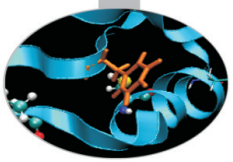
CINECA SCAI
SuperComputing Applications and Innovation



Introduction to MPI

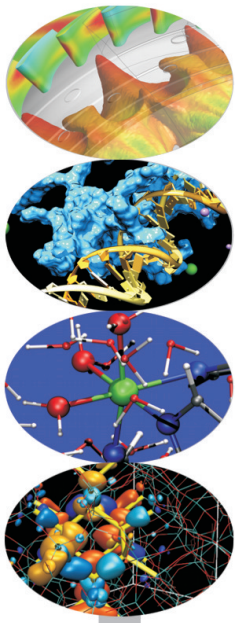
Isabella Baccarelli - i.baccarelli@cineca.it

Mariella Ippolito - m.ippolito@cineca.it

Cristiano Padrin - c.padrin@cineca.it

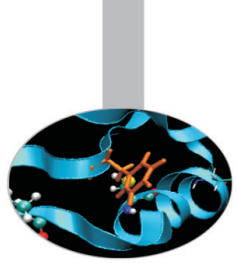
Vittorio Ruggiero - v.ruggiero@cineca.it

SuperComputing Applications and Innovation Department

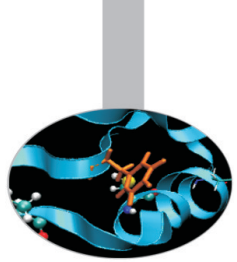


CINECA

Outline



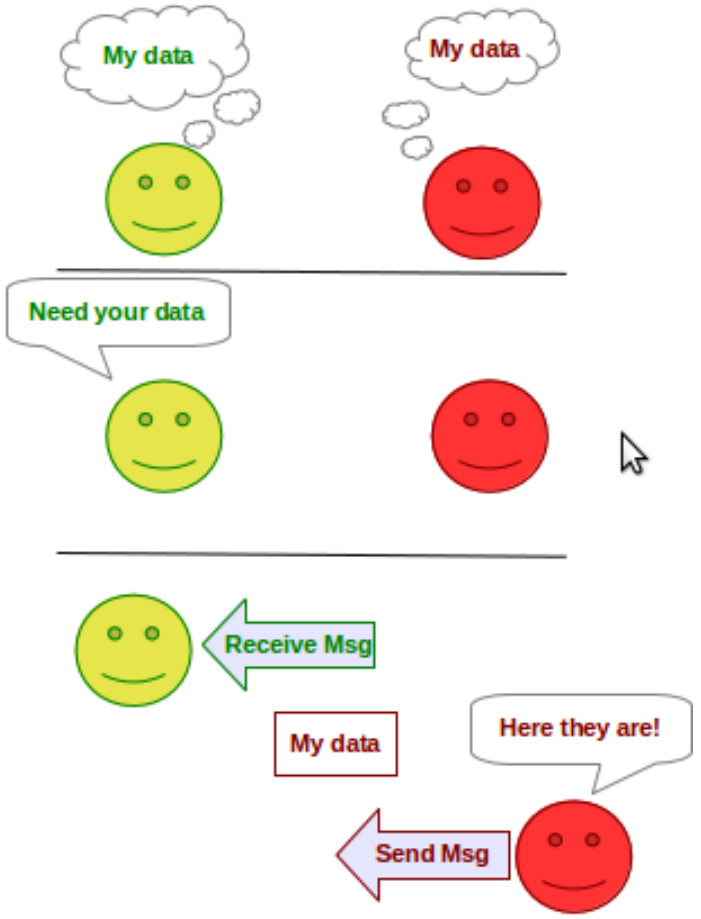
Introduction to the Parallel Computing



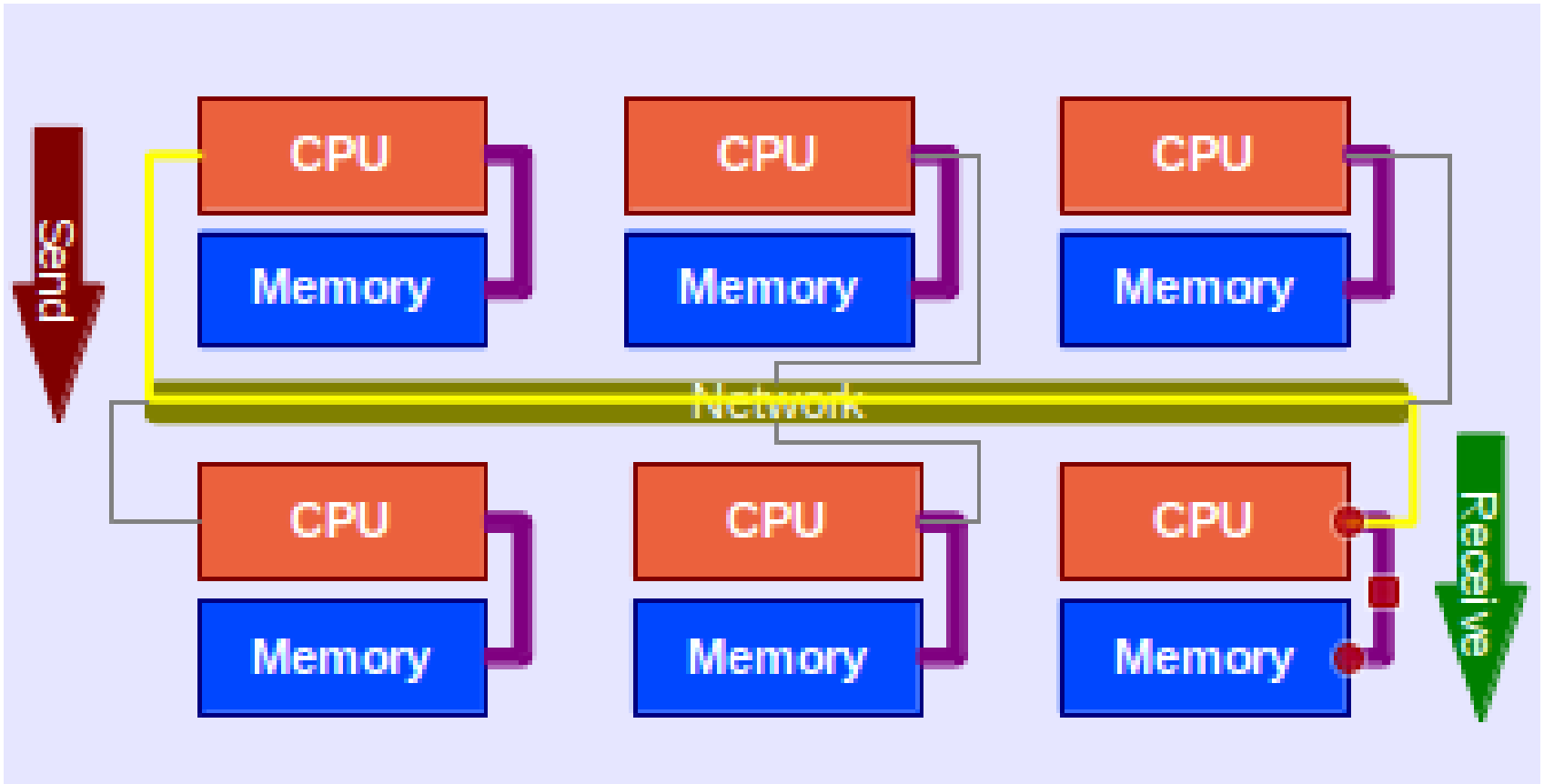
- The Parallel Computing is:
  - the way to solve big data problems exceeding the limits of memory and reducing the compute time;
  - usually, the synchronised usage of more processes to solve computational problems.
- To run with more processes, a problem must be divided in more discrete portions that can be solved concurrently.
- The instructions of each portion are executed simultaneously by different processes.

