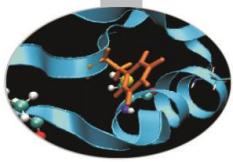
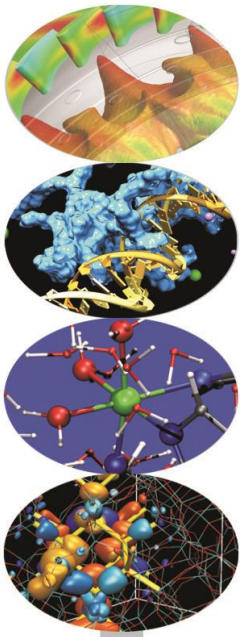
CINECA SCAI
SuperComputing Applications and Innovation



MPI introduction

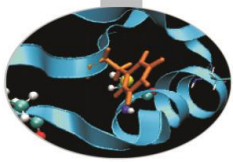


Alessandro Marani

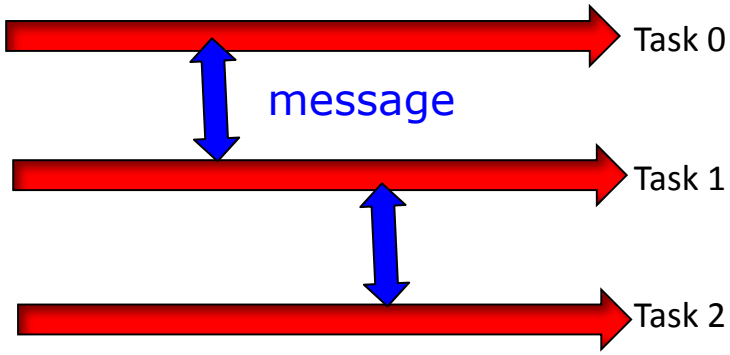
Segrate, November 2017

CINECA

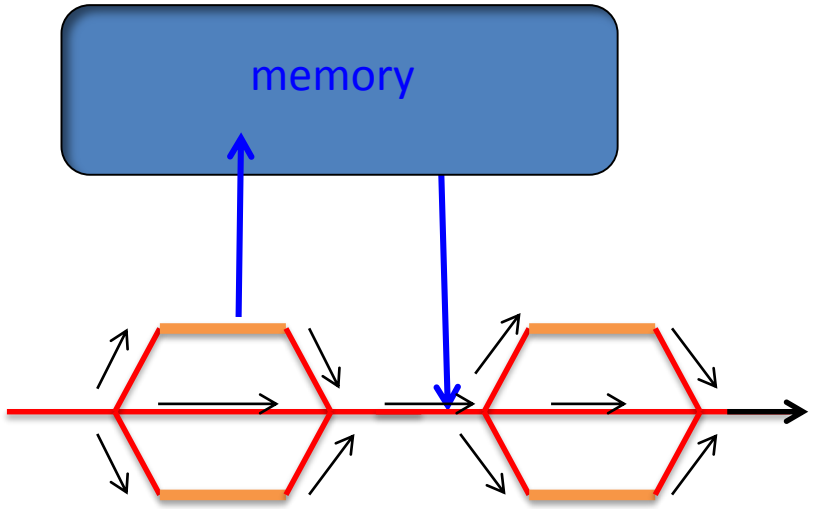
Message passing and shared memory parallelism



message passing



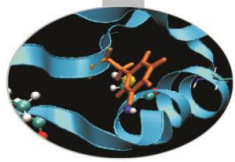
shared memory



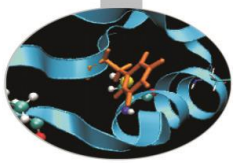
Multiple *tasks* exchange data via explicit messages

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

Message Passing



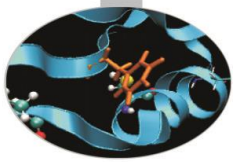
- Unlike the shared memory model, resources are local;
- Each process operates in its own environment (logical address space) and communication occurs via the exchange of messages;
- Messages can be instructions, data or synchronisation signals;
- The message passing scheme can also be implemented on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

MPI (Message Passing Interface)

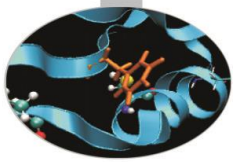


MPI origin: 1992, "Workshop on Standards for Message Passing in a Distributed Memory Environment“

60 experts from more than 40 organisations (IBM T. J. Watson Research Center, Intel’s NX/2, Express, nCUBE’s Vertex, p4, PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL, …).

Many of them coming from the most important constructors of parallel computers or researchers from University, government and private research centres.

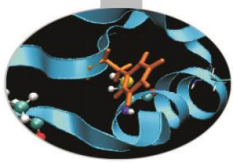
MPI (Message Passing Interface)



MPI development:

- **MPI-1.0**: June 1994;
- **MPI-1.1**: June 1995;
- **MPI-1.2** e **MPI-2**: June 1997
- **MPI-1.3** e **MPI-2.1**: May, June 2008
- **MPI-2.2** : September 2009
- **MPI-3.0** : September 2012
- **MPI-3.1** : June 2015

Goals of the MPI standard



MPI's prime goals are:

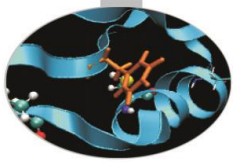
- To allow efficient implementation
- To provide source-code portability

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 neighbourhood collectives.



MPI versions

Some of the public domain most used MPI libraries:

MPICH : Argonne National Laboratory

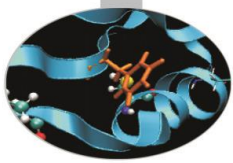
Open MPI : "open source" implementation of MPI-2

CHIMP/MPI : Edinburgh University

LAM : Ohio Supercomputer Center

To realize a (simple) parallel program *only six MPI functions* are needed.

But if the program is a complex one and the best performances are sought for, the whole MPI library may be used, with more than a hundred functions.

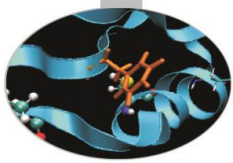


MPI introduction

What we will learn in this lesson on MPI library:

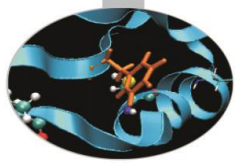
- Compiling and executing MPI programs
- C and Fortran calling syntax
- Environment
- Point to point communications
- Collective communications
- Synchronization

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 you may choose which MPI to use, on other systems you have only the vendor-supplied version.

My first program



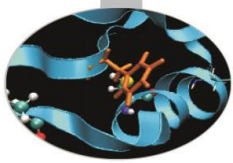
Fortran

```
program first
  character(100) :: &
  &        message="Hello"

  write(6,FMT="(A)") message

   stop
end program first
```

C

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

int main(int argc, char *argv[])
{
    char message[100]="Hello";

    printf("%s\n",message);

    return 0;
}
```

Compiling notes



To compile programs that make use of MPI library:

```
mpif90/mpicc/mpiCC -o <executable> <file 1> <file 2> … <file n>
```

Where: `<file n>` - program source files

`<executable>` - executable file

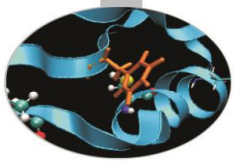
To start parallel execution on one node only:

```
mpirun -np <processor_number> <executable> <exe_params>
```

To start parallel execution on many nodes:

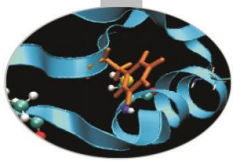
```
mpirun -np <processor_number> -machinefile <node_list_file> \
       <executable> <exe_params>
```

My first program



Try to compile and run with MPI the example *first*, either in Fortran or in C.

My first program

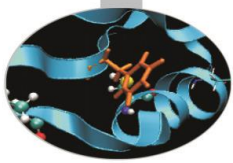


Try to compile and run with MPI the example *first*, either in Fortran or in C.

Well, the result is not much useful, but one thing is shown: every MPI process run the entire program code.

How then do we take advantage of parallel execution?

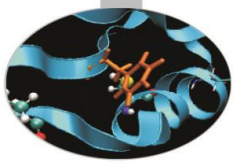
My first MPI program



Fortran

```
program first_mpi
  include 'mpif.h'
  integer :: my_rank
  character(100) :: message="Hello from "
  integer :: ierr

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)

  write(6,FMT="(A,I3)") TRIM(message),my_rank

  call MPI_Finalize(ierr)

  stop
end program first_mpi
```

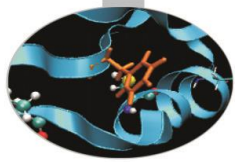
My first MPI program



C

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[]) /* first_mpi */
{
    int my_rank;
    char message[100]="Hello from ";
    int ierr;

    ierr = MPI_Init(&argc,&argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    printf("%s %d\n",message,my_rank);

    MPI_Finalize();

    return 0;
}
```

My first MPI program

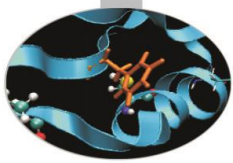


Try to compile and run the example *first_mpi*, either in Fortran or in C.

The result is more interesting, and can suggest how the program might be parallelized.

Partitioning the data or program operations on the basis of the process id can lead to parallel execution.

