23rd Summer School on PARALLEL COMPUTING

Parallel programming with MPI Part I Introduction and Point-to-Point Communications

Paolo Ramieri - p.ramieri@cineca.it
SuperComputing Applications and Innovation Department

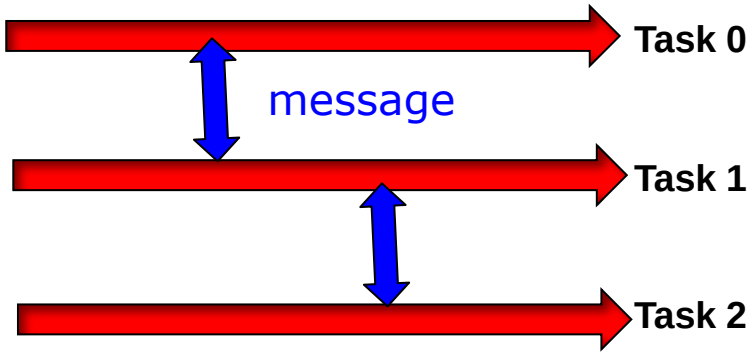
CINECA

Contents

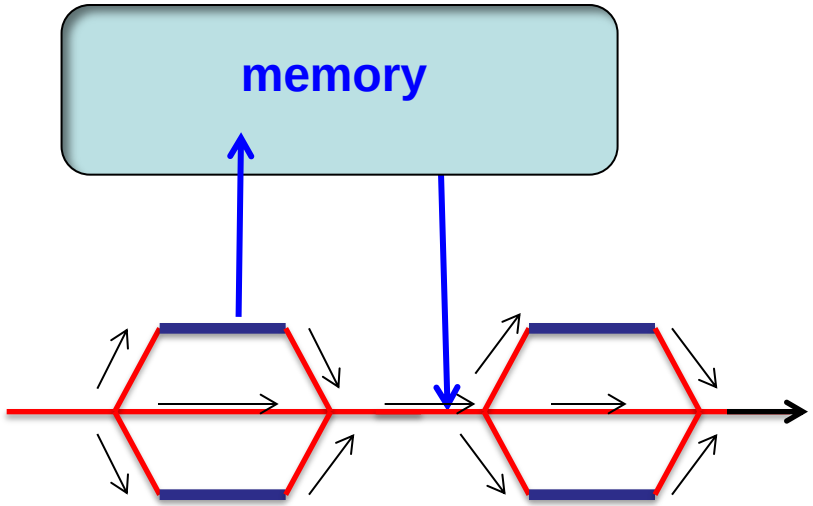
- Introduction to message passing and MPI
- Basic MPI programs
- MPI Communicators
- Send and Receive function calls for point-to-point communications
- Blocking and non-blocking
- How to avoid deadlocks

Message passing and shared memory parallelism

message passing



shared memory



Multiple *tasks* exchange data via explicit messages

Program splits into *threads* which share data via variables in shared memory

Message Passing

- Unlike the shared memory model, resources are local;
- Each process operates in its own environment (logical address space) and communication occurs via the exchange of messages;
- Messages can be instructions, data or synchronisation signals;
- The message passing scheme can also be implemented on shared memory architectures;
- Delays are much longer than those due to shared variables in the same memory space;

Advantages and Drawbacks

- Advantages
  - Communications hardware and software are important components of HPC system and often very highly optimised;
  - Portable and scalable;
  - Long history (many applications already ready written for it);
- Drawbacks
  - Explicit nature of message-passing is error-prone and discourages frequent communications;
  - Most serial programs need to be completely re-written;
  - High memory overheads.

Message Passing – Data transfer and Synchronisation.

