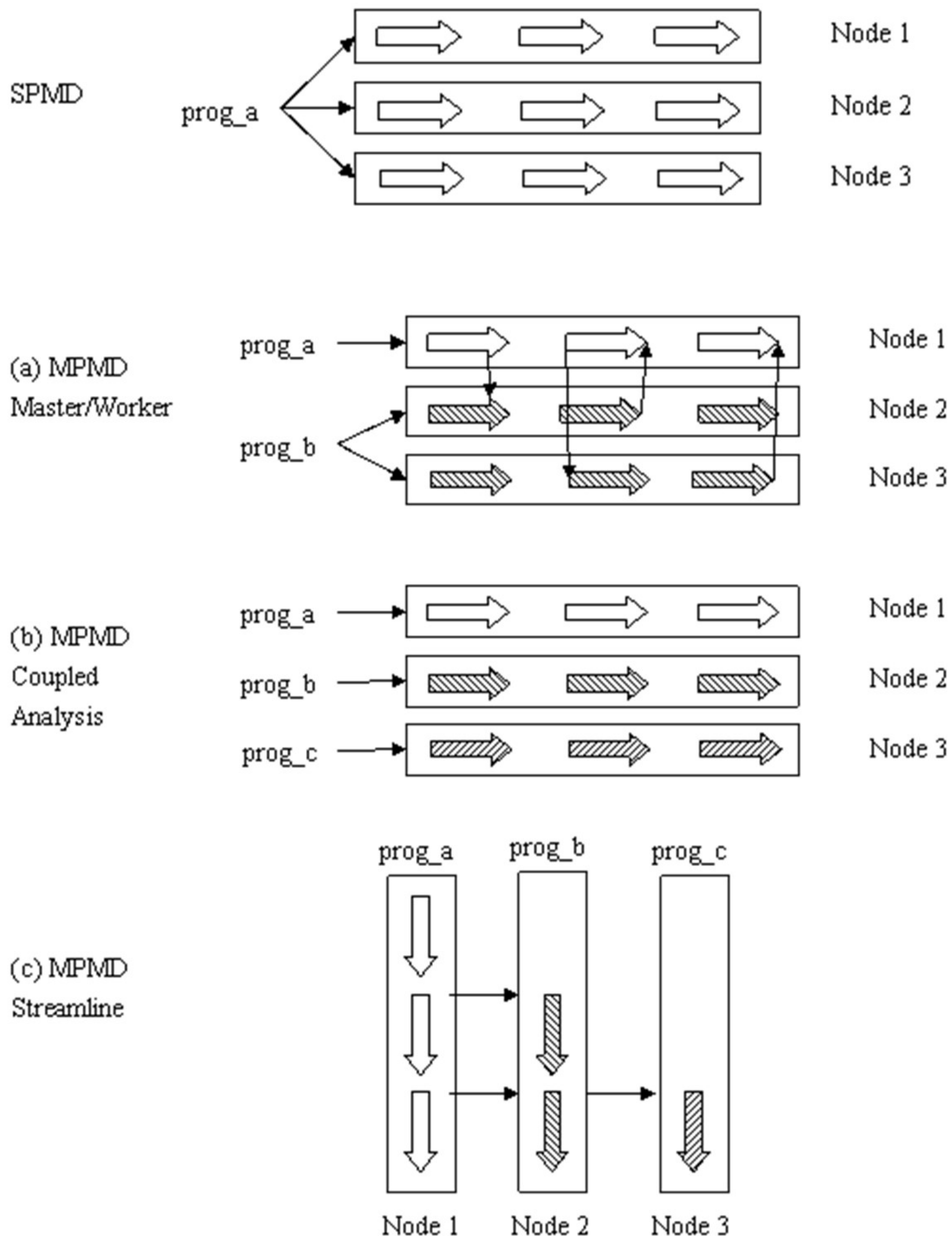
MPI-4.0

MPI-4.0 is a major update to the MPI standard. The largest changes are the addition of large-count versions of many routines to address the limitations of using an `int` or `INTEGER` for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

Documentation:

- Documentation for all versions of the MPI standard is available at: http://www.mpi-forum.org/docs/.




SPMD
prog_a
Node 1
Node 2
Node 3
(a) MPMD
Master/Worker
prog_a
prog_b
Node 1
Node 2
Node 3
(b) MPMD
Coupled
Analysis
prog_a
prog_b
prog_c
Node 1
Node 2
Node 3
(c) MPMD
Streamline
prog_a
prog_b
prog_c
Node 1
Node 2
Node 3

Why-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. Most MPI programs can be written using a dozen or less routines
- **Availability** - A variety of implementations are available, both vendor and public domain.

MPI Implementations and Compilers

- Although the MPI programming interface has been standardized, actual library implementations will differ in which version and features of the standard they support.
- The way MPI programs are compiled and run on different platforms may also vary.
  - **MVAPICH** - (https://mvapich.cse.ohio-state.edu)
  - **Open MPI -** (https://www.open-mpi.org)
  - **Mpich** - (https://www.mpich.org)
  - **Intel MPI**

MVAPICH

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

- MVAPICH2
  – based on MPI 3.1 standard, delivers the best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, Omni-Path, Ethernet/iWARP, RoCE(v1/v2), Cray Slingshot 10 and 11, and Rockport Networks networking technologies.