Message Passing

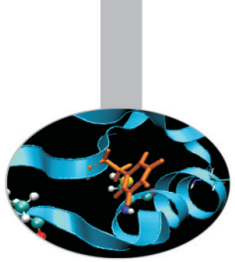
- N processes cooperate exchanging messages.
- Each process:
  - works alone the indipendent portion of the target;
  - has is own memory area;
  - accesses only data available in its own memory area.
- The exchange between processes is needed if:
  - a process must access to data resident on a memory area of another one;
  - more processes have to be synchronized to execute the instructions flow.



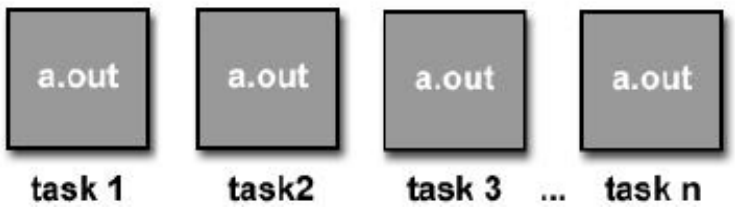
Message Passing on a computer



Execution model SPMD



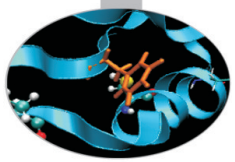
- SPMD is the acronym for Single Program Multiple Data.
- SPMD is the execution model of a Parallel Computing where:
  - each process execute the same program working with different data on its own memory area;



  - different processes can execute different portions of the code.

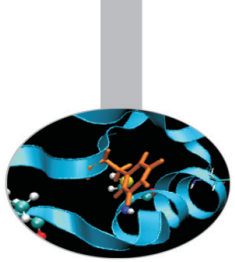
```
if (I am process 1)
  … do something …
if (I am process 2)
  … do something else …
```

What is MPI?

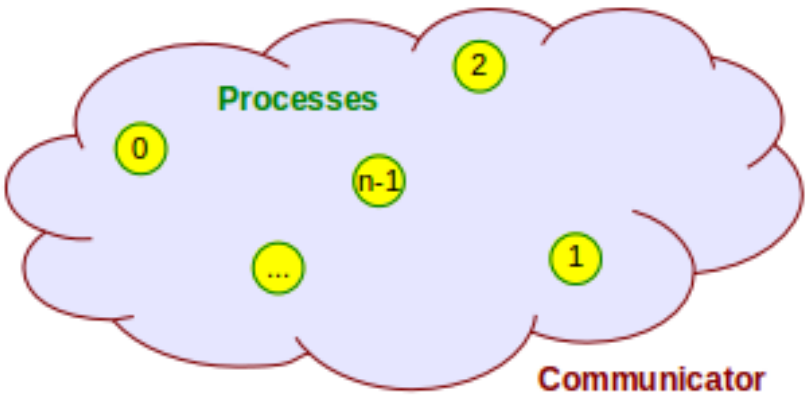


- MPI is the acronym for Message Passing Interface.
- MPI is an Application Programming Interface.
- MPI is a standard for developers and users.
- The MPI libraries allow:
  - management functions for the communication:
    - definition and identification of groups of processes;
    - definition and management of the identity of the single process;
  - functions for the exchange of messages:
    - send and/or receive data from a process;
    - send and/or receive data from a group of processes;
  - new data types and constants (macro) to help the programmer.

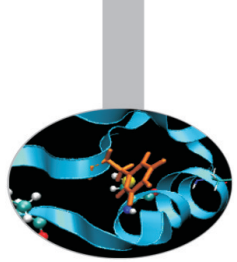
Parallel Computing with MPI



- MPI allows:
  - create and manage a group of processes;
  - exchange data between processes or groups of processes, by a communicator.

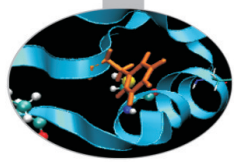


How introduce MPI in a code



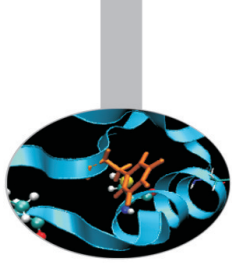
- All programs must include a header file:
  - C/C++: #include <mpi.h>;
  - Fortran77: include mpif.h;
  - Fortran90: use mpi;
    - Fortran90 with MPI3.0: use mpi_f08.
- In the header file there is all you need for the compilation of a MPI program:
  - definitions;
  - macros;
  - prototypes of functions.
- MPI maintains internal data structures related to communication, referenceable via MPI Handles.
- MPI references the standard data types of C/Fortran through MPI Datatype.

How to add the MPI calls

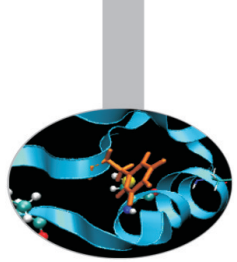


- C/C++:
  - ierr = MPI_Xxxxx (parameter, ...);
  - MPI_Xxxxx (parameter, ...);
  - MPI_ is the prefix of all functions MPI.
  - After the prefix, the first letter is uppercase and all other lowercase.
  - Practically all functions MPI return an error code (integer).
  - Macros are written with uppercase.
- Fortran:
  - call MPI_XXXXX (parameter, ..., IERR)
    - Fortran90 with MPI3.0: call MPI_XXXXX (parameter, ...)
  - MPI_ is the prefix of all functions MPI
  - The last parameter (IERR) is the error code returned (integer);
    - if use mpi, NEVER FORGET the IERR parameter;
    - if use mpi_f08, IERR parameter is optional.