The sender process cooperates with the destination process

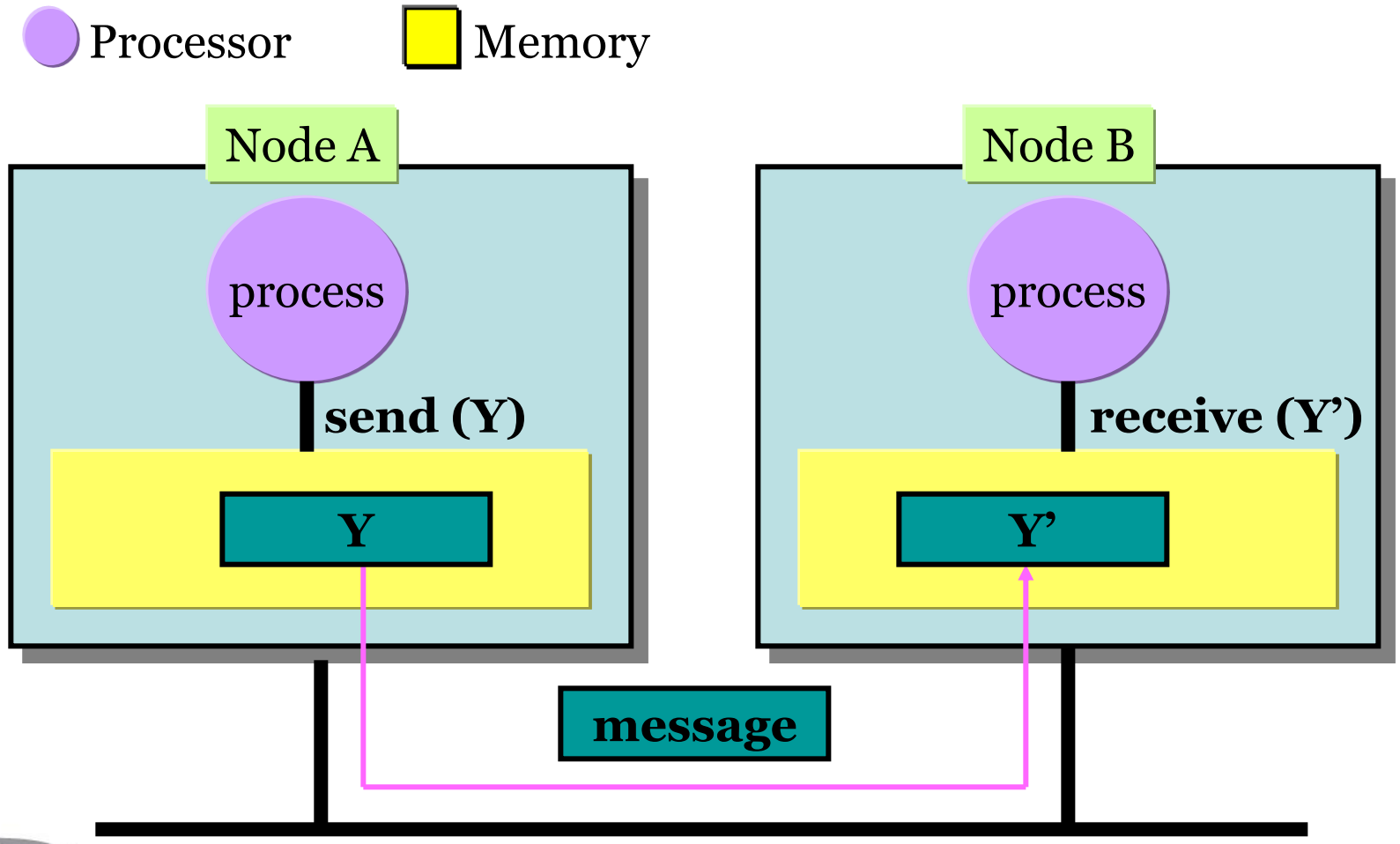
The communication system must allow the following three operations:

send*(message)*

receive*(message)*

synchronisation

MPI Programming Model



The Message Passing Interface - MPI

-MPI is a standard defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org/

-In particular the MPI documents define the APIs (application interfaces) for C, C++, FORTRAN77 and FORTRAN90.

-The actual implementation of the standard is left to the software developers of the different systems

-In all systems MPI has been implemented as a library of subroutines over the network with drivers and primitives

Goals of the MPI standard

MPI's prime goals are:

- To allow efficient implementation
- To provide source-code portability

MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

MPI2 further extends the library power (parallel I/O, Remote Memory Access, Multi Threads, Object Oriented programming)

MPI3 aims to support exascale by including non-blocking collectives, improved RMA and fault tolerance.