The MVAPICH2 software is powering several supercomputers in the TOP500 list. Examples (from the June '22 ranking) include:

- 6th, 10,649,600-core (Sunway TaihuLight) at National Supercomputing Center in Wuxi, China
- 16th, 448, 448 cores (Frontera) at TACC
- 30th, 288,288 cores (Lassen) at LLNL
- 42nd, 570,020 cores (Nurion) in South Korea
- 47th, 367,024 cores (Stampede2) at TACC

Open MPI

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

- Full MPI-3.1 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- Network and process fault tolerance
- Support network heterogeneity
- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- High performance on all platforms
- Portable and maintainable
- Tunable by installers and end-users
- Component-based design, documented APIs
- Active, responsive mailing list
- Open source license based on the BSD license

[Open MPI Announce] Open MPI 4.1.4 released

Barrett, Brian via announce | Thu, 26 May 2022 10:57:58 -0700

- **Members:** 14
- **Partners:** 4
- **Contributors:** 36

- **Individuals:** 16
- **Organizations:** 38

Intel MPI

- The Intel MPI Library is available as a standalone product and as part of the Intel® oneAPI HPC Toolkit
- It implements the Message Passing Interface, version 3.1 (MPI-3.1) specification.

  - Scalability up to 340k processes
  - Low overhead enables analysis of large amounts of data
  - MPI tuning utility for accelerating your applications
  - Interconnect independence and flexible runtime fabric selection

  - Supported Languages:
    - For GNU* compilers: C, C++, Fortran 77, Fortran 95
    - For Intel® compilers: C, C++, Fortran 77, Fortran 90, Fortran 95, Fortran 2008

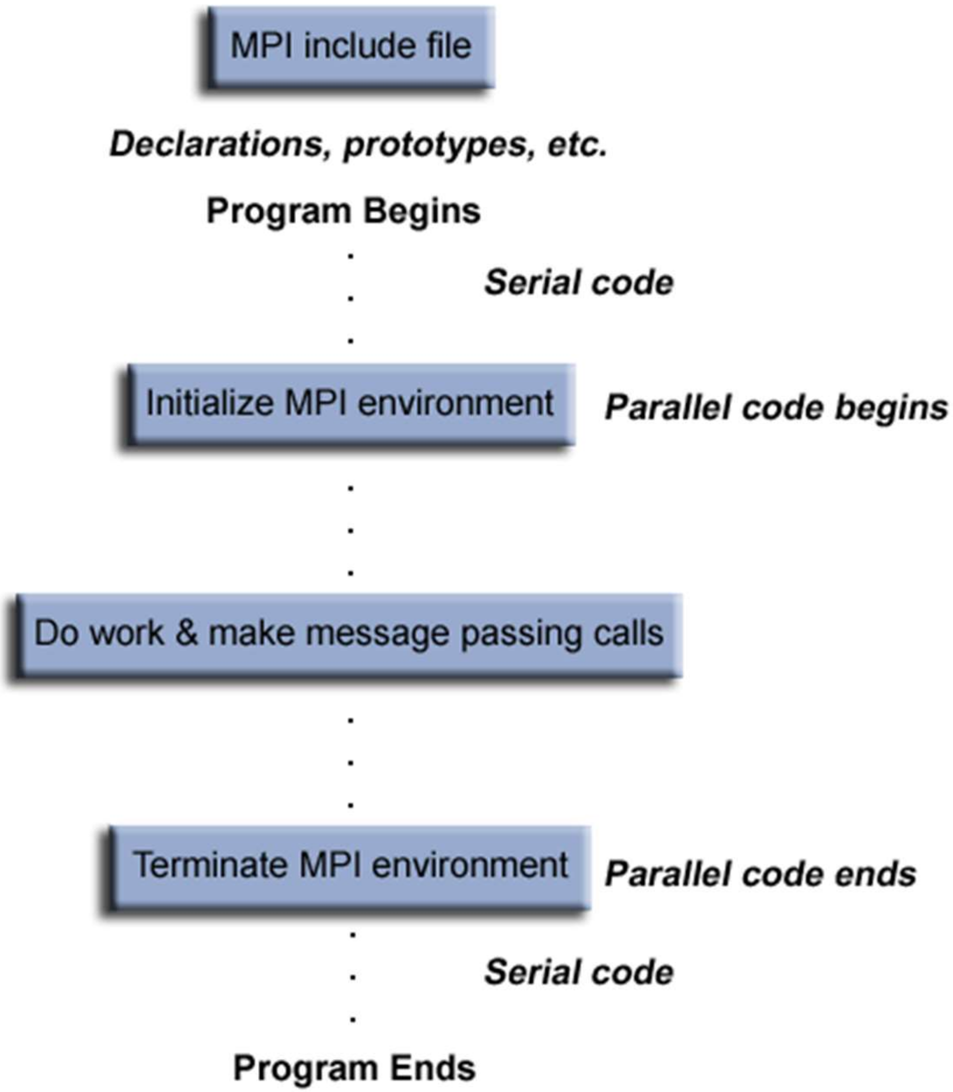
```
[houzx@c01b03 ~]$ ls /public/software/intel/impi/2019.4.243/intel64/bin/mpi
mpicc        mpiexec        mpif77   mpifc    mpigxx   mpiicpc    mpirun    mpivars.csh
mpicxx       mpiexec.hydra  mpif90   mpigcc   mpiicc   mpiifort   mpitune   mpivars.sh
```

MPI Build Scripts

- People usually developed MPI compiler wrapper scripts, which are used to compile MPI programs
- Automatically perform some error checks, include the appropriate MPI #include files, link to the necessary MPI libraries, and pass options to the underlying compiler.
- For additional information:
  - See the man page (if it exists)
  - Issue the script name with the -help option
  - View the script yourself directly

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.