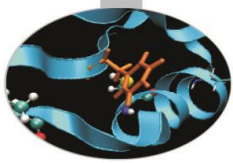
My first MPI program



Fortran - add the tags **-stdin all** to run this program

```
program greetings
    include 'mpif.h'
    integer :: my_rank, ierr
    character(100) :: message=", greetings from "
    character(100) :: name

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)

   write(6,FMT="(A)") "What's your name?"
   read(5,"(A)") name

   write(6,FMT="(3A,I3)") "Hi ",TRIM(name),TRIM(message),my_rank

   call MPI_Finalize(ierr)
end program greetings
```

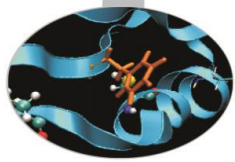
My first MPI program



C - add the tags **-stdin all** to run this program

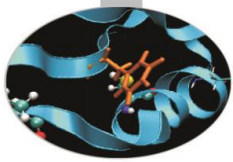
```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[]) {  /* greetings */
    int my_rank, ierr;
    char name[100], message[100]=", greetings from ";

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    printf("What's your name?\n");
    scanf("%s",name);
    printf("Hi %s %s %d\n",name,message,my_rank);

    MPI_Finalize();
    return 0;
}
```

My first MPI program



Compile and run with MPI the example *greetings*, either in Fortran or in C.

My first MPI program

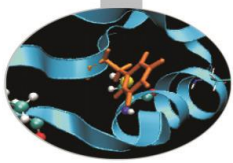


Compile and run with MPI the example *greetings*, either in Fortran or in C.

Although with the flags '-stdin all' the stdin input is shared among the processes, the query is written many times.

Some things in a program should be carried out by one process only, but so how to share data?

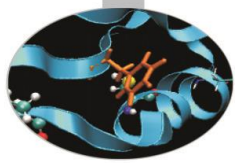
My first MPI program



Fortran

```
program greetings_comm
    include 'mpif.h'
    integer :: my_rank, np, sender, recipient, ierr
    integer :: status(MPI_STATUS_SIZE)
    character(100) :: message=", greetings from "
    character(100) :: name=" "

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, np, ierr)

   sender = my_rank - 1;  recipient = my_rank + 1
   if ( my_rank == 0 ) then
      write(6,FMT="(A)") "What's your name?"
      read(5,"(A)") name
      ln = len_trim(name)
   endif
.  .  .
```

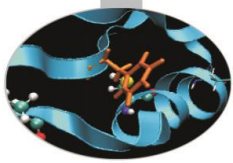
My first MPI program



Fortran

```
.   .   .
    if ( sender >= 0 ) then
       tag = 1
       call MPI_Recv(name, 100, MPI_CHARACTER, sender, tag, &
      &                                 MPI_COMM_WORLD, status, ierr)
    endif

    if ( recipient < np ) then
       tag = 1
       call MPI_Send(name, 100, MPI_CHARACTER, recipient, tag, &
      &                                       MPI_COMM_WORLD, ierr)
    endif

    write(6,FMT="(3A,I3)") "Hi ",TRIM(name),TRIM(message),my_rank

    call MPI_Finalize(ierr)
end program greetings_comm
```

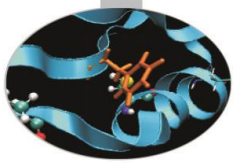
My first MPI program



C

```
#include "mpi.h"
int main( int argc, char *argv[])   /* greetings_comm */
{
    int my_rank, np, sender, recipient, tag, ln, ierr;
    char name[100], message[100]=", greetings from ";
    MPI_Status status;

    ierr = MPI_Init(&argc,&argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    ierr = MPI_Comm_size(MPI_COMM_WORLD,&np);

    sender = my_rank - 1;  recipient = my_rank + 1;
    if ( my_rank == 0 ) {
      printf("What's your name?\n");
      scanf("%s",name);
      ln = strlen(name);
    }
.   .   .
```

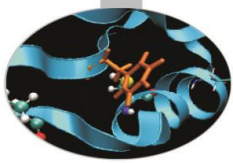
My first MPI program



C

```
. . .

    if ( sender >= 0 ) {
      tag = 0;
      ierr = MPI_Recv(&ln, 1, MPI_INT, sender, tag,
                                       MPI_COMM_WORLD, &status);
      tag = 1;
      ierr = MPI_Recv(name, ln, MPI_CHAR, sender, tag,
                                       MPI_COMM_WORLD, &status);
    }

. . .
```

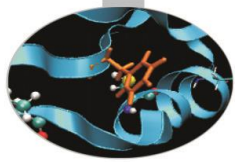
My first MPI program



C

```
. . .

    if ( recipient < np ) {
      tag = 0;
      ierr =  MPI_Send(&ln, 1, MPI_INT, recipient, tag,
                                                MPI_COMM_WORLD);
      tag = 1;
      ierr =  MPI_Send(name, ln, MPI_CHAR, recipient, tag,
                                                MPI_COMM_WORLD);
    }
    name[ln]='\0';
    printf("Hi %s %s %d\n",name,message,my_rank);


    MPI_Finalize();

    return 0;
}
```

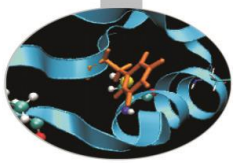
My first MPI program



Compile and run with MPI the example *greetings_comm*, either in Fortran or in C.

Now input data is carried out by one process only. Process 0 is often used as a 'master process' to perform I/O and to manage data and work distribution.

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
```

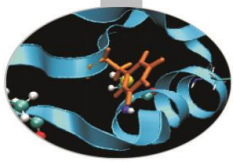
Fortran 08 (MPI-3):

```
USE MPI_F08
```

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

FORTRAN note:
The FORTRAN include and module forms are *not equivalent*: the module can also do type checking. Some compilers gave problems with the module but it is now highly recommended to use the module, particularly for FORTRAN 2008 (most rigorous type-checking)

MPI syntax

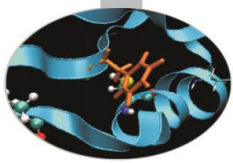


Every Fortran subroutine in the MPI library returns, as the last argument, an INTEGER type error code.

Every MPI C function returns an int value representing the error code.

Whenever a MPI function has exited without errors, the error code should have the value `MPI_SUCCESS`.

The value of `MPI_SUCCESS` is usually 0.

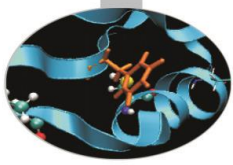


MPI syntax

```
include "mpif.h"
integer :: ierror
....
call MPI_send (..., ierror)
if (ierror .ne. MPI_SUCCESS) then
    write (*,*)"SEND operation failed"
    stop 777
end if
```

Error code values different from `MPI_SUCCESS` are implementation dependent.

MPI syntax



Generally speaking, MPI functions have the following prototype:

```
call  MPI_name  ( parameter, ..., ierror )
```
fortran

```
rt = MPI_Name  (parameter, ...)
```
C/C++

To initialize the MPI environment the `MPI_Init` function must be called:

```
call  MPI_INIT ( ierror )
```
fortran

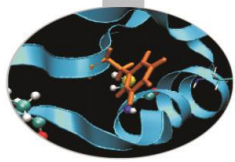
```
rt = MPI_Init(int *argc, char ***argv)
```
C/C++

On ending parallel computations the `MPI_Finalize` function should be called, otherwise processes could remain alive on local or remote computing units:

```
call MPI_FINALIZE  ( ierror )
```
fortran

```
rt = MPI_Finalize();
```
C/C++

Groups of MPI processes

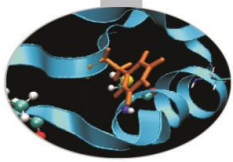


A **group** is an ordered set of processes.

All MPI processes are organized in groups; each process belongs to one or more groups.

Processes are sequentially ordered in an unambiguous way. In each group each process has its own identifying number or **rank**.

Groups of MPI processes

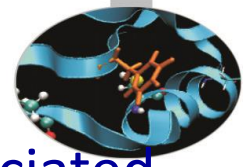


Process identifying numbers are integer numbers in the range **0 – *N-1*** where ***N*** is the group size.

On initializing MPI a default group is created containing all the processes: this group is associated to the default communicator **`MPI_COMM_WORLD`**.

If the processes are not many the default group is sufficient. Otherwise it may be convenient to create new groups defined as subsets, either disjoined or not, of the default group or formerly created groups.

MPI processes



The following function returns the extension of the group associated to a communicator, i.e. the number of processes belonging to the group:

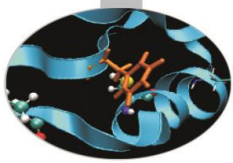
```
call  MPI_COMM_SIZE  ( comm, size, ierr )
```
fortran

```
ierr = int MPI_Comm_size ( MPI_Comm comm, int
*size )
```
C/C++

Where:

`comm` = communicator handle
`size` = number of processes
`ierr` = error code

MPI processes



The following function returns the rank of the calling process:

```
call  MPI_COMM_RANK ( comm, rank, ierr )
```
fortran

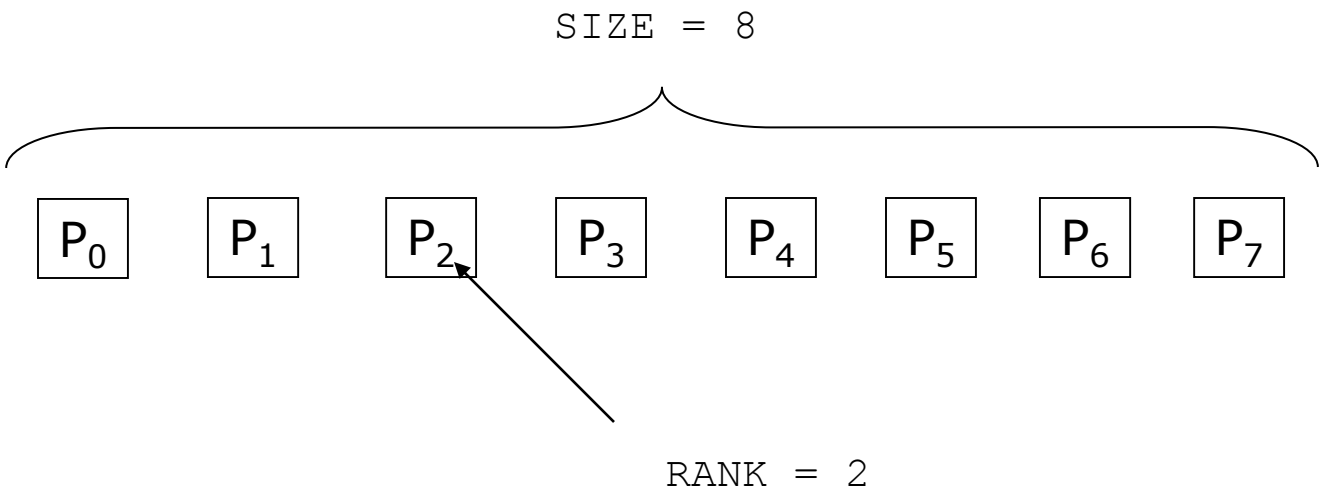
```
ierr = MPI_Comm_rank ( MPI_Comm comm, int
*rank )
```
C/C++

Where:

`comm` = communicator handle
`rank` = process rank (a number in the range 0 - size-1)
`ierr` = error code

Communicator Size and Process Rank / 1

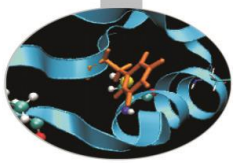
How many processes are contained within a communicator?



`size` is the number of processes 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

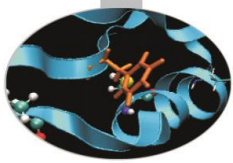
Communication domains (communicators)



On MPI initialization the default communicator `MPI_COMM_WORLD` is generated. It allows all the activated processes to communicate each other.

Sometimes it is necessary to generate new communicators either by duplicating existing ones, or by associating to newly created groups of processes.

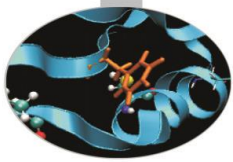
Communication domains (communicators)



A new communicator should be created every time a new group of processes is generated. A new group is always generated by choosing processes from a wider already existing group.

The creation of a group of processes is a local operation, it is realized at process level. On the contrary, the creation of a new communicator is a global operation and involves (hidden) communications among all the processes of the group.

Point to point communications



Point to point communications realize connections between two processes.

From the programmer point of view communications depend on a *communicator* and are identified by a *handle* and a *tag*.

The communicator defines the processes that can be involved.

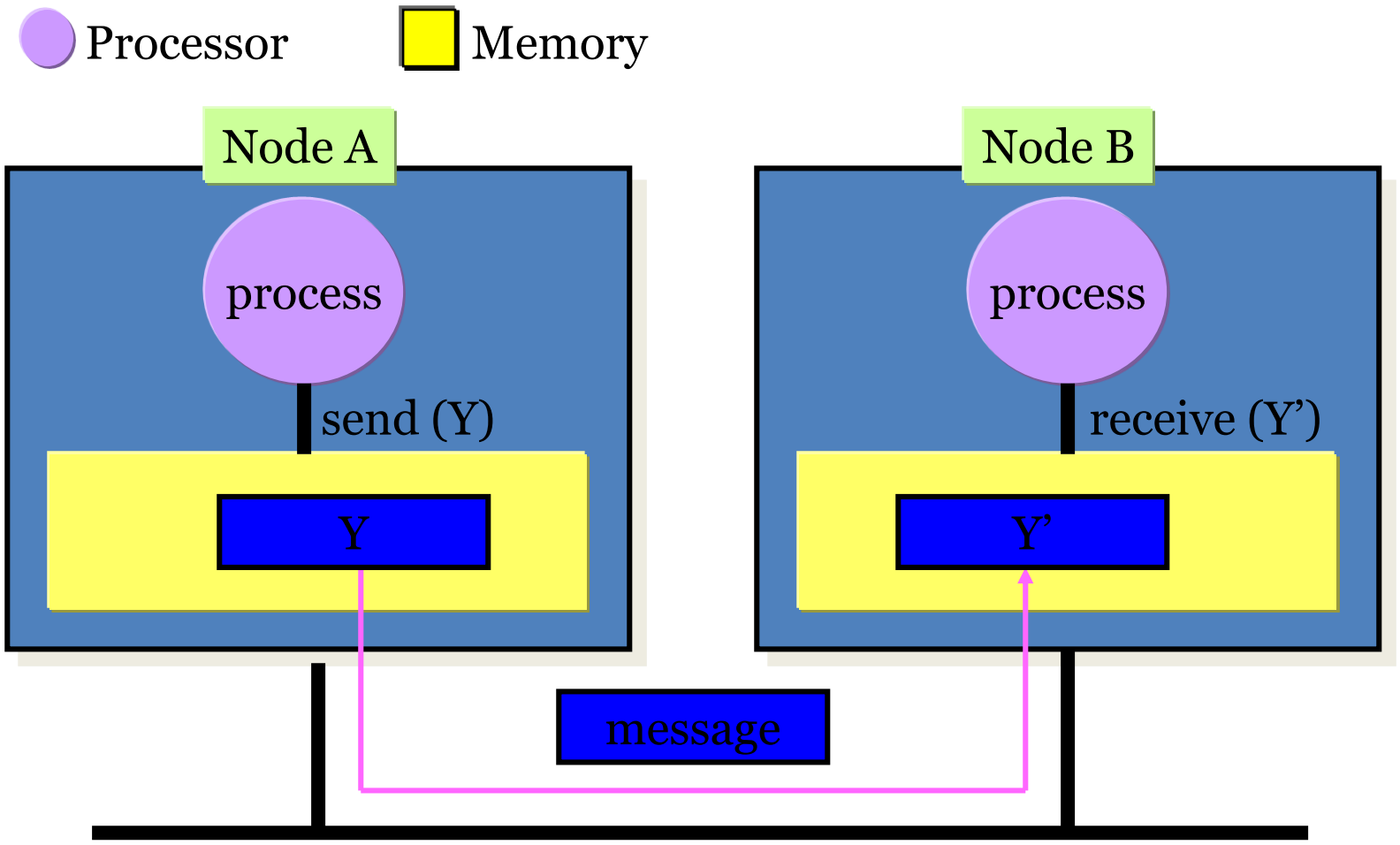
The tag is used to differentiate messages.

The handle may be useful whenever it is necessary to control the completion of the communicating operation.

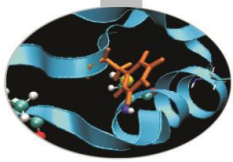
A communication is said to be locally completed if the process has terminated the operation.

A communication is said to be globally completed when all the involved processes have terminated the operation.

MPI Programming Model



Point to point communications



Communication calls may be **blocking** or **nonblocking**.

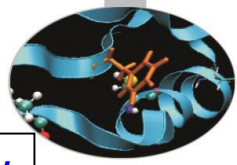
The functions relevant to **blocking calls** do not return control unless data in the message can be safely modified without changing the message data.

These functions (MPI_Send, MPI_Recv) are **very reliable** but the program execution may be slowed down because the processes are blocked until the message has been received.

The functions relevant to **nonblocking calls** are faster but **care must be taken** that the sent data are actually received and are not corrupted.

Therefore data sent by nonblocking calls can not be modified unless it is safe to do so. The functions `MPI_Wait` or `MPI_Test` should be called for checking.

The Message

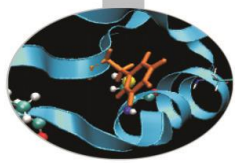


- Data is exchanged in the buffer, an array of *count* elements of some particular MPI data type
- One argument that usually must be given to MPI routines is the *type* of the data being passed.
- This allows MPI programs to run automatically in heterogeneous environments
- C types are different from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



Messages

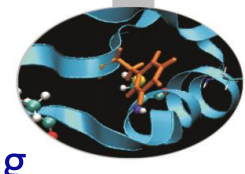
The receiving process may receive messages in random order if they are sent by different processes.

Care must be taken to insure the correct receiving order of the messages.

The following rules are always true:

- Messages with the same tag sent by the same process will be received in the sending sequence.
- Messages sent by nonblocking calls will be received in the sending order. This is important because otherwise large messages could be received after smaller ones sent later.

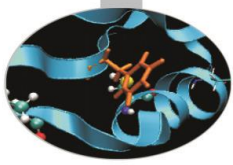
Basic data types



MPI messages are sent as arrays of data *homogeneous in type*. In sending and receiving calls only one data type can and shall be specified. The allowed data types may be either *basic* or *derived*. Derived types shall be explicitly defined in the program and explicitly registered in the MPI system.

Basic types in Fortran	**Basic types in C**
MPI_INTEGER	MPI_CHAR
MPI_REAL	MPI_SHORT
MPI_DOUBLE_PRECISION	MPI_INT
MPI_COMPLEX	MPI_LONG
MPI_DOUBLE_COMPLEX	MPI_UNSIGNED_CHAR
MPI_LOGICAL	MPI_UNSIGNED_SHORT
MPI_CHARACTER	MPI_UNSIGNED
MPI_BYTE	MPI_UNSIGNED_LONG
MPI_PACKED	MPI_FLOAT
	MPI_DOUBLE
	MPI_LONG_DOUBLE
	MPI_BYTE
	MPI_PACKED

Sending calls



The prototype of a sending function is:

```
type :: buf(count)
integer :: count, datatype, dest, tag, comm, ierror
call  MPI_send(buf, count, datatype, dest, tag, &
       & comm, ierror )
```

fortran

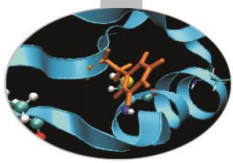
```
ierror = MPI_Send( void *buf, int count, MPI_Datatype
datatype,int dest, int tag, MPI_Comm comm);
```

C/C++

where:

`buf` = array of data to be sent
`count` = how many elements are sent
`datatype` = type of data to be sent
`dest` = rank of the receiving process
`tag` = identifier of the message
`comm` = communicator connecting sending and receiving processes
`ierror` = error code

The starting position of the array to be sent must be passed to the sending call.



Receiving calls

The prototype of a receiving call is:

```fortran
integer :: source, status(MPI_STATUS_SIZE)
call  MPI_recv( buf, count, datatype, source, &
      & tag, comm, status, ierror )
```

fortran

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

C/C++

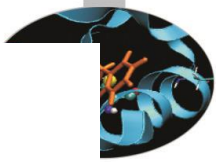
where:

`source` = rank of the sending process

`status` = message info

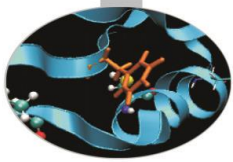
`ierror` = error code

Send and Receive - FORTRAN



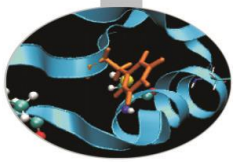
```
PROGRAM send_recv

USE mpi
implicit none

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();
}
```

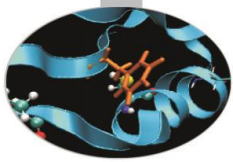
Point to point communications



There are 4 modes of sending data in MPI:

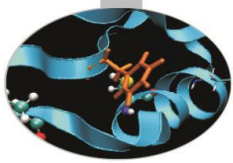
- **Buffered** – Data are copied in a memory area explicitly allocated in the program. Either blocking or nonblocking calls are available, but non blocking calls may lead to problems if the buffer is not large enough to keep all the messages waiting to be sent.
- **Synchronous** – Send operation is considered completed only if the receiving operation has been started, i.e. the receiving processes have provided the memory space needed to copy the sent data. Therefore memory allocation is not an issue because memory buffers are always made available by the sender and the receiver. The problem is that if sending and receiving processes are not synchronized the execution may be slowed down.

Point to point communications



- **Standard** – The operation is automatically managed by the MPI system. If buffered communications are used, memory space is automatically allocated. Again this may lead to memory problems if data sent are too large.

- **Ready** – This mode should be used with care because when the sender starts operation the receiving process must be ready to receive the message. If this is not the case, errors and undefined results are produced. However, if synchronization is granted, this may be the fastest communication mode.



Point to point communications

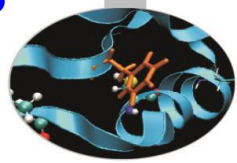
Receiving calls may be blocking or nonblocking only and do not differentiate sending modes.

Summary table

SEND	Blocking	Nonblocking
Standard	MPI_Send	MPI_Isend
Ready	MPI_Rsend	MPI_Irsend
Synchronous	MPI_Ssend	MPI_Issend
Buffered	MPI_Bsend	MPI_Ibsend

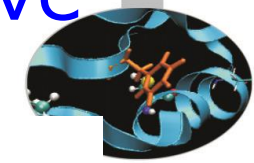
RECEIVE	Blocking	Nonblocking
Standard	MPI_Recv	MPI_Irecv

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 possible. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact that 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.

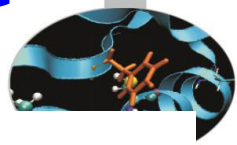
Non-Blocking Send and Receive



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

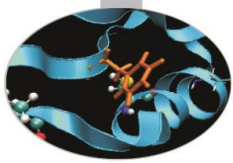


FORTRAN:

```
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)

Point to point communications



The following function stops execution until the data have been safely sent or received:

```
SUBROUTINE MPI_WAIT(REQ, STATUS, IERR)
   INTEGER, INTENT(INOUT) :: REQ
   INTEGER, INTENT(OUT) :: STATUS(MPI_STATUS_SIZE)
   INTEGER, INTENT(OUT) :: IERR
```

fortran

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

C/C++

Where:

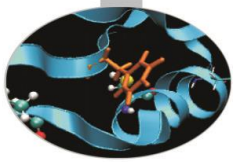
`req` = iSend or iRecv request

`status` = communication status

`ierr` = error code

If req=MPI_REQUEST_NULL nothing is done.

Point to point communications



The following function checks if data have been safely sent or received:

```
SUBROUTINE MPI_TEST(REQ, FLAG, STATUS, IERROR))
   INTEGER, INTENT(INOUT) :: REQ
   LOGICAL, INTENT(OUT) :: FLAG
   INTEGER, INTENT(OUT) :: STATUS(MPI_STATUS_SIZE)
   INTEGER, INTENT(OUT) :: IERR
```

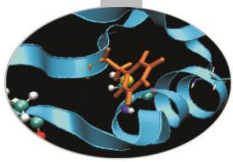
fortran

```
int MPI_Test(MPI_Request *req, int *flag,
      MPI_Status *status)
```

C/C++

Where:

`req` = iSend or iRecv request
`flag` = true if communication has been completed
`status` = communication status
`ierr` = error code



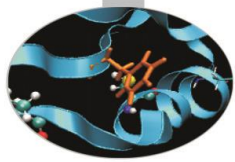
Notes on communications

- A blocking receiving call returns only when the receiving buffer has been completed.

- The nonblocking call `MPI_Irecv` does not return a message status but a message handle `MPI_Request *request`. It can be later used by the function `MPI_Test` to check for communication completion or by the function `MPI_Wait` to wait for completion.

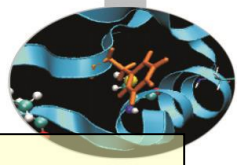
- Message tags and sending processes may be wildcarded using the constant values `MPI_ANY_TAG` and `MPI_ANY_SOURCE` respectively. These may be used to enhance parallel efficiency.

Notes on communications



- On exiting the `status` array will contain useful informations. The array size is `MPI_STATUS_SIZE` and two of the most used infos are:
  - `status(MPI_SOURCE)` = rank of the sender. It may be particularly useful when the sender id is `MPI_ANY_SOURCE`.
  - `status(MPI_TAG)` = message tag. It may be particularly useful when message tag is `MPI_ANY_TAG`.

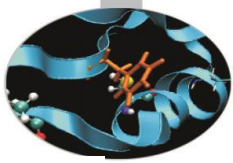
An example



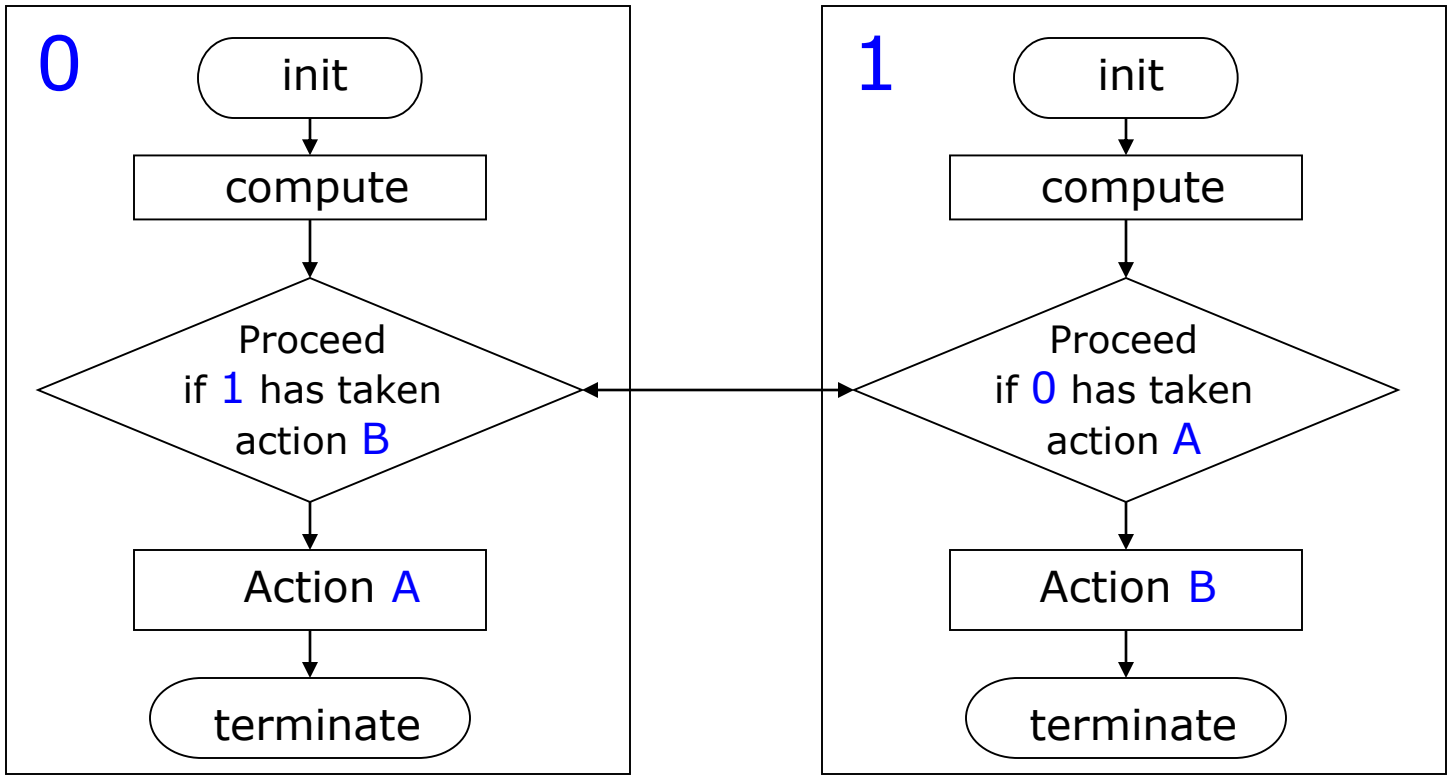
```
integer, dimension (2000) :: box
integer :: error_code, msg_tag=5432, sender=2
integer, dimension (MPI_STATUS_SIZE) :: status
....
call MPI_recv (box(1), 1500, mpi_integer, sender, &
       & msg_tag, MPI_COMM_WORLD, status, error_code)
....
call MPI_recv (box(1501), 500, mpi_integer, &
       & MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
       &  status, error_code)
```

