DATA SCIENCE &
SCIENTIFIC COMPUTING

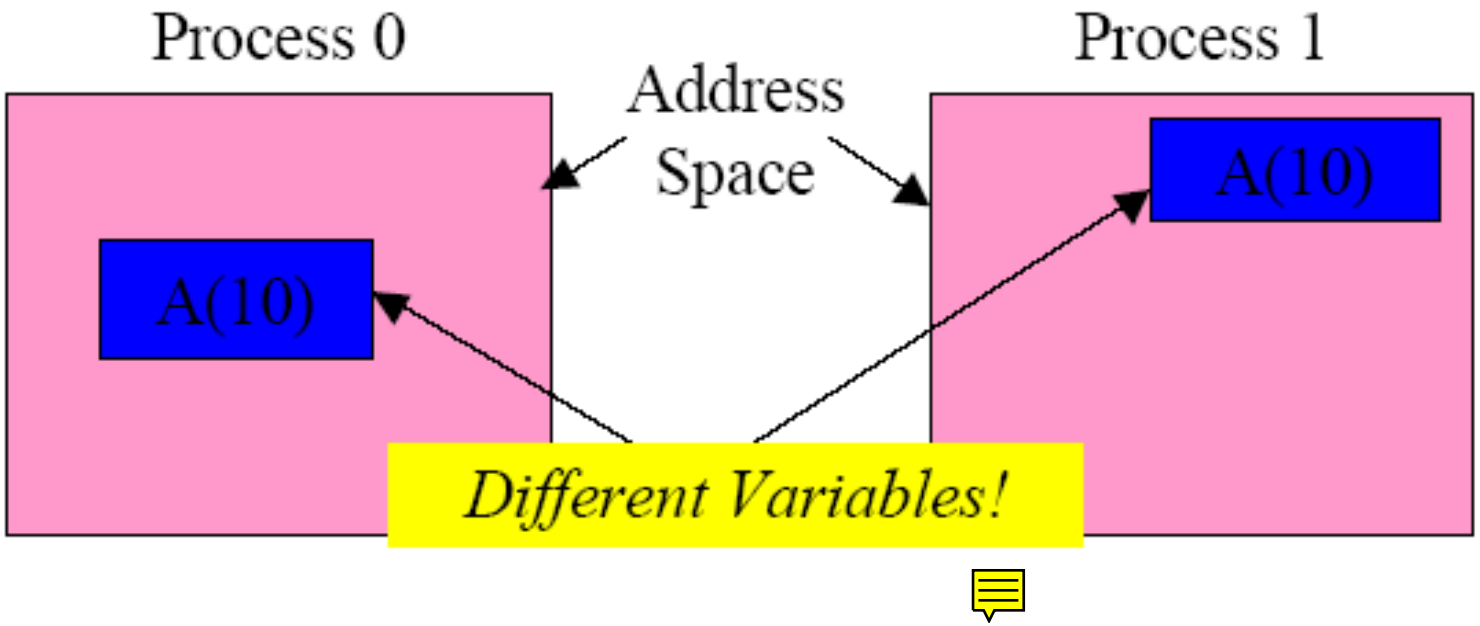
L04: intro to MPI programming

- Stefano Cozzini
- CNR-IOM and eXact lab srl

Message passing paradigm

- Parallel programs consist of separate processes, each with its own address space
- Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
- Programmer manages memory motion
- Collective operations Operations ON GROUPS OF PROCESSES
  - On arbitrary set of processes
- Data distribution => domain decomposition
  - Also managed by programmer

Distributed memory (shared nothing approach)



Different Linux processes have different private memory spaces, reserved to that process.>> Easy execution of "local" MPI codes, like the plain ol' 1CPU-1core.>> MPI

What is MPI ?

- A message-passing library specification
- extended message-passing model
- not a language or compiler specification
- not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers
- Latest version of the standard MPI- 3.1

What is MPI ?

A STANDARD...

- The actual implementation of the standard is demanded to the software developers of the different systems
- In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives
- many different implementations
  - MPICH (the original one)
  - OpenMPI
  - IntelMPI

Some reason to use MPI

- International standard
- MPI evolves: MPI 1.0 was first introduced in 1994, most current version is MPI 3.3 (Nov. 2016)
- Available on almost all parallel systems (free MPICH, Open MPI used on many clusters), with interfaces for C/C++ and Fortran
- Supplies many communication variations and optimized functions for a wide range of needs
- Works both on distributed memory (DM) and shared memory (SM) hardware architectures
- Supports large program development and integration of multiple modules

How to program with MPI ?