General MPI Program Structure



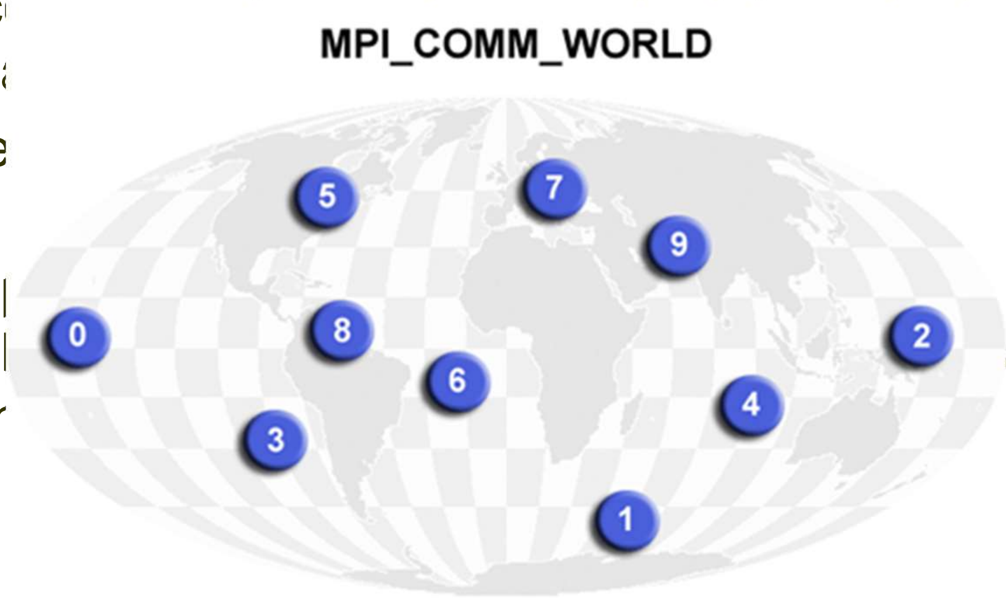
- **Header File:**
  - Required for all programs that make MPI library calls.

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 c
    collection of processes m
  - Most MPI routines require
    argument.
  - Communicators and grou
    now, simply use MPI_CO
    required - it is the predefir
    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).**

The **if-else** construct makes our program SPMD.

- **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. You can consult the error handling section of the relevant MPI Standard documentation located athttp://www.mpi-forum.org/docs/.
  - The types of errors displayed to the user are implementation dependent.

A Simple Framework - C

```
#include <mpi.h>
/* include other usual header files*/
main(int argc, char **argv)
{
    / * No MPI calls before this * /
    /* initialize MPI */
    MPI_Init(&argc, &argv);

    /* main part of program */

    /* terminate MPI */
    MPI_Finalize();
   / * No MPI calls after this */
    exit(0);
}
```

MPI helloworld.c

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

        printf("Hello World from process %d of %d\n",
            rank, numtasks);

        MPI_Finalize();
}
```

Compiling and Running MPI programs

Compile and Run Commands

- **Compile:**
  - [mpicc helloworld.c -o helloworld (standard)]

No. of processes

- **Run:**
  - mpirun -np 3 helloworld **[hosts picked from configuration file automatically]**
  - mpirun -np 3 -machinefile machines.list helloworld

- **The file machines.list contains nodes list:**
  - node1
    …
    node13
  - node1:8
    node2:8
    …

Sample Run and Output

- **A Run with 3 Processes:**
  - mpirun -np 3 -machinefile machines.list helloworld
    - Hello World from process 0 of 3
    - Hello World from process 1 of 3
    - Hello World from process 2 of 3
- **A Run by default**
  - ./helloworld
    - Hello World from process 0 of 1
- **You can also use mpirun to exec standard commands**
  - mpirun -np 4 -machinefile machines.list hostname

Sample Run and Output

- **A Run with 6 Processes:**
  - **manjra> mpirun -np 6 -machinefile machines.list helloworld**
    - Hello World from process 0 of 6
    - Hello World from process 3 of 6
    - Hello World from process 1 of 6
    - Hello World from process 5 of 6
    - Hello World from process 4 of 6
    - Hello World from process 2 of 6
- **Note: Process execution need not be in process number order.**

MPI Routines-C and Fortran

- *(1) Environment Management Routines*
- *(2) Point to Point Communication Routines*
- **(3) Collective Communication Routines**
- **(4) Derived Data Types**
- **(5) Process Group and Communicator Management Routines**
- **(6) Virtual Topologies**
- **Miscellaneous Routines**

The minimal set (6) of MPI routines

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

```
#include "mpi.h“
int main( int argc, char** argv )
{
int rank, size, tag=1;
int senddata,recvdata;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank==0){
senddata=9999;
MPI_Send( &senddata, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
if (rank==1)
MPI_Recv(&recvdata, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
MPI_Finalize();
return (0);
}
```

(1) Environment Management Routines

Environment Management Routines		
MPI_Abort	MPI_Errhandler_create	MPI_Errhandler_free
MPI_Errhandler_get	MPI_Errhandler_set	MPI_Error_class
MPI_Error_string	MPI_Finalize	MPI_Get_processor_name
MPI_Init	MPI_Initialized	MPI_Wtick
MPI_Wtime		

- ## 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_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)
```