In this example the first 1500 elements of the array box are received from the process with rank 2; the remaining elements are received from whichever the sending process is, without even specifying the message tag.

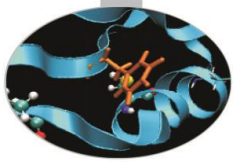
DEADLOCK



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

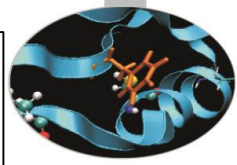


Simple DEADLOCK



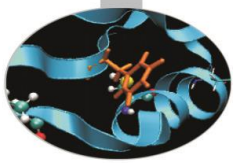
```
PROGRAM deadlock
USE mpi
implicit none
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)

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
USE mpi
Implicit none
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
```

Communications



Sending and receiving may be accomplished by one call only:

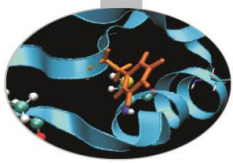
```
type :: SENDBUF, RECVBUF
integer :: SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, &
& RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR

call MPI_SENDRECV( SENDBUF, SENDCOUNT, SENDTYPE, &
       & DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, &
       & SOURCE, RECVTAG, COMM, STATUS, IERROR )
```

fortran

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

C/C++

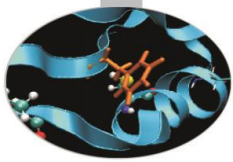
Communications



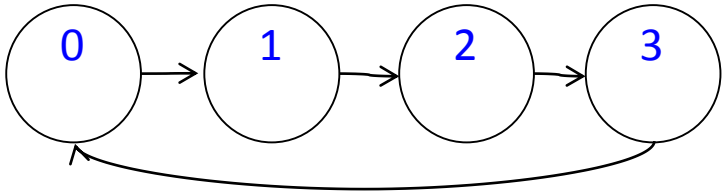
where:

- `SENDBUF` = data buffer to be sent
- `SENDCOUNT` = how many sent elements
- `SENDTYPE` = sent data type
- `DEST` = rank of the receiving process
- `SENDTAG` = sent message tag
- `RECVBUF` = receiving data buffer
- `RECVCOUNT` = how many receiving elements
- `RECVTYPE` = receiving data type
- `SOURCE` = rank of the sending process
- `RECVTAG` = receiving message tag
- `COMM` = communicator
- `STATUS` = message info
- `IERROR` = error code

SendRecv example

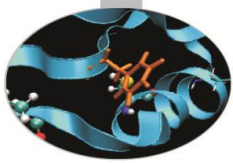


```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int myid, numprocs, left, right,i;
    int buffer[1], buffer2[1];
    MPI_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, 1, MPI_INT, right, 123, buffer2, 1, MPI_INT, left, 123,
  MPI_COMM_WORLD, &status);
  printf("I am processor rank %d and I received the rank of processor
   %d\n",myid,buffer2[0]);
  MPI_Finalize();
 }
```



Useful for cyclic communication patterns

Communications

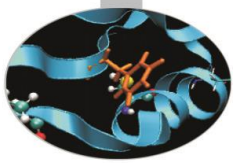


The following function returns how many elements have been received. The number of bytes received is dependent on the received data type.

```
call MPI_get_count (status, datatype, count, ierr)
```
fortran

```
ierror = MPI_get_count (MPI_Status *status,
MPI_Datatype      datatype, int *count )
```
C/C++

Communications



Whenever is necessary to control completion of a lot of communication operations, the following functions may be used instead.

MPI_Waitall does block execution until the operations in LIST_REQUEST are all completed.

MPI_Testall checks if all the operations in LIST_REQUEST are completed (FLAG=.TRUE.).

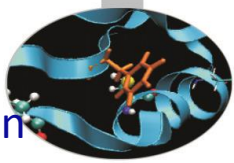
fortran

```
call MPI_waitall(count, list_requests, list_status,
      ierr)
call MPI_testall(count, list_requests, flag, list_status,
      ierr)
```

C/C++

```
ierr = MPI_Waitall (int count, MPI_Request
      list_requests[], MPI_Status list_status[] )
ierr = MPI_Testall (int count, MPI_Request
      list_requests[], int *flag, MPI_Status
      list_status[])
```

Communications



The function **MPI_WAITANY** blocks execution until at least one of the operations in LIST_REQUEST is locally completed.

The function **MPI_TESTANY** checks if at least one of the operations in LIST_REQUEST is locally completed. On output INDEX is the position in LIST_REQUESTS of the completed operation and RETURN_STATUS contains infos about it.

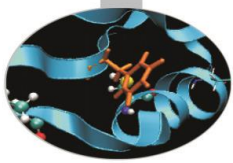
fortran

```
call MPI_waitany(count, list_requests, index,
      return_status, ierr)
call MPI_testany(count, list_requests, index, flag,
      return_status, ierr)
```

C/C++

```
ierr = MPI_Waitany (int count, MPI_Request
      list_requests[], int *index, MPI_Status
      *return_status )
ierr = MPI_Testany(int count, MPI_Request
      list_requests[], int *index, int *flag, MPI_Status
      *return_status )
```

Exercises



http://www.hpc.cineca.it/content/training-mpi

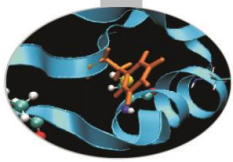
Here you will find some exercises of gradual difficulty, with hints about the functions/routines to be used, solutions in C and Fortran and Q/A in some cases

BE SMART! Exercises are for you to learn. Peeking at solutions shouldn't be the easy way to solve the exercises, but rather a way to compare your ideas with the ones provided by the teachers. Your solution may be better than ours!

For Point-to-point communications, you can do exercises from 1 to 6

When programming with MPI, remember to…think in parallel!

Collective communications

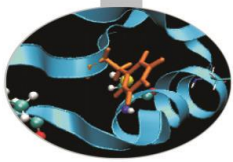


Communications is a very important issue in MPI programs, therefore their optimization is a mandatory effort.

In many cases communications involve a lot of processors and realizing them by point-to-point communications become inefficient and an error prone exertion.

For this reason MPI library contains functions optimized to accomplish collective communications. Therefore using collective communications in such cases is much more effective (and easier) than using point-to-point communications.

Collective communications



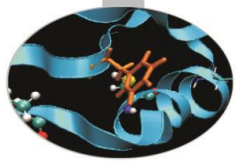
Collective communications do not need tags for messages.

All collective communications are blocking.

Collective communication calls carry out both sending and receiving operations.

The calls to collective communication functions should be issued by all the processes of a given communicator.

Collective communications



Collective communications may be of two types: **data transfer** and **global computations**.