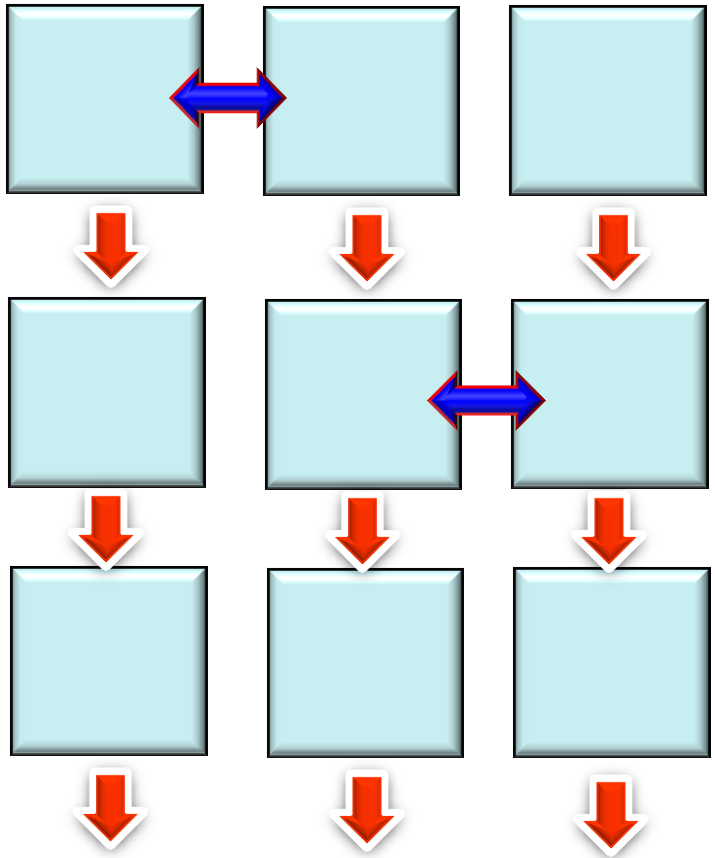
Basic Features of MPI Programs

An MPI program consists of multiple instances of a serial program that communicate by library calls.

Calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and terminate communications
2. Calls used to communicate between pairs of processors. (point to point communication)
3. Calls used to communicate among groups of processors. (collective communication)
4. Calls to create data types.

Single Program Multiple Data (SPMD) programming model



Multiple instances of the **same** program.

A note about MPI Implementations

- The MPI standard defines the functionalities and the API, i.e. what the C or FORTRAN calls should look like.
- The MPI standard **does not define how the calls should be performed at the system level** (algorithms, buffers, etc) or how the environment is set up (env variables, mpirun or mpiexec, libraries, etc). This is left to the **implementation**.
- There are various implementations (IntelMPI, OpenMPI, MPICH, HPMPI, etc) which have different performances, features and standards compliance.
- On some clusters (e.g. PLX, Eurora) you may choose which MPI to use, on other systems you have only the vendor-supplied version (IBM MPI for FERMI).

Compiling and Running MPI programs

- Implementation and system dependent but it is usual to use the “wrapped” version of the compiler to include the MPI headers and link in the MPI libraries. Wrapped compilers tend to be called mpif90, mpicc, mpic++, etc.
- On HPC systems MPI programs are run via the batch system with appropriate settings. For debugging sometimes it is possible to open interactive sessions (e.g. PBS on PLX).
- a program such as **`mpirun`** or **`mpiexec`** is then used to launch multiple instances of the program on the assigned nodes.

Compiling and running MPI on FERMI and PLX/EURORA

FERMI

Compile + link

```
module load  bgq-xl
mpxlf90 -o mpi_prog
mpi_prog.f90
```

Job script

```
....
#@ wall_clock_limit = 01:00:00
#@ bg_size=64
#@ queue
runjob --ranks-per-node 16 -np
1024 : $PWD/mpi_myprog
```

PLX/EURORA

Compile+link

```
module load autoload openmpi
mpicc -o mpi_prog mpi_prog.c
```

Job script

```
#PBS -l
select=1:ncpus=12:mpiprocs=12,w
alltime=1:00:00
cd $PBS_O_WORKDIR
module load autoload openmpi
mpirun -np 12 ./mpi_prog
```

A First Program: Hello World!

Fortran

```
PROGRAM hello

  INCLUDE 'mpif.h'
  INTEGER err

  CALL MPI_INIT(err)
  PRINT *, "hello world!"
  CALL MPI_FINALIZE(err)

END
```

C

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

void main (int argc, char * argv[] )
        {
          int err;

          err = MPI_Init(&argc, &argv);
          printf("Hello world!\n");
          err = MPI_Finalize();
        }
```

Header files

All Subprogram that contains calls to MPI subroutine must include the MPI header file

C:

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

Fortran:

```
include 'mpif.h'
```

Fortran 90:

```
USE MPI
```

FORTRAN note:
The FORTRAN include and module forms are *not equivalent*: the module can also do type checking BUT since the MPI standard is not consistent with FORTRAN some F90 compilers give errors. Many FORTRAN codes prefer to use the include file.

The header file contains definitions of MPI constants, MPI types and functions

MPI function format

C:

```
int error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

FORTRAN:

```
CALL MPI_XXXXX(parameter, IERROR)
INTEGER IERROR
```

Initializing MPI

C:

```
int MPI_Init(int*argc, char***argv)
```

FORTRAN:

```
INTEGER IERROR
MPI_INIT(IERROR)
```

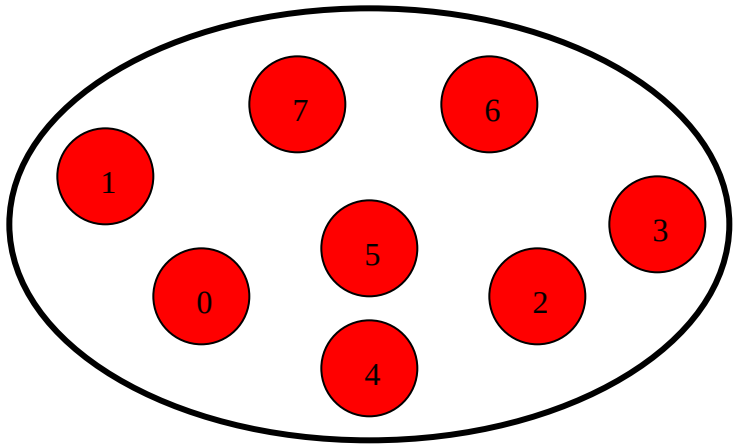
Must be first MPI call: initializes the message passing routines

MPI Communicator

- In MPI it is possible to divide the total number of processes into groups, called *communicators*.
- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.
- The communicator that includes all processes is called MPI_COMM_WORLD
- MPI_COMM_WORLD is the default communicator (automatically defined):

All MPI communication subroutines have a communicator argument.

The Programmer can define many communicators at the same time



MPI_COMM_WORLD

Communicator Size

How many processors are associated with a communicator?

C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

FORTRAN:

```
INTEGER COMM, SIZE, IERR
OUTPUT:  SIZE
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

Process Rank

How can you identify different processes?
What is the ID of a processor in a group?

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

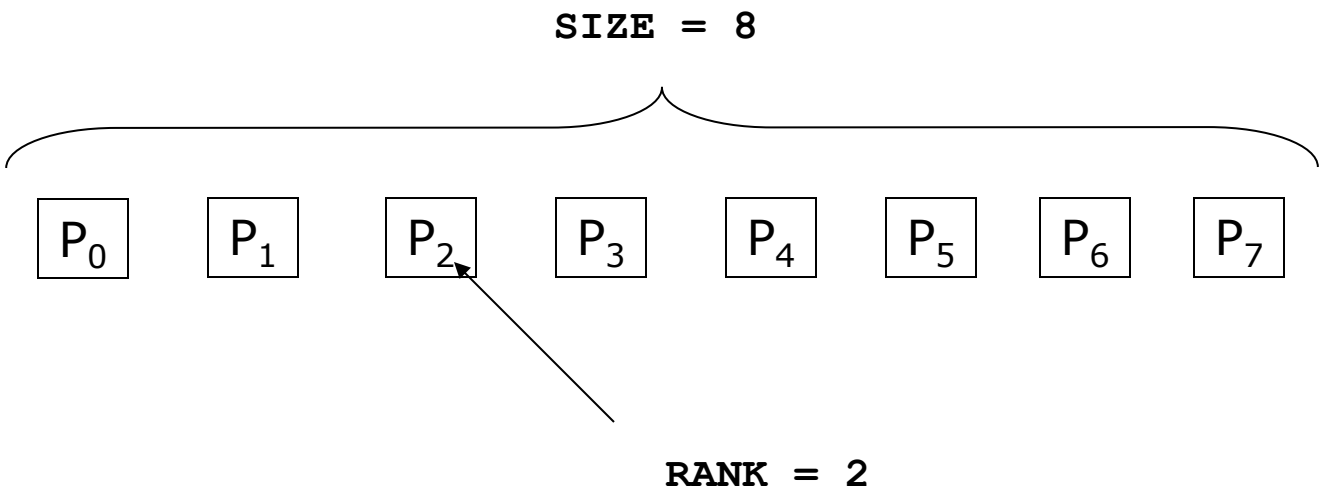
```
CALL MPI_COMM_RANK(COMM, RANK, IERR)
INTEGER COMM, RANK, IERR
OUTPUT:  RANK
```

rank is an integer that identifies the Process inside the communicator *comm*

MPI_COMM_RANK is used to find the rank (the name or identifier) of the Process running the code

Communicator Size and Process Rank / 1

How many processes are contained within a communicator?



`Size` is the number of processors associated to the communicator

`rank` is the index of the process within a group associated to a communicator (**`rank`** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication

Exiting MPI

Finalizing MPI environment

C:

```
int MPI_Finalize()
```

Fortran:

```
INTEGER IERR
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all process, and no other MPI calls are allowed before **`mpi_init`** and after **`mpi_finalize`**. However the program can go on as a serial program

MPI_ABORT

- Usage
  - int MPI_Abort(MPI_Comm comm,
    int errorcode);
- Description
  - Terminates all MPI processes associated with the communicator comm; in most systems (all to date), terminates *all* processes.

A Template for Fortran MPI Programs

```
PROGRAM template

INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      !!! INSERT YOUR PARALLEL CODE HERE !!!

CALL MPI_FINALIZE(ierr)

END
```

A Template for C MPI programs

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /*** INSERT YOUR PARALLEL CODE HERE ***/

  err = MPI_Finalize();
}
```

Example

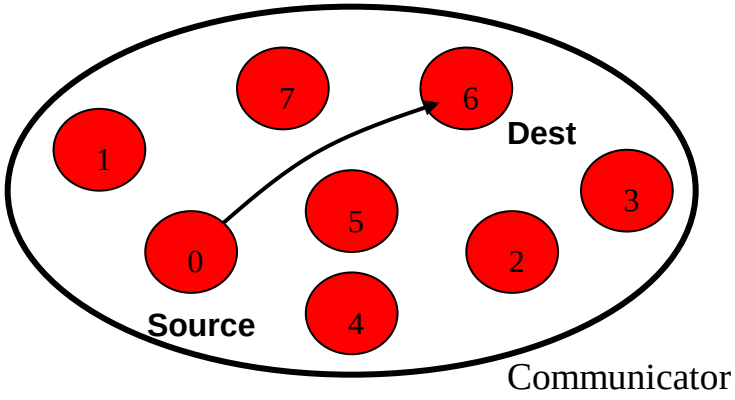
```
PROGRAM hello
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER:: myPE, totPEs, i, ierr

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
CALL MPI_COMM_SIZE( MPI_COMM_WORLD, totPEs, ierr )
PRINT *, "myPE is ", myPE, "of total ", totPEs, " PEs"
CALL MPI_FINALIZE(ierr)
END PROGRAM hello
```

Output (4 Procs)

```
MyPE is 1 of total 4 PEs
MyPE is 0 of total 4 PEs
MyPE is 3 of total 4 PEs
MyPE is 2 of total 4 PEs
```

Point-to-Point Communication

- It is the basic communication method provided by MPI library. Communication between 2 processes
- It is conceptually simple: source process A sends a message to destination process B, B receive the message from A.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator



Point-to-Point communication –quick example

```
…….
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
   CALL MPI_SEND(a, 2, MPI_REAL, 1, 10,
   MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
   CALL MPI_RECV(b, 2, MPI_REAL, 0, 10,
   MPI_COMM_WORLD, status, ierr)
END IF