- MPI is a library:
  - All operations are performed with subroutine calls
- Basic definitions are in
  - mpi.h for C/C++
  - mpif.h for Fortran 77 and 90
  - MPI module for Fortran 90 (optional)

Compiling MPI Programs

NO STANDARD: left to the implementations:

- Generally:
  - You should specify the appropriate include directory (i.e. -I/mpidir/include)
  - You should specify the mpi library (i.e. -L/mpidir/lib -lmpi)
- Usually MPI compiler wrappers do this job for you. (i.e. Mpicc)
  - Check on your machine...

Running MPI programs

The MPI Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.

In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.

Many implementations provided `mpirun -np 4 a.out` to run an MPI program

Many parallel systems use a batch environment to share resources among users

The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer

Writing MPI program..

- Modify your serial program to insert MPI routines to distribute data and loads to different processors.

Basic Features of MPI routines

Calls may be roughly divided into four classes:

- Calls used to initialize, manage, and terminate communications
- Calls used to communicate between pairs of processors. (Pair communication)
- Calls used to communicate among groups of processors. (Collective communication)
- Calls to create data types.

Basic MPI routines

- MPI_INIT: initialize MPI
- MPI_COMM_SIZE: how many Processors?
- MPI_COMM_RANK: identify the Processor
- MPI_SEND : send data 
- MPI_RECV: receive data
- MPI_FINALIZE: close MPI

(Almost) All you need is to know this 6 calls

Your First Program: Hello World!

Fortran

```
PROGRAM hello
 INCLUDE 'mpif.h'
 INTEGER err
 CALL MPI_INIT(err)
 call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
 print *, 'I am ', rank, ' of ', size
 CALL MPI_FINALIZE(err)
END
```

C

```
#include <stdio.h>
#include <mpi.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( "I am %d of %d\n", rank, size );
  MPI_Finalize();
}
```

Some initial notes

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process including the printf/print statements
  - I/O not part of MPI-1 (MPI-IO part of MPI-2)
  - print and write to standard output or error not part of MPI-1 or MPI-2 or MPI-3
- output order is undefined (may be interleaved by character, line, or blocks of characters),
- A consequence of the requirement that non-MPI statements execute independently

Initializing and Exiting MPI

Initializing the MPI environment

C: **int MPI_Init(int *argc, char ***argv);**

Fortran:

```
INTEGER IERR
CALL MPI_INIT(IERR)
```

Finalizing MPI environment

C:

```
int MPI_Finalize()
```

Fortran:

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

This two subprograms should be called by all processes, and no other MPI calls are allowed before **`mpi_init`** and after **`mpi_finalize`**

MPI Communicator

The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.

There is a default communicator (automatically defined):

MPI_COMM_WORLD

It identifies the group of all processes.

All MPI communication subroutines have a communicator argument. The Programmer could define many communicators at the same time

Communicator Size and Process Rank

How many processors are associated with a communicator?

C:

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

Fortran:

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

OUTPUT: SIZE

What is the ID of a processor in a group?

C:

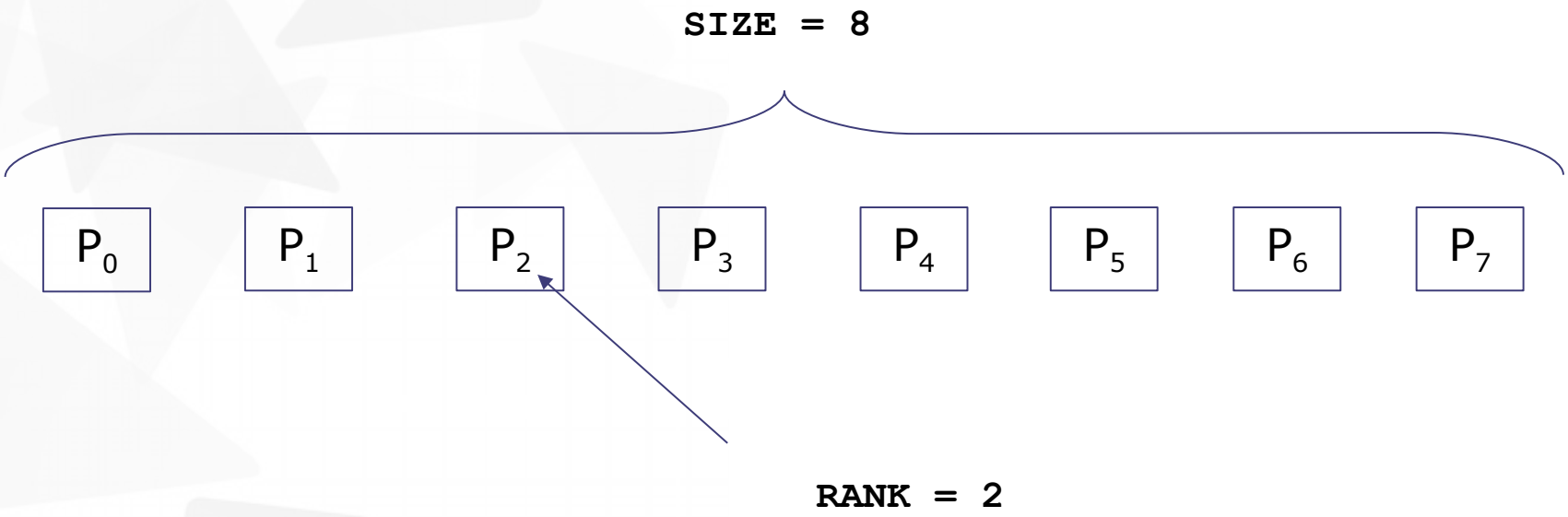
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