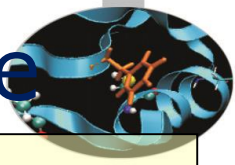
Data transfer functions can be:

- **broadcast** - data are shared among all the processes
- **gather** - data are collected from every process
- **scatter** - data are distributed to the processes

Global computation functions can be:

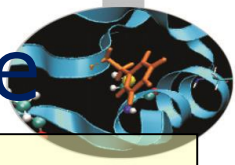
- **reduction** - the result is a computed value
- **scanning** – partial reduction results

Collective communications example



```
subroutine GetData (a, b, n, my_rank, num_procs)
! Point-to-point version
real :: a, b
integer :: n, p, my_rank, num_procs, ierr, tag_a, tag_b, tag_n
include 'mpif.h'

tag_a=1; tag_b=2; tag_n=3;
if (my_rank == 0) then
  print *, 'Enter a, b, and n'
  read *, a, b, n
  do p=1, num_procs-1
    call MPI_Send(a, 1, MPI_REAL, p, tag_a, MPI_COMM_WORLD, ierr)
    call MPI_Send(b, 1, MPI_REAL, p, tag_b, MPI_COMM_WORLD, ierr)
    call MPI_Send(n, 1, MPI_INTEGER, p, tag_n, MPI_COMM_WORLD, ierr)
  enddo
```

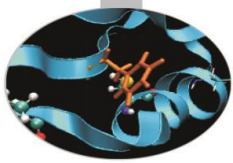
Collective communications example



```
else
  call MPI_Recv(a, 1, MPI_REAL, 0, tag_a, MPI_COMM_WORLD, &
&     status, ierr)
  call MPI_Recv(b, 1, MPI_REAL, 0, tag_b, MPI_COMM_WORLD, &
&     status, ierr)
  call MPI_Recv(n, 1, MPI_INTEGER, 0, tag_n, MPI_COMM_WORLD, &
&     status, ierr)
endif
return
end subroutine GetData
```

6 x (num_procs-1) calls issued for sending 3 scalars

Collective communications



The following function may be used to send the same data to all the processes belonging to a communicator. A loop performing point-to-point communications to all the processes gives the same results but is much less efficient.

fortran

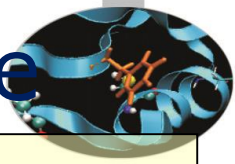
```
type :: array
integer :: count, datatype, root, comm, ierror
call  MPI_BCAST( array, count, datatype, root, comm, ierror )
```

C/C++

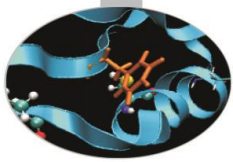
```
ierror = MPI_Bcast( void *buffer, int count, MPI_Datatype
                                  datatype, int root, MPI_Comm comm )
```

where: `array` = data to be sent
`count` = how many elements
`datatype` = data type of the elements
`root` = process owing data to be sent
`comm` = communicator
`ierror` = error code

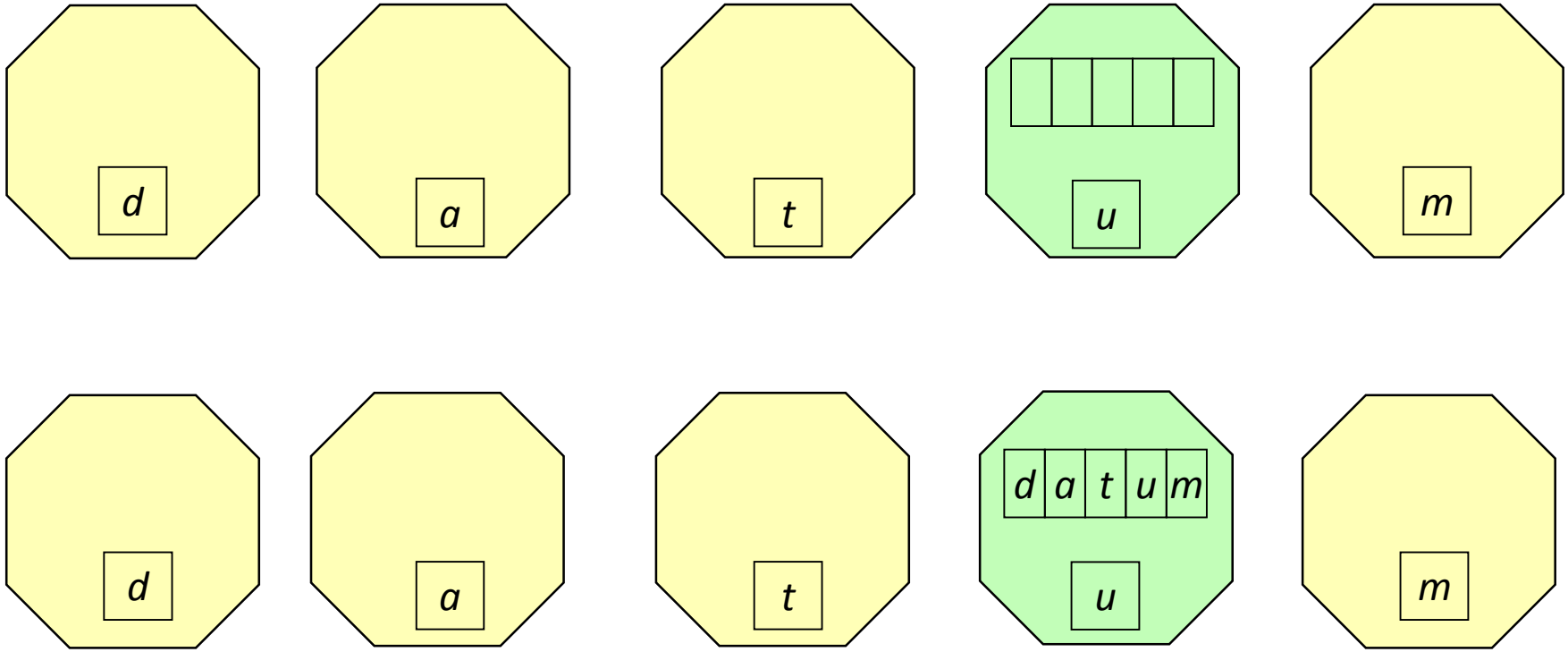
Collective communications example



```
subroutine GetData (a, b, n, my_rank)
! Collective version
real :: a, b
integer :: n, my_rank, ierr
include 'mpif.h'

if (my_rank == 0) then
   print *, 'Enter a, b, and n'
   read *, a, b, n
endif

call MPI_BCAST (a, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (b, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )

end subroutine GetData
```

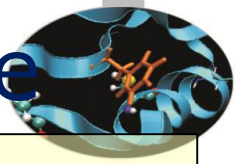


Collective communications



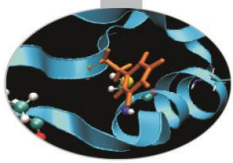
Gather: Every process in the communicator send the content of `send_buf` to the *root* process that receives data and orders them in the `recv_buf` array according to the rank of the sending processes.

Collective communications example



```
subroutine GatherData (my_a, vector_a, my_rank, num_procs)
! Point-to-point version
integer :: p, my_rank, num_procs, ierr, tag_a
real :: my_a, vector_a(num_procs)
include 'mpif.h'
integer :: status(MPI_STATUS_SIZE)

tag_a=1
if (my_rank == 0) then
  vector_a(1)=my_a
  do p=1, num_procs-1
    call MPI_Recv(vector_a(p+1), 1, MPI_REAL, p, tag_a, &
&       MPI_COMM_WORLD, status, ierr)
  enddo
else
  call MPI_Send(my_a, 1, MPI_REAL, 0, tag_a, MPI_COMM_WORLD, ierr)
endif; end subroutine GatherData
```

Collective communications



The following function may be used whenever data dispersed among the processes have to be collected in one ROOT process

where:

SEND_COUNT - how many elements are sent

RECV_COUNT - how many elements have to be received

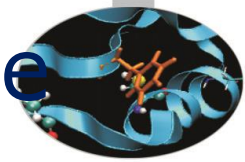
```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, ROOT, &
       & COMM, IERROR, DISP(comm_size)
call MPI_Gather ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
       & RECV_COUNT, RECV_TYPE, ROOT, COMM, IERROR )
```

fortran

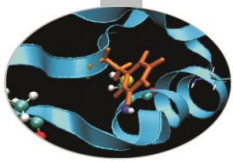
```
ierror = MPI_Gather ( void *send_buf, int send_count, MPI_Datatype
       sendtype, void *recv_buf, int recv_count, MPI_Datatype
       recv_type, int root, MPI_Comm comm )
```

C/C++

Collective communications example



```
subroutine GatherData (my_a, vector_a, num_procs)
! Collective version
 integer :: num_procs, ierr
 real :: my_a, vector_a(num_procs)
 include "mpif.h"

 call MPI_Gather(my_a, 1, MPI_REAL, vector_a, 1, MPI_REAL, &
        &           0, MPI_COMM_WORLD, ierr)

end subroutine GatherData
```



Collective communications

The following function may be used to collect data dispersed among the processes if each process owns a different number of elements.

In the function variants with an ending "v" the array `RECV_COUNT(:)` specify how many elements are stored in each process.

The array `DISP(:)` specifies the position in the receiving buffer where data coming from i^{th} process must be copied. Positions start from 0 even in Fortran.

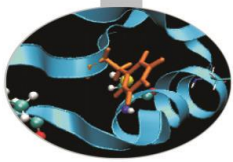
fortran

```
call MPI_Gatherv ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
       & RECV_COUNT, DISP, RECV_TYPE, ROOT, COMM, IERROR )
```

C/C++

```
ierro = MPI_Gatherv ( void *send_buf, int send_count, MPI_Datatype
       send_type, void *recv_buf, int *recv_count,
       int *disp, MPI_Datatype recv_type, int root, MPI_Comm comm )
```

Collective communications

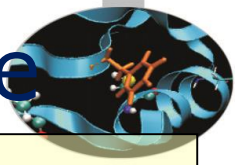


Gathering a dispersed vector when its portions do not have a fixed length is easier (and faster) using MPI_Gatherv.

Of course the length of each portion of the dispersed vector and the position in the global vector should still be known by the gathering process.