Outline



Some problems



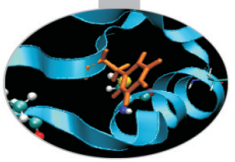
Problem 1 - Fibonacci series

$$f_1 = 1$$
$$f_2 = 1$$
$$f_i = f_{i-1} + f_{i+2} \qquad \forall i > 2$$

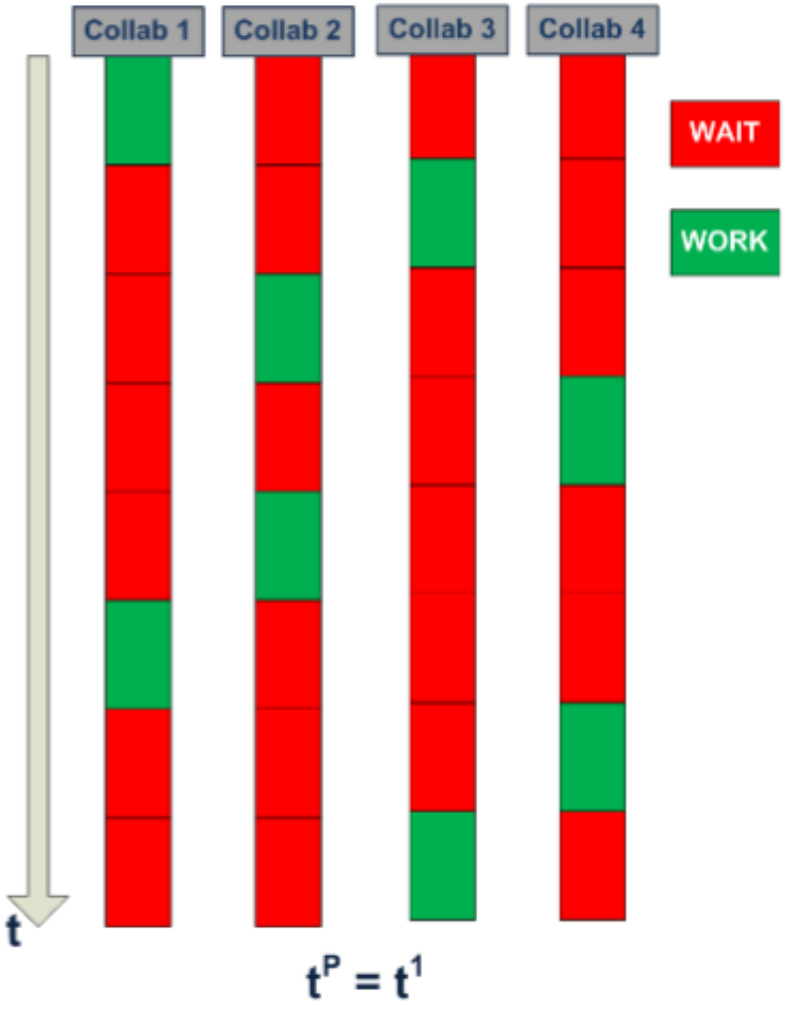
Problem 2 - Geometric series

$$g_N = \sum_{i=1}^{N} x^i$$

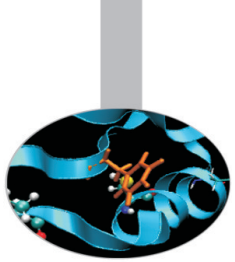
Fibonacci series



- If we try to use MPI to solve the problem 1, we can observe:
  - f_i depends by f_{i-1} and f_{i-2}, and can't be calculated without;
  - a process cannot compute f_i simultaneously to the computation of f_{i-1} or f_{i+1};
  - the execution time is the same of the serial case.



Geometric series

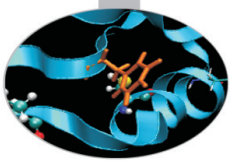


Series

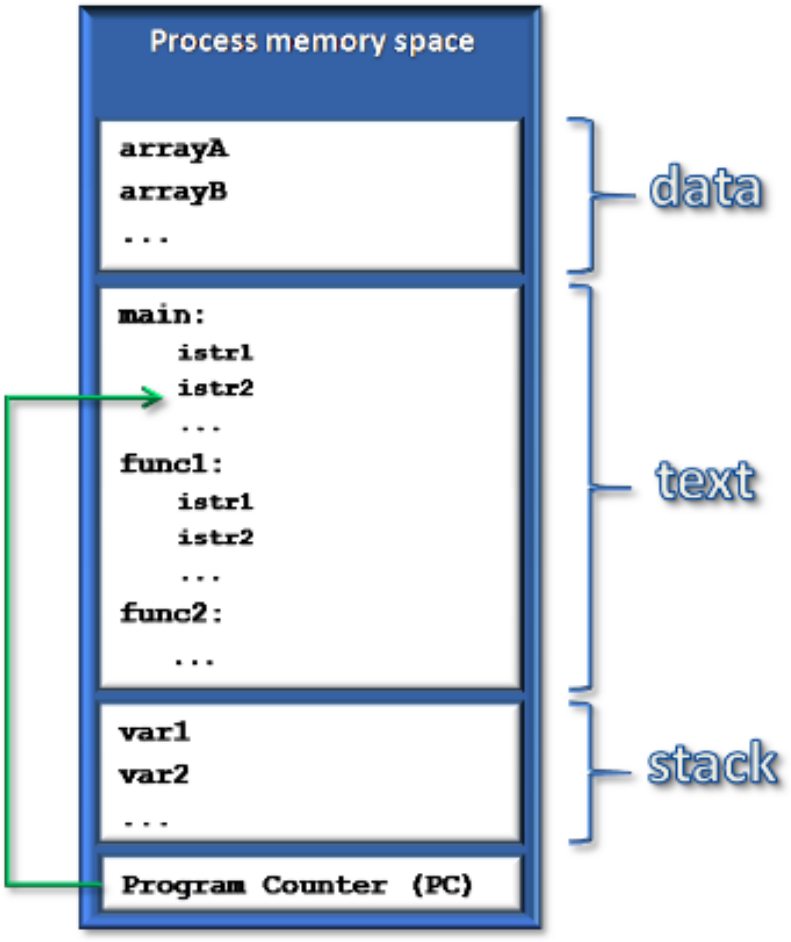
$$g_N = \sum_{i=1}^{P} \left(\sum_{j=1}^{N/P} x^{\frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^{P} S_i$$

- If we try to use MPI to solve the problem 2, we can observe:
  - each process calculates one of the P partial sums S_j;
  - only one of the processes collects the P partial sums;
  - the execution time is $\frac{1}{P}$ of the time of the serial case.

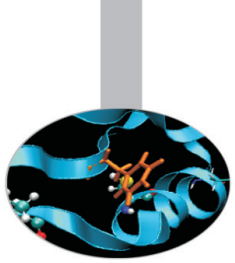
How one process works



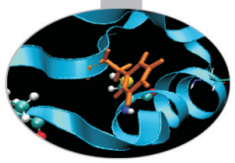
- A process is an instance on execution of a program;
- a process keeps in memory data and instructions of the program, and other informations needed to control the execution flow.



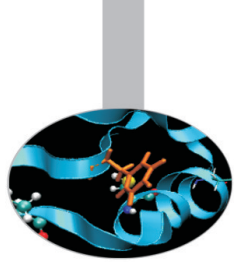
Outline



Outline

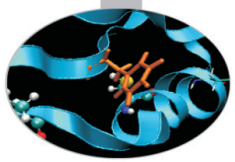


Create a parallel region



- MPI_Init
  - initializes the communication environment;
  - all MPI programs need a call to it;
  - can be introduced once only in the whole code;
  - must be call before other calls to MPI functions.
- MPI_Finalize
  - ends the communication phase;
  - all MPI programs need at least a call to it;
  - provides a cleaned release of the communication environment;
  - must be call after all other calls to MPI functions.