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

OUTPUT: RANK

Communicator Size and Process Rank



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

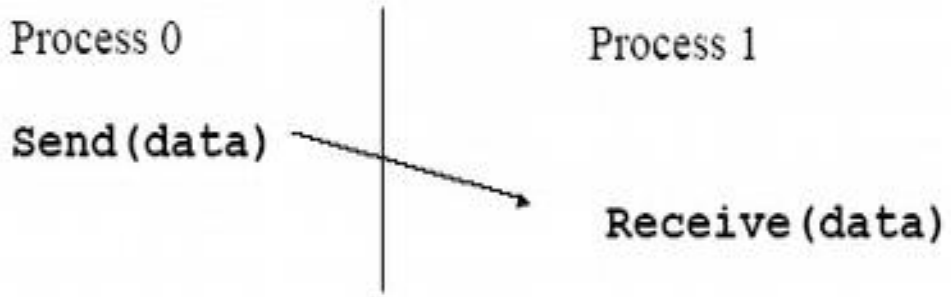
`rank` is the index of the process within a group associated to a communicator (**`rank`** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication

Communication Ingredients

- To send a message via mail we typically have:
  - An envelope (with possibly some hints on the content itself... i.e., advertisement, bills, greetings....)
  - A message
  - A destination address
  - A sender address

For MPI it is exactly the same thing...

MPI basic send/receive



questions:

- How will "data" be described? datatypes
- How will processes be identified? rank/comm
- How will the receiver recognize messages? tag
- What will it mean for these operations to complete? blocking/non-blocking

BLOCKING: Wait until the receiver receives the message;(a.k.a.: wait until it's safe to move on)NONBLOCKING: Send; will check later!

Describing Data

The data in a message to send or receive is described by a triple (address, count, datatype), where

An MPI datatype is recursively defined as:

- predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
- a contiguous array of MPI datatypes
- a strided block of datatypes
- an indexed array of blocks of datatypes
- an arbitrary structure of datatypes

There are MPI functions to construct custom datatypes, in particular ones for subarrays

Fortran - MPI Basic 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 Basic 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	

Data tag

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive

Our First Sent Message....

The simplest call:

MPI_send(buffer, count, data_type, destination,tag, communicator)

where:

BUFFER: data to send

COUNT: number of elements in buffer .

DATA_TYPE : which kind of data types in buffer ?

DESTINATION the receiver

TAG: the label of the message

COMMUNICATOR set of processors involved

..and our First Received message.

The simplest call:

MPI_recv(buffer, count, data_type, source, tag, communicator, status)

Similar to send with the following differences:

- SOURCE is the sender ; can be set as MPI_any_source (receive a message from any processor within the communicator)
- TAG the label of message: can be set as MPI_any_tag: receive any kind of message
- STATUS integer array with information on message in case of error

The status array

Status is a data structure allocated in the user's program.