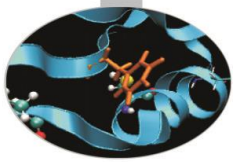
```
subroutine GatherVdata (my_a, l_a, vector_a, l_v, pos, num_proc)
! collective version
integer :: l_a, l_v
integer :: p, num_proc, ierr
real :: my_a(l_a), vector_a(l_v)
integer :: pos(num_proc), n_a(num_proc)
include "mpif.h"
```

Collective communications example

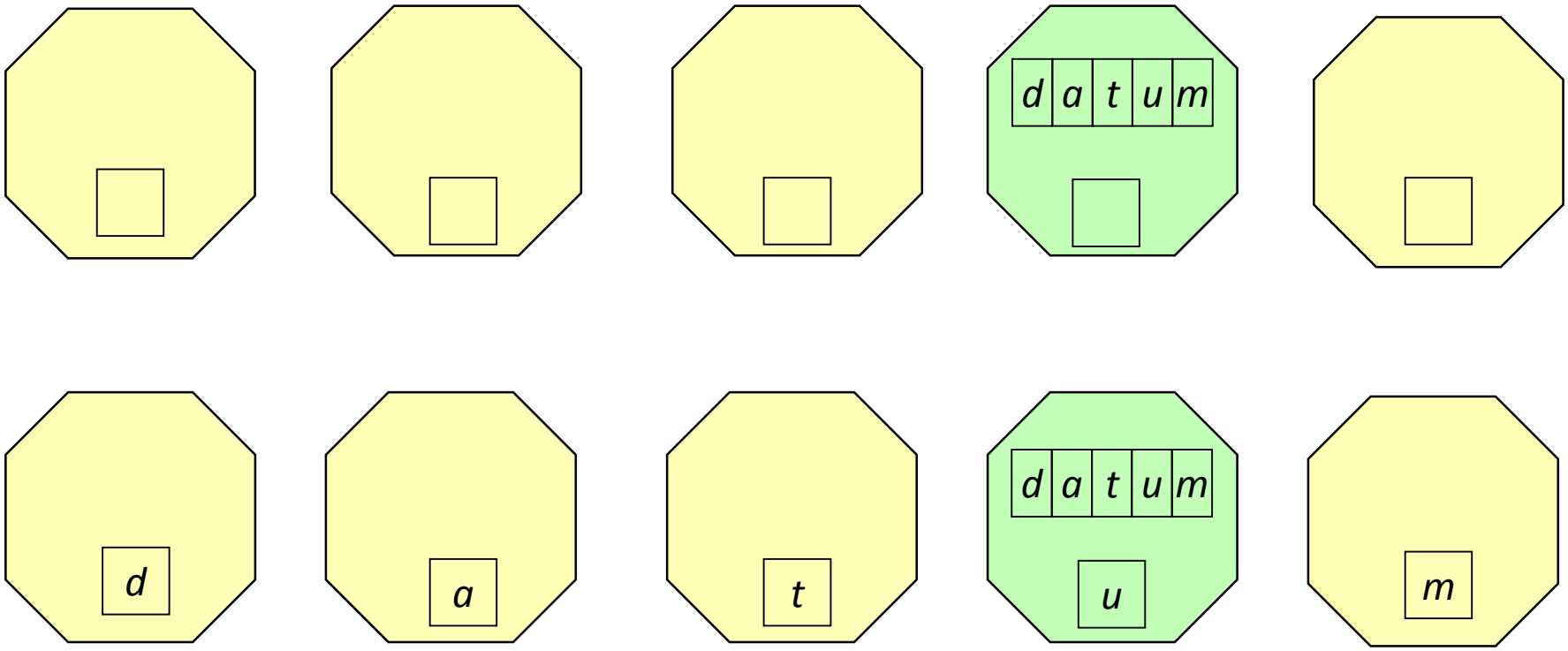


```
! A vector with the length of each portion is needed
do p = 0, num_proc-2
   n_a(p+1) = pos(p+2)-pos(p+1)
enddo
n_a(num_proc) = l_v-pos(num_proc)+1

call MPI_Gatherv ( my_a, l_a, MPI_REAL, vector_a, &
        & n_a, pos, MPI_REAL, 0, MPI_COMM_WORLD, ierr )

end subroutine GatherVdata
```

Remark: the vector pos(:) in this code version must contain the positions of each my_a in vector_a(:) beginning from 0



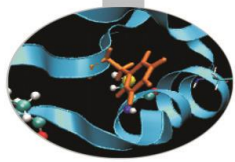
Collective communications



Scatter: the `root` process disperses the content of `send_buf` array to the other processes of the communicator group.

Data are scattered according to the order of the processes.

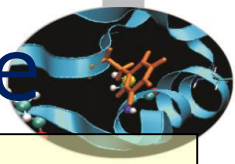
Collective communications



Dispersing a vector from one process to the others requires at least (num_proc - 1) * 2 calls.

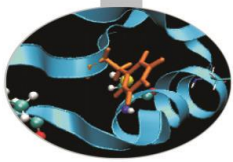
```
subroutine ScatterData (vector_a, l_v, a, l_a, my_rank, num_proc)
! Point to point version
  integer :: l_v, l_a, my_rank, num_proc, tag_a, p, ierr
  real :: vector_a(l_v), a(l_a)
  include "mpif.h"
  integer :: status(MPI_STATUS_SIZE)
```

Collective communications example



```
  tag_a=1
  if (my_rank == 0) then
    do p = 1, num_proc-1
      call MPI_Send(vector_a(l_a*p+1), l_a, MPI_REAL, p, tag_a, &
&                    MPI_COMM_WORLD, ierr)
    enddo
    a(1:l_a)=vector_a(1:l_a)
  else
    call MPI_Recv(a, l_a, MPI_REAL, 0, tag_a, &
&                  MPI_COMM_WORLD, status, ierr)
   endif

end subroutine ScatterData
```

Collective communications



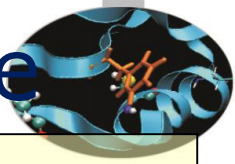
fortran

```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, &
                & RECV_TYPE, ROOT, COMM, IERROR


call MPI_Scatter ( SEND_BUF, SEND_COUNT, SEND_TYPE, &
                RECV_BUF, RECV_COUNT, RECV_TYPE, &
                ROOT, COMM, IERROR )
```

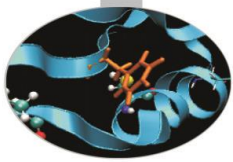
```
ierror = MPI_Scatter ( void *send_buf, int send_count,
      MPI_Datatype send_type,
      void *recv_buf, int recv_count, MPI_Datatype
      recv_type, int root, MPI_Comm comm )
```

C/C++

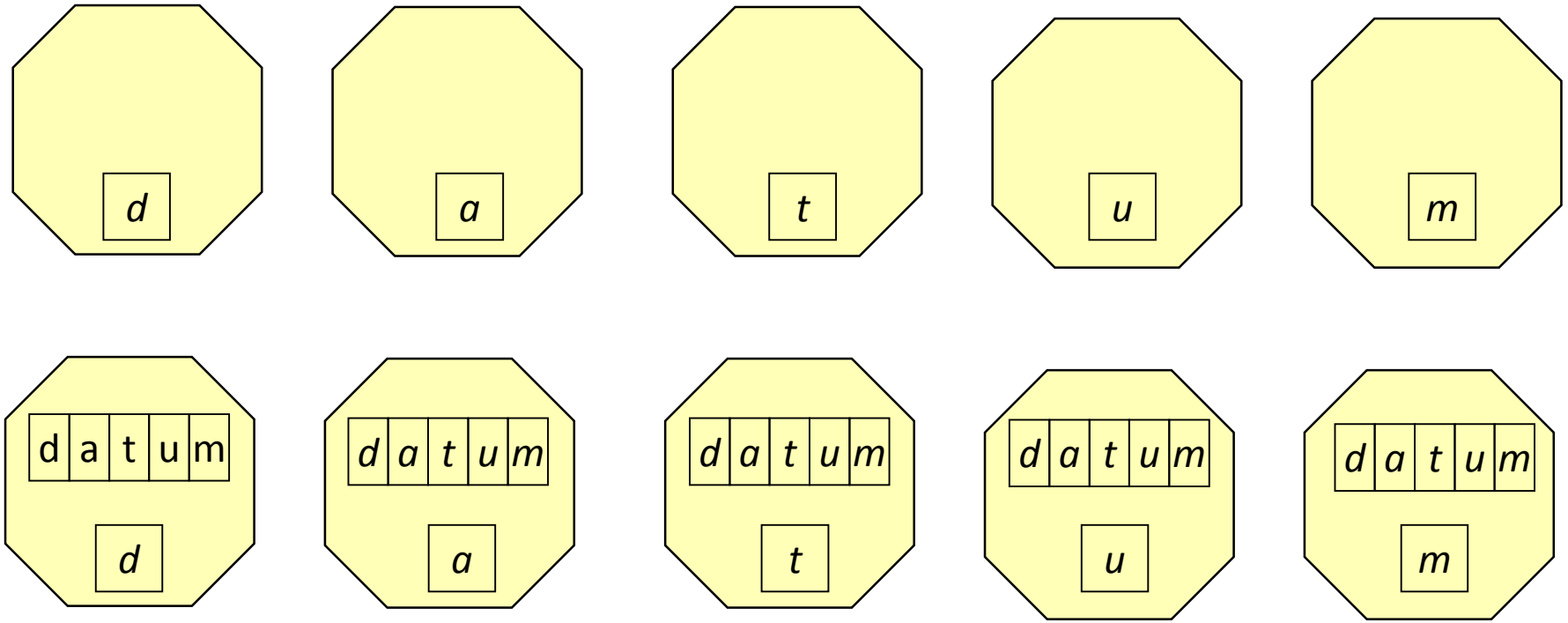
Collective communications example



```
subroutine ScatterData (vector_a, l_v, a, l_a)
! Collective version
   integer :: l_v, l_a
   real :: vector_a(l_v), a(l_a)
   integer :: ierr
   include "mpif.h"

   call MPI_Scatter ( vector_a, l_a, MPI_REAL, &
&             a, l_a, MPI_REAL, 0, MPI_COMM_WORLD, ierr )

end subroutine ScatterData
```

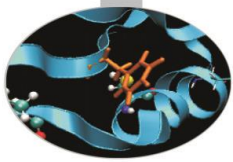


Collective communications



Allgather: Every process in the communicator send the content of `send_buf` to *all the other* processes that receive data and order them in the `recv_buf` array according to the rank of the sender.