Sintax



C/C++

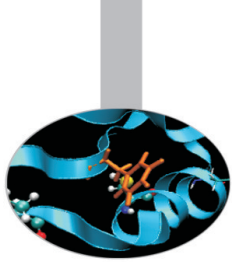
```
int MPI_Init(int *argc, char **argv)
int MPI_Finalize(void)

MPI_Init(&argc, &argv);
MPI_Finalize();
```

With MPI2.0 and higher, also:

```
MPI_Init(null, null);
```

- NOTE:
  - the MPI_Init function does the parsing of the arguments supplied to the program from the command line.

Sintax



Fortran

```
MPI_INIT(IERR)
MPI_FINALIZE(IERR)
```

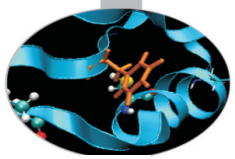
With mpif.h & mpi:

```
INTEGER ::  IERR

CALL MPI_INIT(IERR)
CALL MPI_FINALIZE(IERR)
```

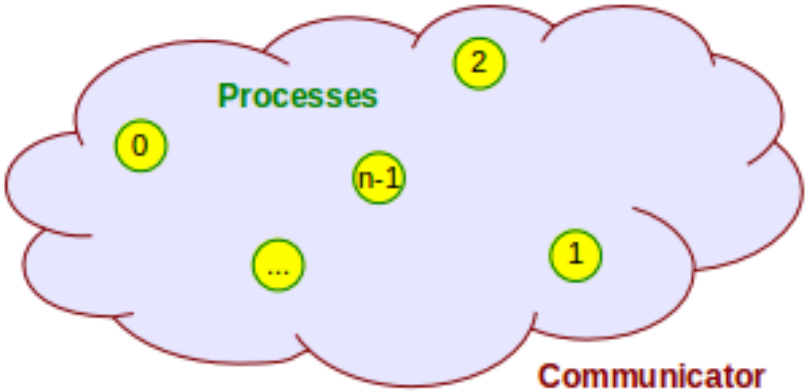
With mpi_f08:

```
INTEGER, OPTIONAL ::  IERR

CALL MPI_INIT()
CALL MPI_FINALIZE()
```

Communicators

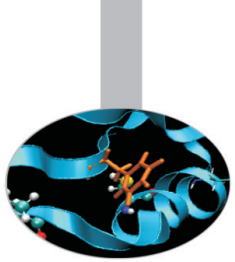


- A communicator is an "object" that contains a group of processes and a set of linked features.
- In a communicator, each process has a unique identification number.



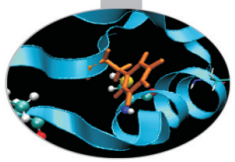
  - Two or more processes can communicate only if they are contained in the same communicator.
  - The MPI_Init function initializes the default communicator: *MPI_COMM_WORLD*.
  - The *MPI_COMM_WORLD* contains all processes that contribute to the parallel code.
  - In an MPI program it's possible to define more than one communicator.

Size & Rank



- *Communicator Size*:
  - the number of processes contained in a communicator is its **`size`**.
  - a process can estimate the size of its own communicator with the function MPI_Comm_size;
  - the **`size`** of a communicator is an *integer*.
- *Process Rank*:
  - a process can estimate its own identification number with the function MPI_Comm_rank;
  - the ranks are *integer* and consecutive numbers from `0` to **`size-1`**.

Sintax



C/C++

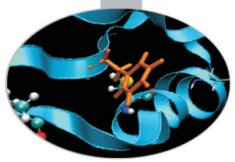
```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran

```
MPI_COMM_SIZE(comm, size, ierr)
MPI_COMM_RANK(comm, rank, ierr)
```

- *Input*:
  - **comm** has type *MPI_Comm* (INTEGER), and it is the communicator;
- *Output*:
  - **size** has type *int* (INTEGER);
  - **rank** has type *int* (INTEGER).

Sintax



C/C++

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Fortran

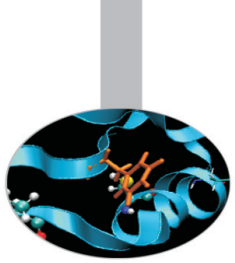
With mpif.h & mpi:

```
INTEGER ::  size, rank, ierr
INTEGER ::  MPI_COMM_WORLD     (optional)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
```

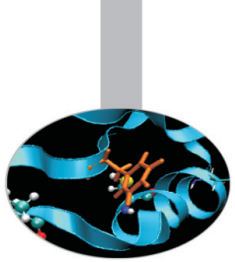
With mpi_f08:

```
INTEGER ::  size, rank
INTEGER, OPTIONAL ::  ierr
TYPE(MPI_Comm) ::  MPI_COMM_WORLD     (optional)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
```

Outline

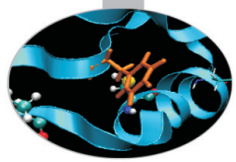


Hello!



- This first program prints the unique identification number of each process and the size of the communicator. These are the needed operations:
  - initialize the MPI environment;
  - ask to the default communicator the rank of each process;
  - ask to the default communicator its own size;
  - print one string with these two informations;
  - close the MPI environment.

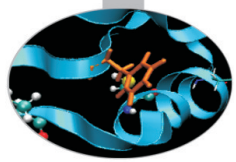
Hello!



C

```c
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[]) {
    int rank, size;
    /* 1.  Initialize MPI */
    MPI_Init(&argc, &argv);
    /* 2.  Get process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* 3.  Get the total number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* 4.  Print rank and size */
    printf("Hello!  I am %d of %d \n", rank, size);
    /* 5.  Terminate MPI */
    MPI_Finalize();
 }
```

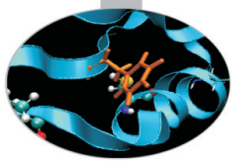
Hello!



Fortran with mpif.h

```
PROGRAM Hello

INCLUDE 'mpi.f'
INTEGER rank, size, ierr

    !  1.  Initialize MPI
    CALL   MPI_Init(ierr)

    !  2.  Get process rank
    CALL   MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

    !  3.  Get the total number of processes
    CALL   MPI_Comm_size(MPI_COMM_WORLD, size, ierr);

    !  4.  Print rank and size
    PRINT*   "Hello!  I am ", rank, " of ", size

    !  5.  Terminate MPI
    CALL   MPI_Finalize(ierr)
END
```

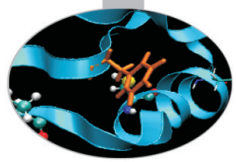
Hello!



Fortran with mpi