Timers

- C: double MPI_Wtime(void)
  - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
- Time is measured in seconds
  - Time to perform a task is measured by consulting the time before and after

```
double MPI_Wtime(void);

double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

C Language - Environment Management Routines

```c
// 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();
}
```

Fortran - Environment Management Routines

```
program simple

! required MPI include file
include 'mpif.h'

integer numtasks, rank, len, ierr
character(MPI_MAX_PROCESSOR_NAME) hostname

! initialize MPI
call MPI_INIT(ierr)

! get number of tasks
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

! get my rank
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

! this one is obvious
call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
print *, 'Number of tasks=',numtasks,' My rank=',rank,' Running on=',hostname


     ! do some work with message passing


! done with MPI
call MPI_FINALIZE(ierr)

end
```

(2) Point to Point Communication Routines

- A simplest form of message passing
- One process sends a message to another
- Several variations on how sending a message can interact with execution of the sub-program

Point-to-Point Communication Routines		
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_attach
MPI_Buffer_detach	MPI_Cancel	MPI_Get_count
MPI_Get_elements	MPI_Ibsend	MPI_Iprobe
MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv
MPI_Recv_init	MPI_Request_free	MPI_Rsend
MPI_Rsend_init	MPI_Send	MPI_Send_init
MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend
MPI_Ssend_init	MPI_Start	MPI_Startall
MPI_Test	MPI_Test_cancelled	MPI_Testall
MPI_Testany	MPI_Testsome	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome

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:
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Synchronous send
  - Buffered send
  - Ready send
  - Combined send/receive
  - 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.

Point-to-Point variations

- **Blocking operations**
  - **only return from the call when operation has completed**

- **Non-blocking operations**
  - **return straight away - can test/wait later for completion**

- The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account.
- From a global viewpoint, it is reasonable to distinguish between synchronous and asynchronous communications.

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.

<table>
<tr><th>Blocking Send</th><th>Non-blocking Send</th></tr>
<tr><td>
<pre>
myvar = 0;

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

   /* do some work */

   }
</pre>
</td><td>
<pre>
myvar = 0;

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

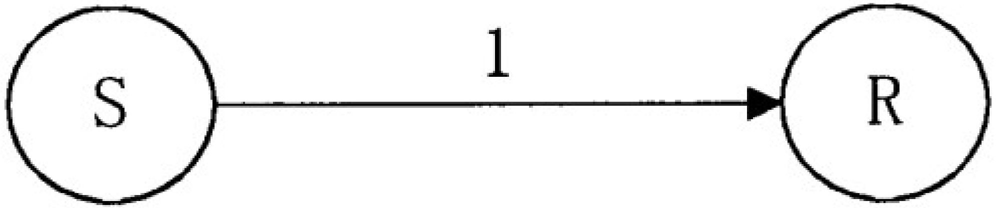
   /* do some work */

   MPI_Wait (...);
   }
</pre>
</td></tr>
<tr><td>Safe. Why?</td><td>Unsafe. Why?</td></tr>
</table>

Communication Modes for both blocking and non-blocking communication operations

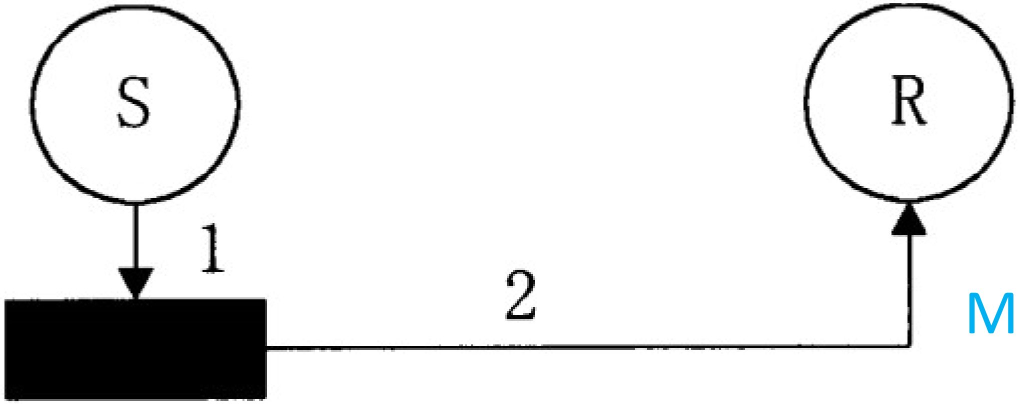
- Standard

  MPI_Send()



- Buffering

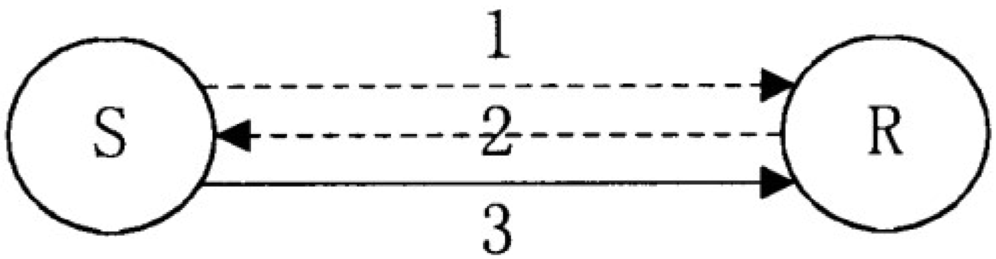
  MPI_Bsend()



MPI_Recv()

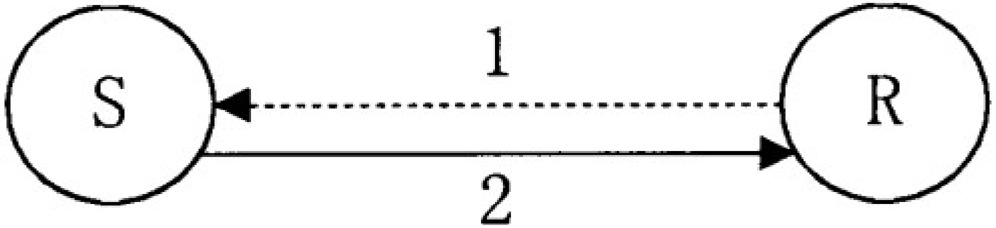
- Synchronous

  MPI_Ssend()



- Ready

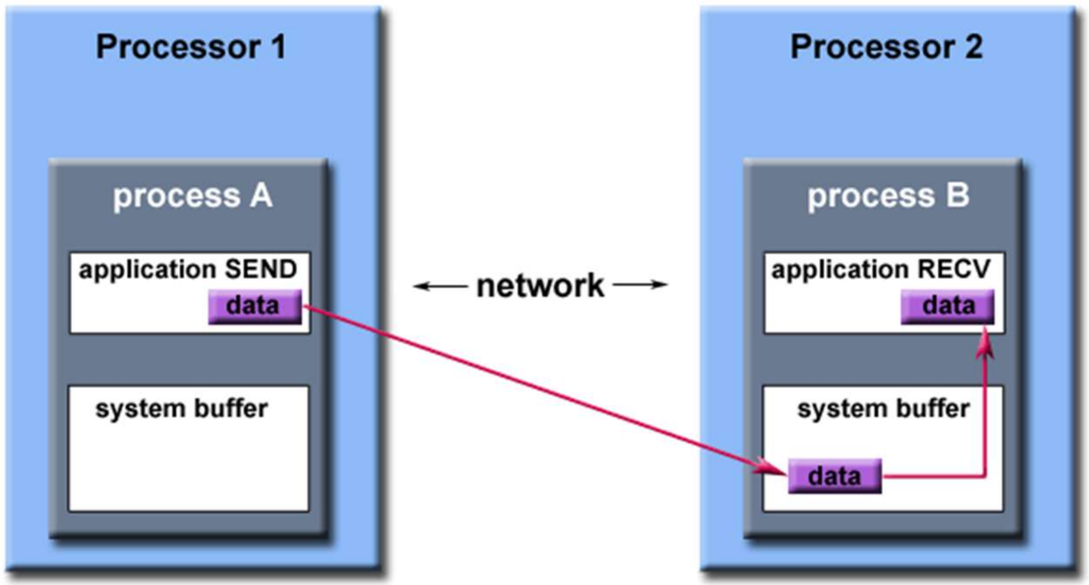
  MPI_Rsend()



Buffering:

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a **system buffer** area is reserved to hold data in transit. For example:

- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.
- User managed address space (i.e. your program variables) is called the **application buffer**. MPI also provides for a user managed send buffer.

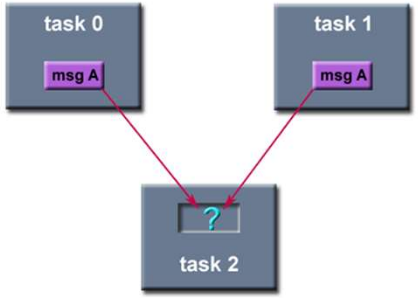


Path of a message buffered at the receiving process

Example of MPI_Bsend()