Collective communications



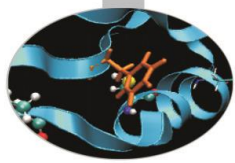
fortran

```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT,  RECV_TYPE,
COMM, IERROR

call MPI_Allgather ( SEND_BUF, SEND_COUNT, SEND_TYPE,
       RECV_BUF, RECV_COUNT, RECV_TYPE, COMM, IERROR )
```

C/C++

```
ierror = MPI_Allgather ( void *send_buf, int send_count,
MPI_Datatype send_type,
void *recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_Comm comm )
```

The above function may be used to collect data from all the processes to all the processes. It is equivalent to a sequence of calls to `MPI_Gather` in which each call identifies a different process as *root*.



Collective communications

Collective communications include global computations, of a reduction type.

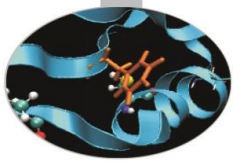
The result of the computations may be:

- stored in one process only
- broadcasted to all the processes
- scattered to all the processes

Three functions are available:

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter

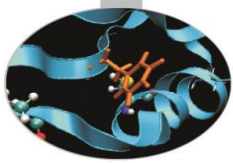
Collective communications



In the following example the quadrature of a function is computed between two extremes.

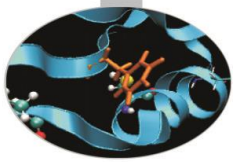
```
program integral
!  Point to point version
        .       .       .
!  Domain decomposition and computation
   x_intrvl = (x1-x0)/num_procs
   my_x0 = my_rank*x_intrvl; my_x1 = my_x0+x_intrvl
   my_n = 0; x = my_x0; my_s = 0.0
   do while ( x < my_x1 )
      my_s = my_s + f(x)
      my_n = my_n + 1
      x = x + dx
   enddo
```

Collective communications



```
! Gather results and sum them up
   tag_s=10; tag_n=20
   if ( my_rank == 0 ) then
      s = my_s; n = my_n
      do p = 1, num_procs-1
         call MPI_Recv(my_s, 1, MPI_DOUBLE_PRECISION, p, tag_s, &
&              MPI_COMM_WORLD, status, ierr); s = s + my_s
      enddo
   else
      call MPI_Send(my_s, 1, MPI_DOUBLE_PRECISION, 0, tag_s, &
&              MPI_COMM_WORLD, ierr)
   endif

  if (my_rank==0) write(6,*)"After ",n," steps result=",s/dble(n)
end program integral
```

Collective communications



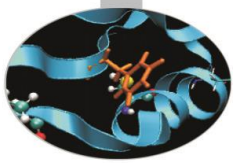
The following function may be used to compute reduction operations such as sum, product, logical, min/max and others. It must be called by all the processes of the communicator `comm`. The result is stored in the process identified as `root`. If `count>1` then `send_buf` and `recv_buf` are arrays and the computation is executed element by element.

fortran

```
type :: send_buf, recv_buf
integer :: count, datatype, op, root, comm, ierror
call MPI_REDUCE ( send_buf, recv_buf, count, &
                  datatype, op, root, comm, ierror )
```

C/C++

```
ierror = MPI_Reduce ( void *send_buf, void *recv_buf,
                int count, MPI_Datatype datatype,
                MPI_Op op, int root, MPI_Comm comm )
```



Collective communications

`send_buf` = data to be used for computation, operands

`recv_buf` = result buffer (received by root process only)

`count` = buffer size

`datatype` = type of the elements

`op` = reduction operation (es.: MPI_SUM, MPI_MAX, …)

`root` = which process stores the results

`comm` = communicator

`ierror` = error code

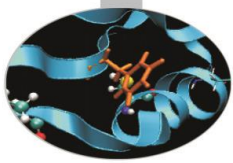
As an example, if `count=3` and `op=MPI_SUM`, then:

Recv_buf(0) = $send\_buf_{proc0}(0) + \ldots + send\_buf_{procN-1}(0)$

Recv_buf(1) = $send\_buf_{proc0}(1) + \ldots + send\_buf_{procN-1}(1)$

Recv_buf(2) = $send\_buf_{proc0}(2) + \ldots + send\_buf_{procN-1}(2)$

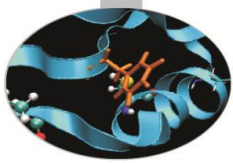
Collective communications



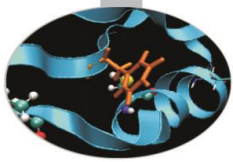
The available operations are:

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	maximum value and location
MPI_MINLOC	minimum value and location

Collective communications



```
program integral_c
!  Collective version
        .       .       .
!  Domain decomposition and computation
   x_intrvl = (x1-x0)/num_procs
   my_x0 = my_rank*x_intrvl; my_x1 = my_x0+x_intrvl
   my_n = 0; x = my_x0; my_s = 0.0
   do while ( x < my_x1 )
      my_s = my_s + f(x)
      my_n = my_n + 1
      x = x + dx
   enddo
   call MPI_REDUCE ( my_s, s, 1, MPI_DOUBLE_PRECISION, &
&                             MPI_SUM, 0, MPI_COMM_WORLD, ierr )
   if (my_rank==0) write(6,*)"After ",n," steps result=",s/dble(n)
end program integral_c
```



Collective communications

fortran

```
type :: OPERAND(*), RESULT(*)
integer :: COUNT, DATATYPE, OP ,COMM, IERROR
call MPI_AllReduce ( OPERAND, RESULT, COUNT, DATATYPE, OP,
      COMM, IERROR )
```

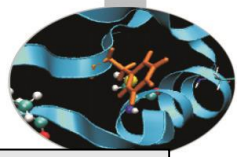
C/C++

```
ierror = MPI_Allreduce ( void *operand, void *result, int
      count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

This function differs from the previous one because the operation result is stored in all the processes of the communicator `comm`.

Mpi_AllReduce = MPI_Reduce + MPI_Bcast

Collective communications



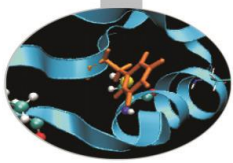
```
<type>, IN :: SENDBUF(*)
<type>, OUT :: RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR


call MPI_REDUCE_SCATTER (SENDBUF, RECVBUF, RECVCOUNTS,
   DATATYPE, OP, COMM, IERROR)
```

fortran

```
ierror = MPI_Reduce_scatter ( void *sendbuf, void *recvbuf,
      int *recvcounts, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm )
```

C/C++

Using the function `MPI_Reduce_scatter`, the reduction result is first computed element by element, then the obtained vector is split into disjoined segments and dispersed to all the processes. The array `recvcounts(:)` is used to specify how many elements each process will store.



Process synchronization

Whenever it is necessary that all the processes come to a determined point at the same time, then synchronization barriers must be used. To avoid heavy loss of performances barriers should be used with care and only if it is unavoidable, i.e. the implemented algorithm requires it.

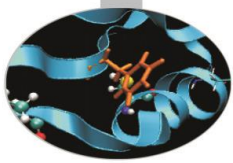
The following function can be used to define a synchronising point:

```
integer :: comm, ierror
call MPI_BARRIER ( comm, ierror )
```
fortran

```
ierror = MPI_Barrier ( MPI_Comm comm )
```
C/C++

Where: `comm` – communicator whose processes must be synchronized
`ierror` – error code.

This function returns only after all the processes have called it.



Performance evaluation

It is often useful to measure computing time of portions of the program. The following functions may be used. Both functions return a double floating point value.

fortran

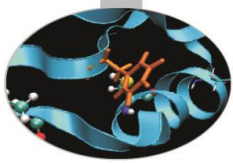
```
REAL(8) :: t1, t2, elapsed
t1 = MPI_WTIME ( )
…
t2 = MPI_WTIME ( )
elapsed = t2 - t1
```

C/C++

```
double t1, dt
t1 = MPI_Wtime()     /* elapsed time in seconds */
dt = MPI_Wtick()     /* time resolution in seconds */
```

Time values are process dependent unless MPI_WTIME_IS_GLOBAL is defined and its value is .TRUE..

Exercises



http://www.hpc.cineca.it/content/training-mpi

After having learned about collective communications, you have «unlocked» exercises from 7 to 9

If you have not finished the point-to-point yet, you are free to keep working on them. Teacher's suggestion is to try to do some collective exercises in any case, just to fix the concepts.

When programming with MPI, remember to…think in parallel!