```
PROGRAM Hello
USE mpi
INTEGER rank, size, ierr
    !  1.  Initialize MPI
    CALL   MPI_Init(ierr)
    !  2.  Get process rank
    CALL   MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    !  3.  Get the total number of processes
    CALL   MPI_Comm_size(MPI_COMM_WORLD, size, ierr);
    !  4.  Print rank and size
    WRITE(*,*)   "Hello!  I am ", rank, " of ", size
    !  5.  Terminate MPI
    CALL   MPI_Finalize(ierr)
END PROGRAM Hello
```

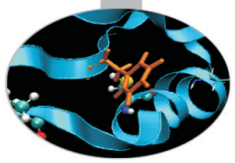
Hello!



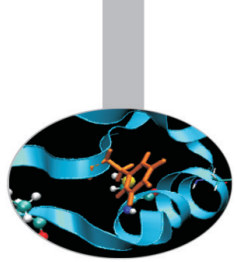
Fortran with mpi_f08

```
PROGRAM Hello
USE mpi_f08
INTEGER rank, size
    !  1.  Initialize MPI
    CALL   MPI_Init()
    !  2.  Get process rank
    CALL   MPI_Comm_rank(MPI_COMM_WORLD, rank)
    !  3.  Get the total number of processes
    CALL   MPI_Comm_size(MPI_COMM_WORLD, size);
    !  4.  Print rank and size
    WRITE(*,*)   "Hello!  I am ", rank, " of ", size
    !  5.  Terminate MPI
    CALL   MPI_Finalize()
END PROGRAM Hello
```

Outline

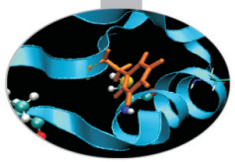


Message Passing



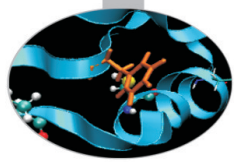
- In parallel programming, the processes cooperate through explicit operations of communication *interprocess*.
- The basic operation of communication is the **`point-to-point`**:
  - a *sender* process *sends a message*;
  - a *receiver* process *receive the message* sent.
- A message contains a number of elements of some particular datatypes.
- MPI datatypes:
  - basic datatype;
  - derived datatype.
- Derived datatypes can be built from basic or other derived datatypes.
- C datatypes are different from Fortran datatypes.

What is a message



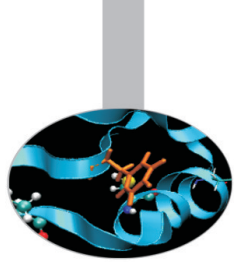
- A message is a block of data that has to be transferred between processes.
- A message is composed by an *Envelope* and a *Body*;
  - Envelope, can contain:
    - **`source`** - rank of the sender process;
    - **`destination`** - rank of receiver process;
    - **`communicator`** - communicator where the message and the processes are;
    - **`tag`** - identification number to classify the message.
  - Body, contains:
    - **`buffer`** - message data;
    - **`datatype`** - type of data that the message contains;
    - **`count`** - how many data of **`datatype`** the message contains.

MPI Datatype for C



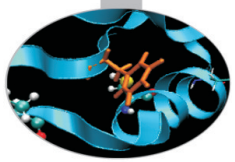
MPI datatype	C datatype
MPI_CHAR	`char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)

MPI Datatype for C



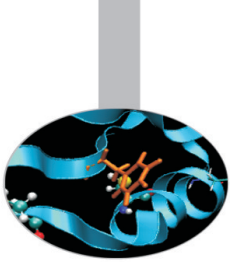
MPI_C_BOOL	`_Bool`
MPI_INT8_T	`int8_t`
MPI_INT16_T	`int16_t`
MPI_INT32_T	`int32_t`
MPI_INT64_T	`int64_t`
MPI_UINT8_T	`uint8_t`
MPI_UINT16_T	`uint16_t`
MPI_UINT32_T	`uint32_t`
MPI_UINT64_T	`uint64_t`
MPI_C_COMPLEX	`float _Complex`
MPI_C_FLOAT_COMPLEX (as a synonym)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	`long double _Complex`
MPI_BYTE	
MPI_PACKED	

MPI Datatype for Fortran

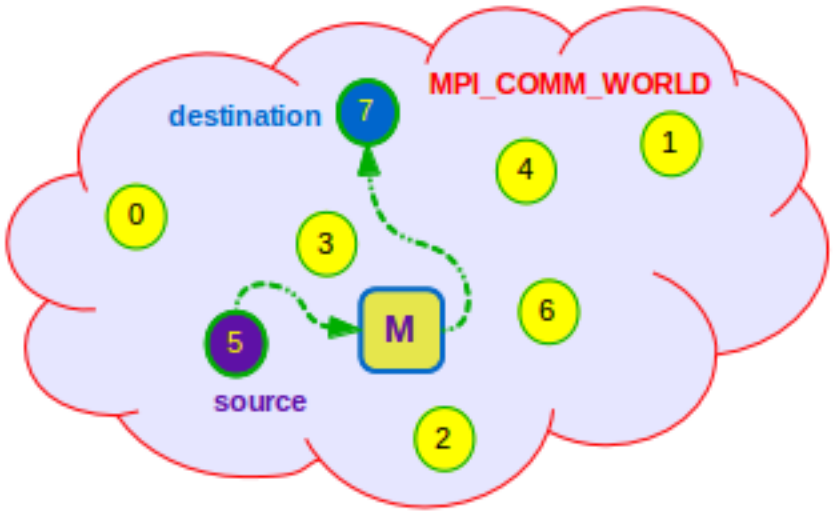


MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

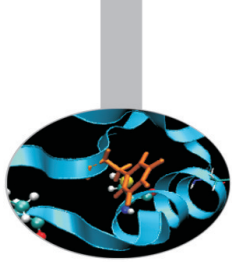
Steps of Point-to-Point



- Send a message:
  - the sender process (*SOURCE*) calls an MPI function;
  - in this function, the rank of the destination process must be declared.
- Receive a message:
  - the receiver process (*DESTINATION*) calls an MPI function;
  - in this function, the rank of sender process must be declared.

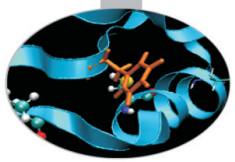


Send a message



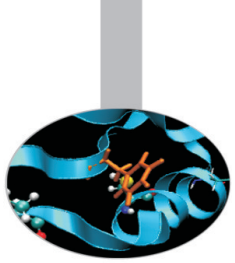
- The **`source`** process calls an MPI function where specifies univocally the *envelope* and the *body* of the message that has to send:
  - the identity of the **`source`** is implicit;
  - the elements which complete the message (identification number of the message, destination identity, communicator for the sending) are defined explicitly by arguments that the process passes to the function for the *send*ing.

Receive a message

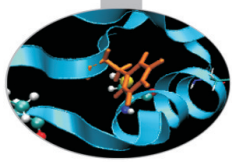


- The **`destination`** process calls an MPI function where is specified univocally the *envelope* (by the *tag*) of the message that has to be received;
- MPI compares the envelope of the message in receiving with the others message that have to be yet received (*pending messages*) and if the message is on hand, this is received; otherwise, the receiving operation can't be completed until a message with the envelope requested will be in the pending messages.

Outline



MPI_Send



C/C++

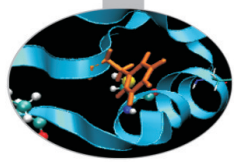
```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
             int dest, int tag, MPI_Comm comm)
```

Fortran