```
MPI_Comm comm=MPI_COMM_WORLD;
MPI_Init(&argc,&argv);
MPI_Comm(comm,&rank);
MPI_Pack_size(7,MPI_CHAR,comm,&s1);
MPI_Pack_size(2,MPI_DOUBLE,comm,&s2);
buffersize=2*MPI_BSEND_OVERHEAD+s1+s2;
buf=(char*)malloc(buffersize);
MPI_Buffer_attach(buf,buffersize);
if(rank==src){
MPI_Bsend(msg1,7,MPI_CHAR,dest,tag,comm);
MPI_Bsend(msg2,2,MPI_DOUBLE,dest,tag,comm);
}
if(rank==dest){
        MPI_Recv(rmsg1,7,MPI_CHAR,src,tag,comm,MPI_STATUS_IGNORE);
        MPI_Recv(rmsg2,2,MPI_DOUBLE,src,tag,comm,MPI_STATUS_IGNORE);
}
MPI_Buffer_detach(&buf,&buffersize);
free(buf);
MPI_Finalize();
return 0;
}
```

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 primitive	**blocking**	**non-blocking**
Standard Send	*MPI_Send*	*MPI_Isend*
Synchronous Send	*MPI_ Ssend*	*MPI_ Issend*
Buffered Send	*MPI_ Bsend*	*MPI_ Ibsend*
Ready Send	*MPI_ Rsend*	*MPI_ Irsend*
Receive	MPI_Recv	MPI_Irecv
Completion Check	MPI_Wait	MPI_Test

	Point-to-point	Collective

Blocking	`MPI_Send()`
	`MPI_Ssend()`
	`MPI_Bsend()`
	`MPI_Recv()`
		`MPI_Barrier()`
		`MPI_Bcast()`
		`MPI_Scatter()/`
		`MPI_Gather()`
		`MPI_Reduce()`
		`MPI_Reduce_scatter()`
		`MPI_Allreduce()`

Nonblocking	`MPI_Isend()`
	`MPI_Irecv()`
	`MPI_Wait()/MPI_Test()`
	`MPI_Waitany()/`
	`MPI_Testany()`
	`MPI_Waitsome()/`
	`MPI_Testsome()`
	`MPI_Waitall()/`
	`MPI_Testall()`
		N/A

- MPI’s communication modes and a nonexhaustive overview of the corresponding subroutines.

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.

- **Data Type**
  - For reasons of portability, MPI predefines its elementary data types. The table below lists those required by the standard.

C Data Types		Fortran Data Types	
MPI_CHAR	char	MPI_CHARACTER	character(1)
MPI_WCHAR	wchar_t - wide character		
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4	integer integer*1 integer*2 integer*4
MPI_LONG	signed long int		
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int		
MPI_SIGNED_CHAR	signed char		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_UNSIGNED_LONG_LONG	unsigned long long int		
MPI_FLOAT	float	MPI_REAL MPI_REAL2 MPI_REAL4 MPI_REAL8	real real*2 real*4 real*8

MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	float _Complex	MPI_COMPLEX	complex
MPI_C_DOUBLE_COMPLEX	double _Complex	MPI_DOUBLE_COMPLEX	double complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex		
MPI_C_BOOL	_Bool	MPI_LOGICAL	logical
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	int8_t int16_t int32_t int64_t		
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	uint8_t uint16_t uint32_t uint64_t		
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

- **Notes:**
  - Programmers may also create their own data types (see Derived Data Types).
  - MPI_BYTE and MPI_PACKED do not correspond to standard C or Fortran types.
  - Types shown in GRAY FONT are recommended if possible.
  - Some implementations may include additional elementary data types (MPI_LOGICAL2, MPI_COMPLEX32, etc.). Check the MPI header file.

- **Destination**
  - An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.
- **Source**
  - An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.
- **Tag**
  - Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

- **Communicator**
  - Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.
- **Status**
  - For a receive operation, **indicates the source of the message and the tag of the message, MPI_error**. 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 queried later.

- **Status is useful when one receiver receives different messages with different size and different tag from some processes.**
- **e.g. if multiple client processes send messages to the server process, the server process adopts different service according to the tags.**