…
```

Point-to-Point communication –quick example

The construction

```
if rank equals i
   send information
else if rank equals j
   receive information
```

is very common in MPI programs. Often one rank (usually rank 0) is selected for particular tasks which can be or should be done by one task only such as reading or writing files, giving messages to the user or for managing the overall logic of the program (e.g. master-slave).

The Message

- Data is exchanged in the buffer, an array of count elements of some particular MPI data type
- One argument that usually must be given to MPI routines is the *type* of the data being passed.
- This allows MPI programs to run automatically in heterogeneous environments
- C types are different from Fortran types.

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

Data Types

- MPI Data types
  - Basic types (portability)
  - Derived types (MPI_Type_xxx functions)
- Derived type can be built up from basic types
- User-defined data types allows MPI to automatically scatter and gather data to and from non-contiguous buffers

MPI defines '*handles*' to allow programmers to refer to data types and structures

- **C/C++** handles are macro to structs (#define MPI_INT ...)
- **Fortran** handles are INTEGER

Fortran - MPI Intrinsic Datatypes

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	
MPI_BYTE	

C - MPI Intrinsic Datatypes

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

For a communication to succeed ..

1. Sender must specify a valid destination rank.
2. Receiver must specify a valid source rank.
3. The communicator must be the same.
4. Tags must match.
5. Buffers must be large enough.

Completion

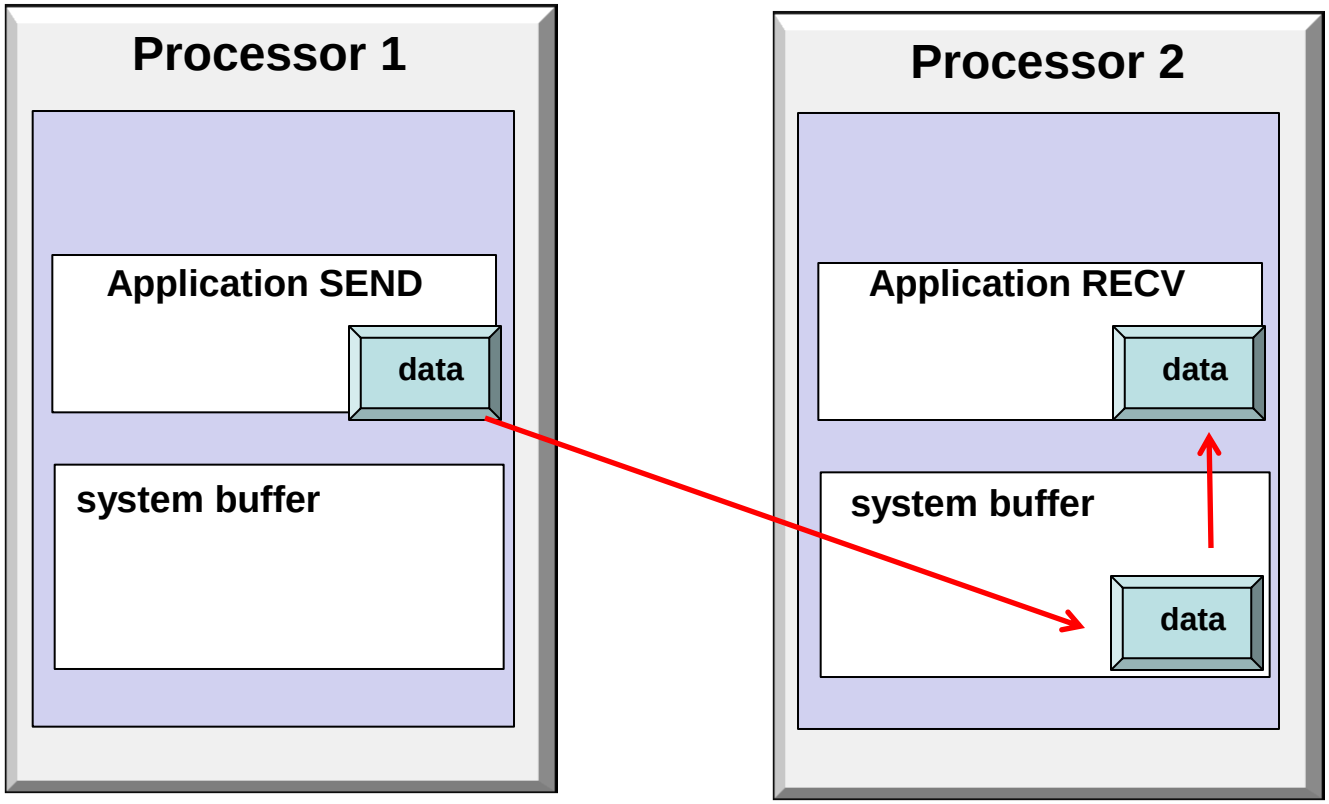
- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. The MPI implementation is able to deal with storing data when the two tasks are out of sync.

- **Completion** of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can be used after completion

Blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking:
  - A blocking send returns after it is safe to modify the application buffer (your send data) for reuse. 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
  - 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.

Blocking Communications



Standard Send and Receive

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm);

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

Standard Send and Receive

Basic blocking point-to-point communication routine in MPI.

Fortran:

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

Message body — Message envelope

buf array of type **type** see table.
count (INTEGER) number of element of **buf** to be sent
type (INTEGER) MPI type of **buf**
dest (INTEGER) rank of the destination process
tag (INTEGER) number identifying the message
comm (INTEGER) communicator of the sender and receiver
status (INTEGER) array of size **MPI_STATUS_SIZE** containing communication status information (Orig Rank, Tag, Number of elements received)
ierr (INTEGER) error code (if **ierr=0** no error occurs)

Send and Receive - FORTRAN