```
MPI_SEND(buf, count, dtype, dest, tag, comm, ierr)
```

- *Input* arguments:
  - buf is the initial address of the *send* buffer;
  - count is the number of elements of the *send* buffer (**integer**);
  - dtype is the type of every element of the *send* buffer (**MPI_Datatype**);
  - dest is the *rank* of the receiver in the communicator *comm* (**integer**);
  - tag is the identity number of the message (**integer**);
  - comm is the communicator where is the *send* (**MPI_Comm**);
- *Output* arguments:
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

MPI_Recv



C/C++

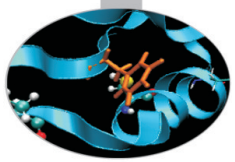
```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,
             int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran

```
MPI_RECV(buf, count, dtype, src, tag, comm, status, ierr)
```

- *Input* arguments:
  - count is the number of elements of the *receive* buffer (**integer**);
  - dtype is the type of every element of the *receive* buffer (**MPI_Datatype**);
  - src is the *rank* of the sender in the communicator *comm* (**integer**);
  - tag is the identity number of the message (**integer**);
  - comm is the communicator where is the *send* (**MPI_Comm**);
- *Output* arguments:
  - buf is the initial address of the *receive* buffer;
  - status contains informations about the received message (**MPI_Status**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int count, dest, src, tag;
MPI_Status status;
MPI_Send(&buf, count, dtype, dest, tag, MPI_COMM_WORLD);
MPI_Recv(&buf, count, dtype, src, tag, MPI_COMM_WORLD, &status);
```

Fortran

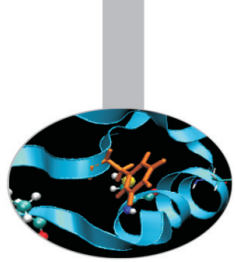
With mpif.h & mpi:

```
INTEGER ::  count, dest, src, ierr (,dtype, MPI_COMM_WORLD)
INTEGER ::  status(MPI_STATUS_SIZE)

CALL MPI_SEND(buf, count, dtype, dest, tag, MPI_COMM_WORLD,
              ierr)
CALL MPI_RECV(buf, count, dtype, src, tag, MPI_COMM_WORLD,
              status, ierr)
```

With mpi_f08:

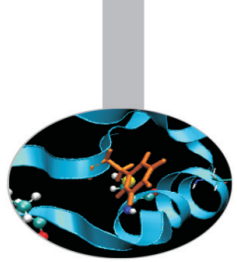
```
INTEGER ::  count, dest, src, tag   TYPE(MPI_Datatype) ::  dtype
TYPE(MPI_Comm) ::  MPI_COMM_WORLD   TYPE(MPI_Status) ::  status

CALL MPI_SEND(buf, count, dtype, dest, tag, MPI_COMM_WORLD)
CALL MPI_RECV(buf, count, dtype, src, tag, MPI_COMM_WORLD,
              status)
```

What is `status`



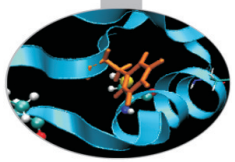
- When the length of received message (*recv_count*) is different if compaired with the lenght of the sent message (*send_count*):
  - if *send_count* > *recv_count*, an overflow error will returned;
  - if *send_count* < *recv_count*, only the first "send_count" allocations of "recv_buf" will be updated.
- Therefore, the length of the *send_count* can be $\leq$ to the length of the *recv_count*;
  - ... but if < the program is wrong!!!
- To the end of a *receive*, it is possible to know the real length of the received message by analyzing the *status* argument.

How use `status`



- The **`status`** argument is:
  - a struct in C/C++;
  - an array of integer (of length MPI_STATUS_SIZE) in Fortran
- The **`status`** argument contains 3 fields:
  - MPI_TAG
  - MPI_SOURCE
  - MPI_ERROR
- To know the real length of the received message we have to use the function MPI_Get_count.

MPI_Get_count



C/C++

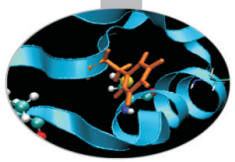
```
int MPI_Get_count(MPI_Status *status, MPI_Datatype dtype,
                  int *count)
```

Fortran

```
MPI_Get_count(status, dtype, count, ierr)
```

- Input:
  - status contains the informations about the received message (**MPI_Status**);
  - dtype is the type of every element of the received buffer (**MPI_Datatype**);
- Output:
  - count is the number of elements of the *receive* buffer (**integer**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int count;
MPI_Status status;
MPI_Get_count(&status, dtype, count);
```

Fortran

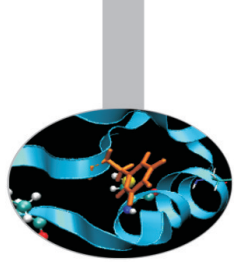
With mpif.h & mpi:

```
INTEGER ::  count, ierr
INTEGER ::  status(MPI_STATUS_SIZE)
CALL MPI_GET_COUNT(status, dtype, count, ierr)
```

With mpi_f08:

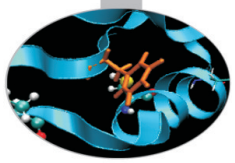
```
INTEGER ::  count
TYPE(MPI_Status) ::  status
CALL MPI_GET_COUNT(status, dtype, count)
```

Some notes

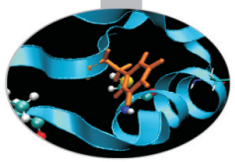


- The **`status`** is:
  - an *output* argument of those functions that **`take part`** to the communication (*e.g.* *MPI_Recv*);
  - an *input* argument of those functions that **`supervise`** the communication (*e.g.* *MPI_Get_count*).
- If you use **`MPI_Get_count`** for checking several *status* arguments, you have to identify uniquely the *status* for each communication operation.