In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

MPI: a FORTRAN example..

```
Program MPI
   Implicit None
!
   Include 'mpif.h'
!
   Integer                                          :: rank
   Integer                                          :: buffer
   Integer, Dimension( 1:MPI_status_size )          :: status
   Integer                                          :: error
!
   Call MPI_init( error )
   Call MPI_comm_rank( MPI_comm_world, rank, error )
!
   If( rank == 0 ) Then
      buffer = 33
      Call MPI_send( buffer, 1, MPI_integer, 1, 10, &
                     MPI_comm_world, error )
   End If
!
   If( rank == 1 ) Then
      Call MPI_recv( buffer, 1, MPI_integer, 0, 10, &
                     MPI_comm_world, status, error )
      Print*, 'Rank ', rank, ' buffer=', buffer
      If( buffer /= 33 ) Print*, 'fail'
   End If
   Call MPI_finalize( error )
End Program MPI
```

Blocking and Non-Blocking

Q: When is a SEND instruction complete?
A: When it is safe to change the data that we sent.

Q: When is a RECEIVE instruction complete?
A: When it is safe to access the data we received.

With both communications (send and receive) we have two choices:

Start a communication and wait for it to complete:
BLOCKING approach
Start a communication and return control to the main program:
NON-BLOCKING approach

The Non-Blocking approach REQUIRES us to check for completion before we can modify/access the sent/received data!!!

MPI_send and MPI_recv

MPI_SEND() and MPI_RECV() are blocking operations.

Arere they really blocking ?

MPI_SEND() and MPI_RECV() are blocking operations.

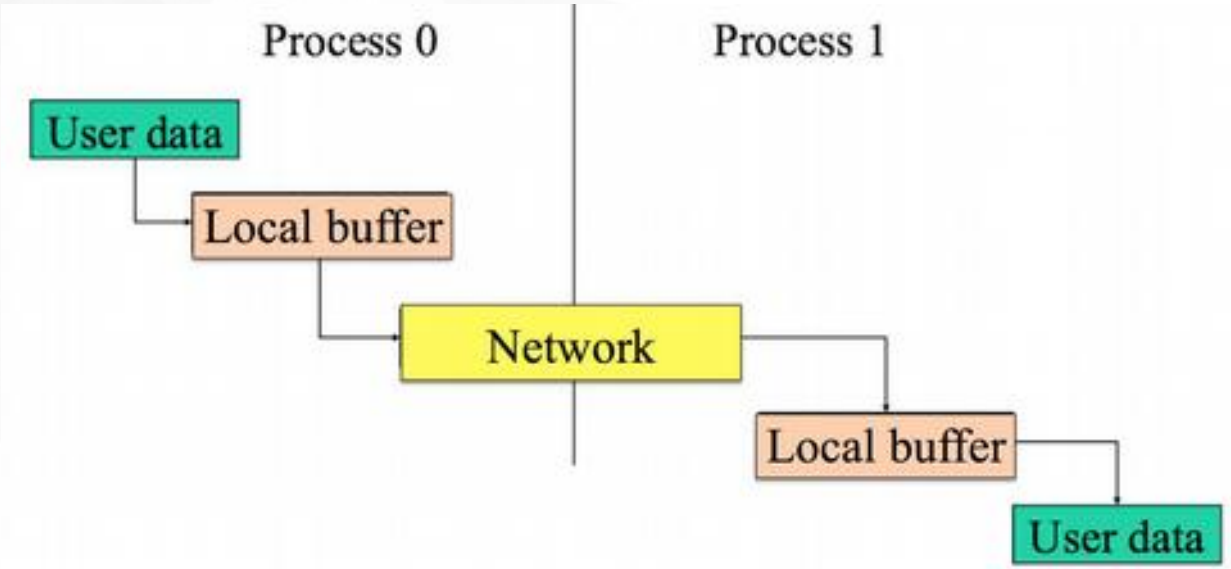
However: often a system buffer is used that allows small messages to be non-blocking send-recv handshakes, but large messages will be blocking.

MPI implementation (not the MPI standard) decides this

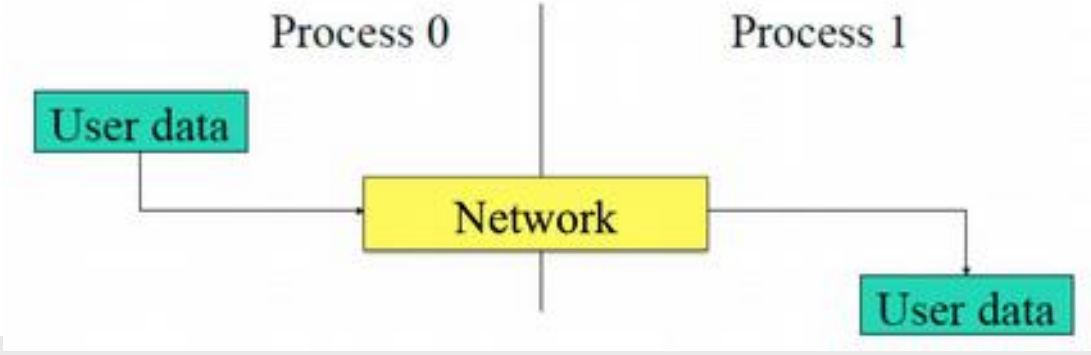
Blocking communication can be unsafe and may lead to deadlocks.