```
PROGRAM send_recv

INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  A(1) = 3.0
  A(2) = 5.0
  CALL MPI_SEND(A, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  CALL MPI_RECV(A, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  WRITE(6,*) myid,': a(1)=',a(1),' a(2)=',a(2)
END IF

CALL MPI_FINALIZE(ierr)
END
```

Send and Receive - C

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;
  MPI_Status status;
  float a[2];

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if( myid == 0 ) {
    a[0] = 3.0, a[1] = 5.0;
    MPI_Send(a, 2, MPI_FLOAT, 1, 10, MPI_COMM_WORLD);
  } else if( myid == 1 ) {
    MPI_Recv(a, 2, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
    printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
  }

  err = MPI_Finalize();
}
```

Non Blocking communications

- Non-blocking send and receive routines will return almost immediately. They do not wait for any communication events to complete
- 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 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.

Non-Blocking Send and Receive

C:

```
int MPI_Isend(void *buf, int count,
  MPI_Datatype type, int dest, int tag,
  MPI_Comm comm, MPI_Request *req);

int MPI_Irecv (void *buf, int count,
  MPI_Datatype type, int source, int tag,
  MPI_Comm comm, MPI_Request *req);
```

Non-Blocking Send and Receive

FORTRAN:

```
MPI_ISEND(buf, count, type, dest, tag, comm, req,
   ierr)
MPI_IRECV(buf, count, type, source, tag, comm, req,
   ierr)
```

buf array of type **type** see table.
count (INTEGER) number of element of **buf** to be sent
type (INTEGER) MPI type of **buf**
dest (INTEGER) rank of the destination process
tag (INTEGER) number identifying the message
comm (INTEGER) communicator of the sender and receiver
req (INTEGER) output, identifier of the communications handle
ierr (INTEGER) output, error code (if **ierr=0** no error occurs)

Waiting for Completion

FORTRAN:

```
MPI_WAIT(req, status, ierr)
MPI_WAITALL (count,array_of_requests,array_of_statuses, ierr)
```

A call to this subroutine cause the code to wait until the communication pointed by req is complete.

req(INTEGER): input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

Status(INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

ierr(INTEGER) output, error code (if **ierr=0** no error occours).

C:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
Int MPI_Waitall (count,&array_of_requests,&array_of_statuses)
```

Testing Completion

FORTRAN:

```
MPI_TEST(req, flag, status, ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
```

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

Req(INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

Flag(LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise

Status(INTEGER) array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

Ierr(INTEGER) output, error code (if **ierr=0** no error occurs).

C:

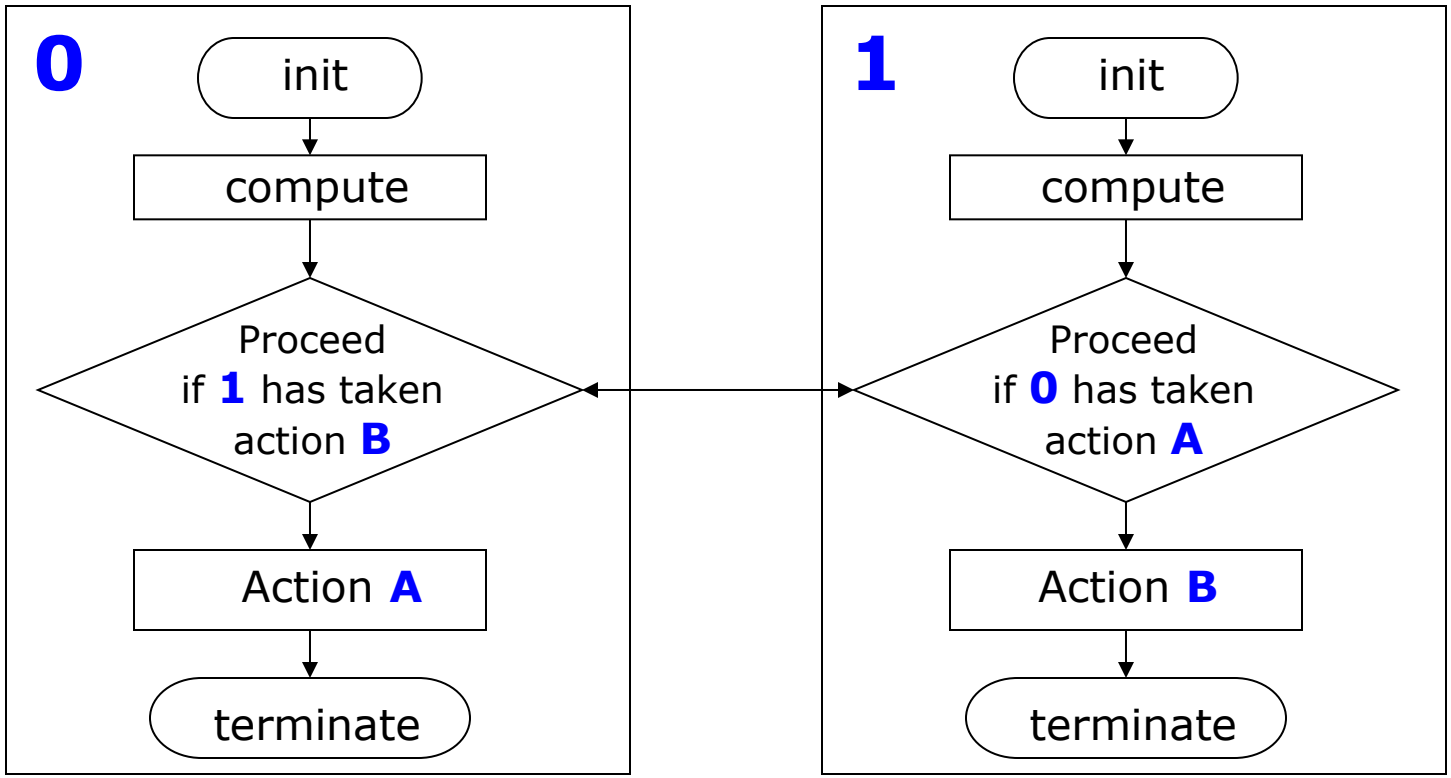
```
int MPI_Test (&request,&flag,&status)
Int MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
```

Wildcards

- Both in FORTRAN and C **`MPI_RECV`** accepts wildcard:
- To receive from any source: MPI_ANY_SOURCE
- To receive with any tag: MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.

DEADLOCK

Deadlock or a Race condition occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.



Simple DEADLOCK