```
while (true){
 MPI_Recv(received_request,100,MPI_BYTE,MPI_Any_source,MPI_An
    y_tag,comm,&Status);
 switch (Status.MPI_Tag) {
 case tag_0: perform service type0;
 case tag_1: perform service type1;
 case tag_2: perform service type2;
 }
}
```

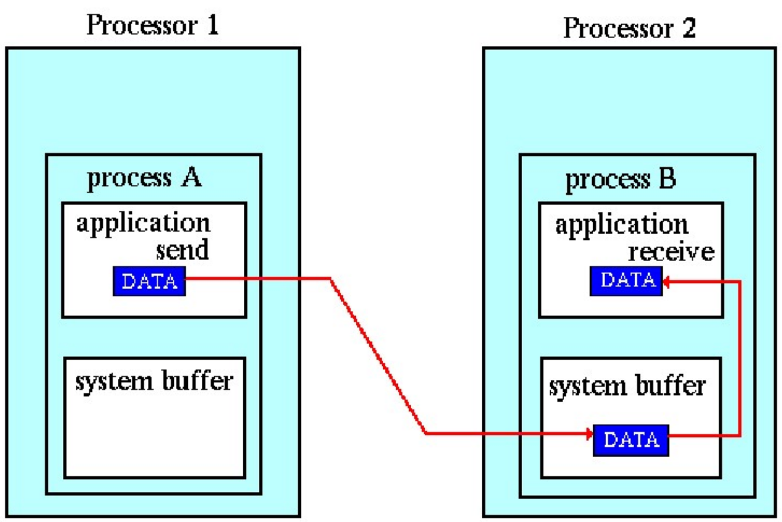
- **Request**
  - Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine 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.

Message

- Messages are packets of data moving between sub-programs.

- Messge buffer (content)
  (buf, count, datatype)

- Message envelope
  (dest, tag, comm)



Path of a message buffered at the receiving process

Common Blocking Message Passing Routines

- **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)
```

```fortran
<type> buf(*)
integer :: count, datatype, dest, tag, comm, ierror
call MPI_Send(buf,        ! message buffer
              count,      ! # of items
              datatype,   ! MPI data type
              dest,       ! destination rank
              tag,        ! message tag (additional label)
              comm,       ! communicator
              ierror)     ! return value
```

```c
int MPI_Send(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

```
<type> buf(*)
integer :: count, datatype, source, tag, comm,
integer :: status(MPI_STATUS_SIZE), ierror
call MPI_Recv(buf,        ! message buffer
              count,      ! maximum # of items
              datatype,   ! MPI data type
              source,     ! source rank
              tag,        ! message tag (additional label)
              comm,       ! communicator
              status,     ! status object (MPI_Status* in C)
              ierror)     ! return value
```

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```

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)
```

Suppose process q calls `MPI_Send` with

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Also suppose that process r calls `MPI_Recv` with

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Then the message sent by q with the above call to `MPI_Send` can be received by r with the call to `MPI_Recv` if

- `recv_comm = send_comm`,
- `recv_tag = send_tag`,
- `dest = r`, and
- `src = q`.