Again on this

- Small messages make use of system-supplied buffer



- Large message are really blocking:



Pros and Cons of Non-Blocking Send/ Receive

- Non-Blocking communications allows the separation between the initiation of the communication and the completion.
- Advantages:
  - between the initiation and completion the program could do other useful computation (latency hiding).
- Disadvantages:
  - the programmer has to insert code to check for completion.

Communication mode

- 4 different send types:
  - Standard: let MPI decide the best strategy…
  - Synchronous: it is complete when the receiver acknowledged the reception of the message
  - Buffered: it is complete when the data has been copied to a local buffer
  - Ready: requires a receiver to be already waiting for the message
- 1 single receive type

Communication Modes and MPI Subroutines

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	**Message sent (receive state unknown)**	**MPI_SEND**	**MPI_ISEND**
receive	**Completes when a message has arrived**	**MPI_RECV**	**MPI_IRECV**
Synchronous send	**Only completes when the receive has completed**	MPI_SSEND	MPI_ISSEND
Buffered send	**Always completes, irrespective of receiver**	MPI_BSEND	MPI_IBSEND
Ready send	**Always completes, irrespective of whether the receive has completed**	MPI_RSEND	MPI_IRSEND

Non-Blocking Send and Receive

Fortran:

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

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 dest, int tag,
  MPI_Comm comm, MPI_Request *req);
```

Waiting and Testing for Completion

Fortran:

MPI_WAIT(req, status, 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);

Waiting and Testing for Completion

Fortran:

MPI_TEST(req, flag, status, 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 occours).

C:

```
int MPI_Wait(MPI_Request *req, int *flag, MPI_Status *status);
```

MPI: a case study

Problem: exchanging data between two processes

```
If( rank == 0 ) Then
     Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
     Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 ) Then
     Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
     Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
  End If
```

DEADLOCK

Solution A

USE BUFFERED SEND: bsend
send and go back so the deadlock is avoided

```
If( rank == 0 ) Then
     Call MPI_Bsend( buffer1, 1, MPI_integer, 1, 10, &
                     MPI_comm_world, error )
     Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                     MPI_comm_world, status, error )
  Else If( rank == 1 ) Then
     Call MPI_Bsend( buffer2, 1, MPI_integer, 0, 20, &
                     MPI_comm_world, error )
     Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                     MPI_comm_world, status, error )
  End If
```

NOTES:

1. Requires a copy therefore is not efficient for large data set memory problems

Solution B

Use non blocking SEND : isend

send go back but now is not safe to change the buffer

```
If( rank == 0 ) Then
     Call MPI_Isend( buffer1, 1, MPI_integer, 1, 10, &
                     MPI_comm_world, REQUEST, error )
     Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                     MPI_comm_world, status, error )
  Else If( rank == 1 ) Then
     Call MPI_Isend( buffer2, 1, MPI_integer, 0, 20, &
                     MPI_comm_world, REQUEST, error )
     Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                     MPI_comm_world, status, error )
  End If
  Call MPI_wait( REQUEST, status ) ! Wait until send is complete
```

NOTES:

1 A handle is introduced to test the status of message.

2. More efficient of the previous solutions

Solution C

Exchange send/recv order on one processor

```
If( rank == 0 ) Then
     Call MPI_send( buffer1, 1, MPI_integer, 1, 10, &
                    MPI_comm_world, error )
     Call MPI_recv( buffer2, 1, MPI_integer, 1, 20, &
                    MPI_comm_world, status, error )
  Else If( rank == 1 ) Then
     Call MPI_recv( buffer1, 1, MPI_integer, 0, 10, &
                    MPI_comm_world, status, error )
     Call MPI_send( buffer2, 1, MPI_integer, 0, 20, &
                    MPI_comm_world, error )
  End If
```

NOTES:

efficient and suggested !

Exercises and tutorials

- Compile/Run and understand usage of MPI programs
  - mpi_pi.c
  - hello_world.c/f90
  - mpi_env_call.c
  - send_message.f90
- First MPI exercise: fix deadlock problem
- Second MPI exercise: play with MPI_send call on mpi_pi.c
- Third MPI exercise: implement the sum of N number using MPI paradigm

END OF FIRST PART