```
PROGRAM deadlock
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
END IF
WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

Avoiding DEADLOCK

```
PROGRAM avoid_lock
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
  CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

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.
- The easiest way to send and receive data without worrying about deadlocks

FORTRAN:

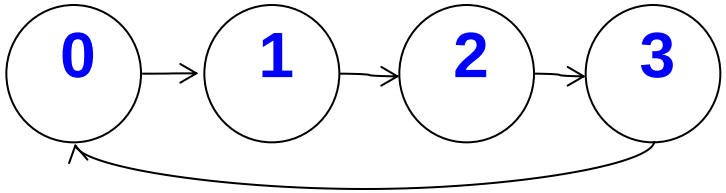
```
CALL MPI_SENDRECV(sndbuf,snd_size, snd_type,destid,tag,
rcvbuf, rcv_size, rcv_type, sourceid, tag,
comm, status, ierr)
```

Sender side

Receiver side

SendRecv example

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

int main(int argc, char *argv[])
{
    int myid, numprocs, left, right,i;
    int buffer[1], buffer2[1];
    MPI_Request request;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    right = (myid + 1) % numprocs;
    left = myid - 1;
    if (left < 0)
        left = numprocs - 1;

   buffer[0]=myid;
   MPI_Sendrecv(buffer, 10, MPI_INT, left, 123, buffer2, 10, MPI_INT, right,
    123, MPI_COMM_WORLD, &status);
}
```



Useful for cyclic communication patterns

SEND and RECV variants

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	Message sent (receive state unknown)	`MPI_SEND`	`MPI_ISEND`
receive	Completes when a matching message has arrived	`MPI_RECV`	`MPI_IRECV`
Synchronous send	Only completes after a matching recv() is posted and the receive operation is started.	`MPI_SSEND`	`MPI_ISSEND`
Buffered send	Always completes, irrespective of receiver Guarantees the message being buffered	`MPI_BSEND`	`MPI_IBSEND`
Ready send	Always completes, irrespective of whether the receive has completed	`MPI_RSEND`	`MPI_IRSEND`

Final Comments

- MPI is a standard for message-passing and has numerous implementations (OpenMPI, IntelMPI, MPICH, etc)
- MPI uses send and receive calls to manage communications between two processes (point-to-point)
- The calls can be blocking or non-blocking.
- Non-blocking calls can be used to overlap communication with computation but wait routines are needed for synchronisation.
- Deadlock is a common error and is due to incorrect order of send/receive