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send
src = q

MPI_Recv
dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

- **Are these conditions enough for the message to be *successfully* received?**

If `recv_type` = `send_type` and `recv_buf_sz` ≥ `send_buf_sz`, then the message sent by q can be successfully received by r.

Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

MPI_SOURCE
MPI_TAG
MPI_ERROR

```
MPI_Status*  status;

status.MPI_SOURCE
status.MPI_TAG
```

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)
```

Examples: Blocking Message Passing Routines

C Language - Blocking Message Passing Example

```c
#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 recieve 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();
}
```

Fortran - Blocking Message Passing Example

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

integer numtasks, rank, dest, source, count, tag, ierr
integer stat(MPI_STATUS_SIZE)   ! required variable for receive routines
character inmsg, outmsg
outmsg = 'x'
tag = 1

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

! task 0 sends to task 1 and waits to receive a return message
if (rank .eq. 0) then
   dest = 1
   source = 1
   call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD, ierr)
   call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag, MPI_COMM_WORLD, stat, ierr)

! task 1 waits for task 0 message then returns a message
else if (rank .eq. 1) then
   dest = 0
   source = 0
   call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag, MPI_COMM_WORLD, stat, err)
   call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD, err)
endif

! query recieve Stat variable and print message details
call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
print *, 'Task ',rank,': Received', count, 'char(s) from task', &
         stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)

call MPI_FINALIZE(ierr)

end
```

Non-blocking Message Passing Routines

- **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)
```

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, request, ierror
call MPI_Isend(buf,          ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               dest,         ! destination rank
               tag,          ! message tag
               comm,         ! communicator
               request,      ! request handle (MPI_Request* in C)
               ierror)       ! return value
```

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)
```

```
<type> buf(*)
integer :: count, datatype, source, tag, comm, request, ierror
call MPI_Irecv(buf,          ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               source,       ! source rank
               tag,          ! message tag
               comm,         ! communicator
               request,      ! request handle
               ierror)       ! return value
```

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)
```

```
while (Not_Done){
   if (X==Xbuf0)
         {X=Xbuf1; Y=Ybuf1; Xin=Xbuf0; Yout=Ybuf0;}
   else {X=Xbuf0; Y=Ybuf0; Xin=Xbuf1; Yout=Ybuf1;}

   MPI_Irevc(Xin, …, recv_handle);
   MPI_Isend(Yout, …, send_handle);

   Y=Q(X);            /* overlap comp. and comm.*/

   MPI_Wait(recv_handle,recv_status);
   MPI_Wait(send_handle,send_status);
}
```

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.

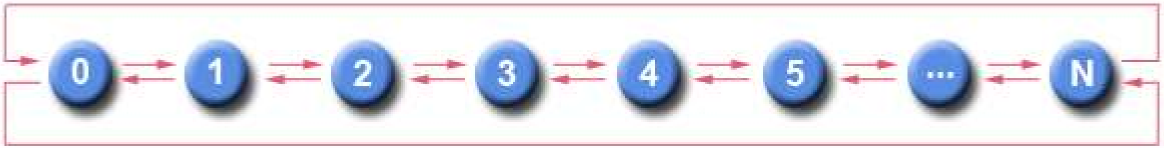
```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISSEND (buf,count,datatype,dest,tag,comm,request,ierr)
```

- **MPI_Test** 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. MPI_test will not be blocked. 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)
```

Examples: Non-blocking Message Passing Routines

Nearest neighbor exchange in a ring topology



C Language - Non-blocking Message Passing Example

```c
#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();
}
```

Fortran - Non-blocking Message Passing Example

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

integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr
integer reqs(4)   ! required variable for non-blocking calls
integer stats(MPI_STATUS_SIZE,4)   ! required variable for WAITALL routine
tag1 = 1
tag2 = 2

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

! determine left and right neighbors
prev = rank - 1
next = rank + 1
if (rank .eq. 0) then
   prev = numtasks - 1
endif
if (rank .eq. numtasks - 1) then
   next = 0
endif

! post non-blocking receives and sends for neighbors
call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1, MPI_COMM_WORLD, reqs(1), ierr)
call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2, MPI_COMM_WORLD, reqs(2), ierr)

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

   ! do some work while sends/receives progress in background

! wait for all non-blocking operations to complete
call MPI_WAITALL(4, reqs, stats, ierr);

   ! continue - do more work

call MPI_FINALIZE(ierr)

end
```

Example1: MPI Send/Receive a Character

```
// mpi_com.c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, tag=1;
char inmsg, outmsg='X';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Rank0 sent: %c\n", outmsg);
source = 1;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
```

Example1: MPI Send/Receive a Character(cont.)

```
else if (rank == 1) {
source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
     &Stat);
  printf("Rank1 received: %c\n", inmsg);
 dest = 0;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Execution Demo

- mpicc –o mpi_com mpi_com.c
- [raj@manjra mpi]$ mpirun -np 2 mpi_com

  Rank0 sent: X

  Rank1 received: X

Example 2: Ping Pong

1. Write a program in which two processes repeatedly pass a message back and forth.
2. Insert timing calls to measure the time taken for one message.
3. Investigate how the time taken to exchange messages varies with the size of the message.

Example 2: A simple Ping Pong.c (cont.)

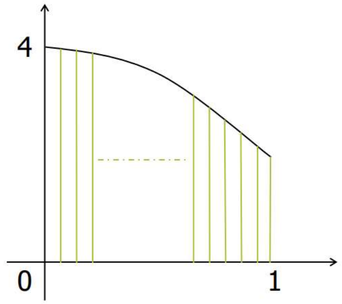
```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, tag=1;
char inmsg, outmsg='X';
char pingmsg[10]; char pongmsg[10]; char buff[100];
MPI_Status Stat;

strcpy(pingmsg, "ping");
strcpy(pongmsg, "pong");