Outline



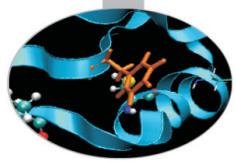
Send and receive an *integer*



C

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    MPI_Status status;
    int rank, size;
    int data_int;          /* INTEGER SENT AND RECEIVED */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {
        data_int = 10;
        MPI_Send(&data_int, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv( &data_int, 1, MPI_INT, 0, 123, MPI_COMM_WORLD,
                  &status);
        printf("Process 1 receives %d from process 0.\n",
               data_int);
    }
    MPI_Finalize();
    return 0;
 }
```

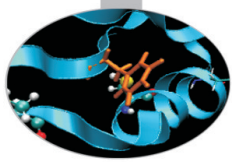
Send and receive a *part of array* (1/2)



C

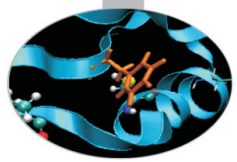
```c
#include <stdio.h>
#include <mpi.h>
#define VSIZE 50
#define BORDER 12

int main(int argc, char *argv[]) {
    MPI_Status status;
    int rank, size, i;
    int start_sbuf = BORDER;
    int start_rbuf = VSIZE - BORDER;
    int len = 10;
    int vec[VSIZE];

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

    (...continue)
```

Send and receive a *part of array* (2/2)



C

```
    for (i=0; i< VSIZE; i++) vec[i] = rank;
    if (rank == 0) {
        MPI_Send( &vec[start_sbuf], len, MPI_INT, 1, 123,
                  MPI_COMM_WORLD);
    }
    if (rank == 1) {
        MPI_Recv( &vec[start_rbuf], len, MPI_INT, 0, 123,
                  MPI_COMM_WORLD, &status);
        printf("Process 1 receives the following vector from
            process 0.\n");
        for (i=0; i<VSIZE; i++) {
                  printf("%6.2f ",vec[i]);
        }
    }
    MPI_Finalize();
    return 0;
}
```

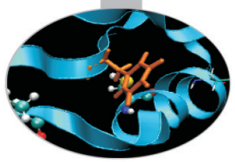


Send and receive a *matrix of double* (1/2)

Fortran with mpi_f08

```
PROGRAM Main

USE mpi_f08
IMPLICIT NONE
INTEGER ::  rank, size
INTEGER ::  i, j
TYPE(MPI_Datatype) ::  status
INTEGER, PARAMETER ::  MSIZE = 10
REAL*8 ::  matrix(MSIZE,MSIZE)

    CALL   MPI_Init()
    CALL   MPI_Comm_rank(MPI_COMM_WORLD, rank)
    CALL   MPI_Comm_size(MPI_COMM_WORLD, size);

    IF (rank .eq.  0) THEN
      DO i=1,MSIZE
        DO j=1,MSIZE
          matrix(i,j)=dble(i+j)
        ENDDO
      ENDDO
    (...continue)
```

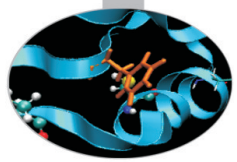
Send and receive an *array of double* (2/2)



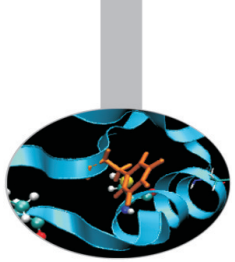
Fortran with mpi_f08

```
      CALL   MPI_SEND(matrix, MSIZE*MSIZE,   &
                      MPI_DOUBLE_PRECISION,   &
                      1, 123, MPI_COMM_WORLD)
    ELSE IF (rank .eq.  1) THEN
      CALL   MPI_RECV(matrix, MSIZE*MSIZE,   &
                      MPI_DOUBLE_PRECISION,   &
                      0, 123, MPI_COMM_WORLD,   &
                      status
      WRITE(*,*)'Proc 1 receives the following ' &
              'matrix from proc 0 '
      WRITE(*,'(10(f6.2,2x)) ') matrix
    ENDIF

    CALL   MPI_Finalize()
END PROGRAM Main
```

Outline

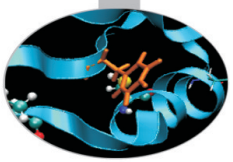


Types of pattern

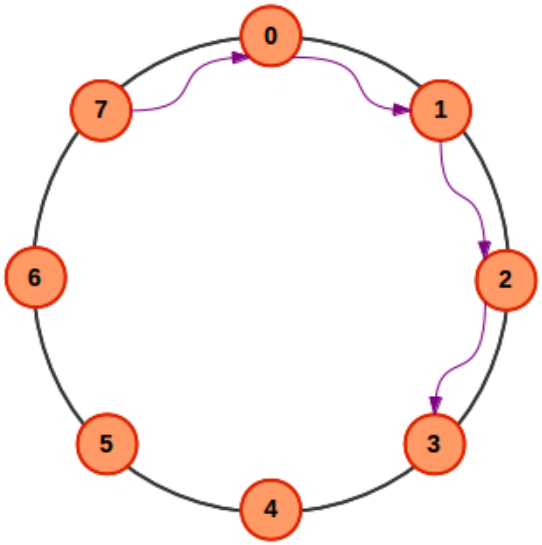


- In real parallel programs, several patterns of sending/receiving of a message are largely widespread.
- The communication patterns can be:
  - point-to-point, are the types that involve only two processes;
  - collectives, are the types that involve more processes.
- By the use of MPI functions, is possible to implement some communication patterns in a correct, easy and sturdy way:
  - the correctness doesn't have to be dependent by the number of processes.

Shift

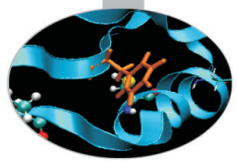


- Several parallel algorithms have the need of the communications between each process and one (or more) of its neighbours with rank greater or lower.
- This point-to-point pattern is known as *shift*.

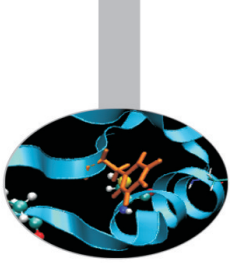


- Each process sends/receives a dataset along a direction (positive/negative) with a specific distance between ranks; for example:
  - the process i communicates with the process $i + 3$ if Δrank = 3;
  - the process i communicates with the process $i - 1$ if Δrank = 1 along a negative direction;
- If the *shift* is **`periodic`**:
  - the process with rank = size - Δrank sends the dataset to the process 0;
  - the process with rank = size - Δrank + 1 sends the dataset to the process 1.

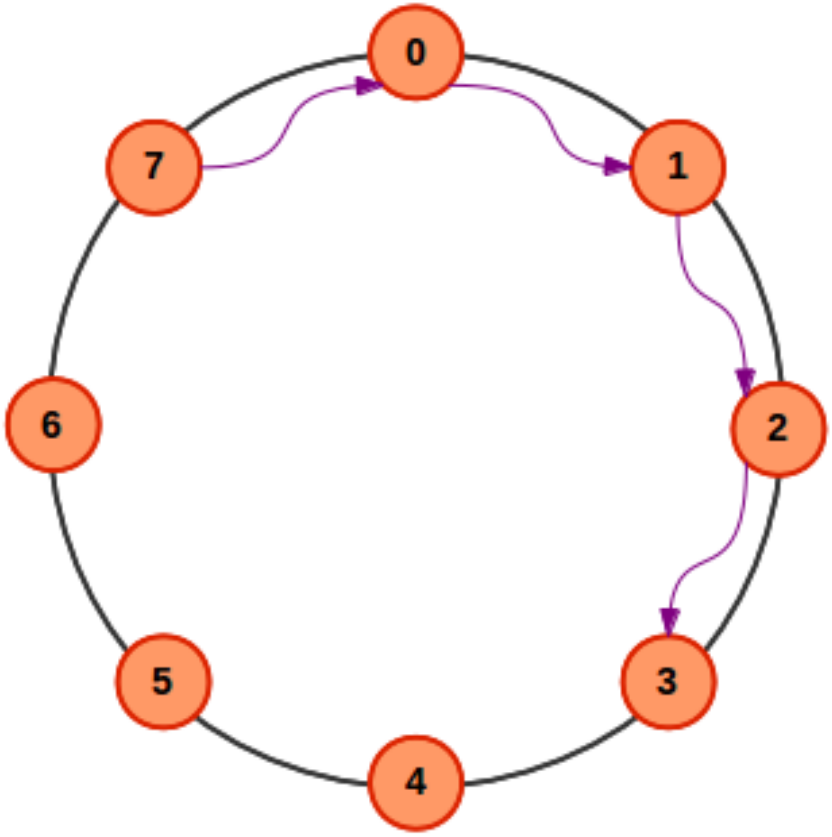
Outline



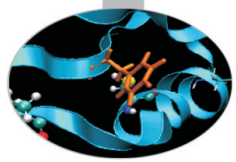
Circular periodic shift



- Each process produces an array *A* where all elements are the integers equal to the double of its own rank.
- Each process sends its own array *A* to the process with rank immediately consecutive.
  - Periodic Boundary: the last process sends the array to the first one.
- Each process receives an array *A* from the process with rank immediately preceding and stores it in an array *B*.
  - Periodic Boundary: the first process receives the array from the last one.



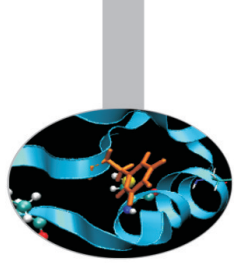
Circular periodic shift - Naive version



C/C++ (Portion of code)

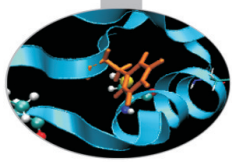
```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