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

if (rank == 0) {   /* Send Ping, Receive Pong */
  dest = 1;
  source = 1;
  rc = MPI_Send(pingmsg, strlen(pingmsg)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(buff, strlen(pongmsg)+1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
   printf("Rank0 Sent: %s & Received: %s\n", pingmsg, buff);
  }
```

Why + 1 ?

Example 2: A simple Ping Pong.c (cont..)

```
else if (rank == 1) { /* Receive Ping, Send Pong */
  dest = 0;
  source = 0;
  rc = MPI_Recv(buff, strlen(pingmsg)+1, MPI_CHAR, source, tag,
     MPI_COMM_WORLD, &Stat);
  printf("Rank1 received: %s & Sending: %s\n", buff, pongmsg);
  rc = MPI_Send(pongmsg, strlen(pongmsg)+1, MPI_CHAR, dest, tag,
     MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Example 3.1: Calculation of π



$$f(x) = 4/(1+x^2)$$

$$\int_0^1 f(x)dx = \pi$$

$$\pi \approx \sum_{i=1}^{n} f\left(\frac{2\times i-1}{2\times N}\right)\times\frac{1}{N}$$

$$= \frac{1}{N}\times\sum_{i=1}^{n} f\left(\frac{i-0.5}{N}\right)$$

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
double f(double x);/* Definition of f(x) */
{
    return(4.0/(1.0+x*x));
}
int main (int argc,char * argv[])
{
     int done =0,n,myid,numprocs,i;
     double PI25DT=3.141592653589793238462643;
     double mypi,pi,h,sum,x;
     double startwtime=0.0,endwtime;
     int namelen;
     char processor_name[MPI_MAXPROCESSOR_NAME];
     MPI_Status status;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);
fprint(stdout,”Process %d of %d on % s\n”,myid,numprocs,
   processor_name);
n=0;
if (myid==0)
   {
       printf(“Please give N=”);
       scanf(&n);
       startwtime=MPI_Wtime();
       for (j=1;j<numprocs;j++)
      {
          MPI_Send(&n,1,MPI_INT,j,99,MPI_COMM_WORLD);
       }
   }
```

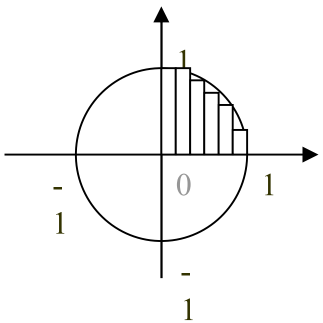
```
 else
{
    MPI_Recv(&n,1,MPI_INT,MPI_ANY_SOURCE,99,MPI_COMM_
    WORLD,&status);
}
      h=1.0/(double) n;
      sum=0.0;
      for(i=myid+1;i<=n;i+=numprocs)
/*  Each process computes some part of the rectangle, e.g. if
    numprocs is 4, 0-1 area is divided by 100 rectangles, then the 4
    processes compute the rectangles respectively as follows:
      P0   1, 5, 9, 13, ……, 97
      P1   2, 6, 10, 14, ……, 98
      P2   3, 7, 11, 15, ……, 99
      P3   4, 8, 12, 16, ……, 100   */
```

```
        {
            x=h*((double)i-0.5);
            sum+=f(x);
        }
        mypi=h*sum; /* part of sum for each process*/

/* Accumulate all the part of sum to obtain the area of all the
rectangles, this area is the approximate value of PI.*/
     if (myid != 0)
    MPI_Send(&mypi,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD) ;
    else
    {
        pi=0.0;
        pi=pi+mypi;
```

```
            for (j=1;j<numprocs;j++)
            {
             MPI_Recv(&mypi,1,MPI_DOUBLE,MPI_ANY_SOURCE,
                         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
              pi=pi+mypi;
             }

            printf("pi is approximately %.16f,Error is %.16f\n",
                     pi,fabs(pi-PI25DT));
            endwtime=MPI_Wtime();
            printf("wall clock time=% f\n",endwtime-startwtime);
            fflush(stdout);
        }
      MPI_Finalize();
}
```

Example 3.2: Calculation of π

$$\frac{1}{4}\pi \approx \sum_{x=0}^{N-1} \frac{1}{N}\sqrt{1-\left(\frac{x}{N}\right)^2}$$




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

const int N = 1000000;
double start,finish;
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int numprocs, myid;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
      start = MPI_Wtime();
    double partSum = 0.0;
    double pi = 0.0;
    for (int i = myid; i < N; i += numprocs)
        {
            partSum += sqrt(1 - (double(i) / N)*(double(i) / N)) / N;
        }
    MPI_Reduce(&partSum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    cout << "Myid" << myid << ", partSum:" << partSum << endl;
    if (myid == 0)
    {
     pi *= 4.0;
     finish = MPI_Wtime();
     cout << " The value of pi:" << pi << endl;
     cout<<"#"<<numprocs<<"run time:"<< finish - start<< endl;
    }

    MPI_Finalize();
    return 0;
}
```

Thank you!