tag = 201;
to = (rank + 1) %size;
from = (rank + size - 1) %size;

for(i = 1; i < MSIZE; i++)   A[i] = rank

MPI_Send(A, MSIZE, MPI_INT, to, tag, MPI_COMM_WORLD);
printf("Proc %d sends %d integers to proc %d\n", rank, MSIZE,
to);

MPI_Recv(B, MSIZE, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
printf("Proc %d receives %d integers from proc %d\n", rank,
MSIZE, from);

printf("Proc %d has A[0] = %d, B[0] = %d\n\n", rank, A[0],
B[0]);

MPI_Finalize();
```

Circular periodic shift - Test



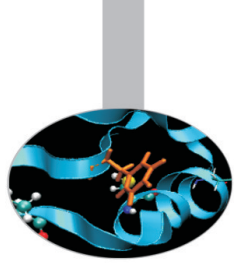
- Try to run the example with:
  - `MSIZE` = 100;
  - `MSIZE` = 500;
  - `MSIZE` = 1000;
  - `MSIZE` = 2000.
- What's happen?

Deadlock

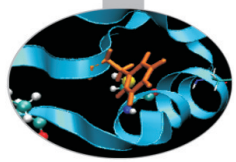


- The naive implementation for the circular periodic shift is wrong: for **`MSIZE`** > 1000 is produced a *deadlock*.
- The *deadlock* is a condition where each process is waiting for another one to end the communication and go on with the execution of the program.
- To understand the reason why the *deadlock* happens for **`MSIZE`** > 1000, we need to analyze better the inner working of the exchange of messages between processes.
- Please note that the value 1000 for MSIZE is limited to the laptop in use! The deadlock happens when the MSIZE value is greater than the maximum size of the memory buffer.

Outline

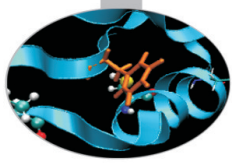


Buffered & Synchronous

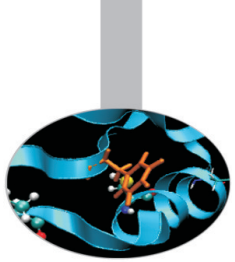


- The send standard functions of MPI don't return until the message sending operation is completed in according with one of the two ways:
  - Buffered: the sending of the message is executed through a copy of the sender buffer in a system buffer;
  - Synchronous: the sending of the message is executed through the direct copy of the sender buffer in the receiver buffer.
- The **`MPI_Send`** works with one of these two ways depending the size of the data that must be sent:
  - Buffered: for little sizes of data;
  - Synchronous: for big sizes of data.

Outline

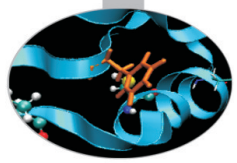


How MPI_Send works



- In the naive implementation of the circular periodic shift, MPI_Send works in buffered mode up to the size 1000 integers, and in synchronous mode for greater sizes:
  - for **`MSIZE`** $=$ 1000 the process can complete the "send" operation after that *A* is copied in the local buffer of the system where the process is running;
  - for **`MSIZE`** $=$ 2000 the process can complete the "send" operation only when exist a "receive" operation ready to take *A*.

Why the deadlock

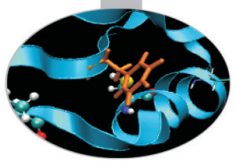


In the naive implementation the algorithm is of type:

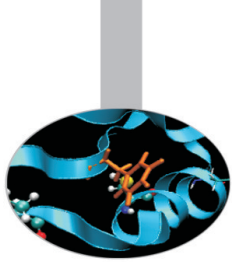
```
if (myrank = 0)
    SEND A to process 1
    RECEIVE B from process 1
else if (myrank = 1)
    SEND A to process 0
    RECEIVE B from process 0
endif
```

- For **MSIZE** = 2000 there are two "send" operations that to be completed are waiting for two "receive" operations...
- ...but each "receive" operation can be executed only after the corresponding "send" is completed...
- DEADLOCK!

Outline

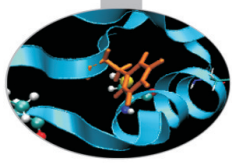


Solve the issue of circular shift



- To solve the just seen deadlock, we need a function that manages simultaneously the communications send and receive.
- The MPI function to do this is MPI_Sendrecv:
  - is useful when a process have to send and receive data at the same time;
  - can be invoked to implement communication pattern of shift type.

MPI_Sendrecv



C/C++

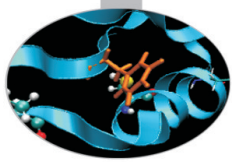
```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype sdtype,
                 int dest, int stag, void *rbuf, int rcount,
                 MPI_Datatype rdtype, int src, int rtag,
                 MPI_Comm comm, MPI_Status *status)
```

Fortran

```
MPI_SENDRECV(sbuf, scount, sdtype, dest, stag,
             rbuf, rcount, rdtype, src, rtag,
             comm, status, ierr)
```

- First arguments are the same of the MPI_Send;
- others are the same of the MPI_Recv.
- ierr [*only Fortran*] is the error handler (**integer**):
  - with **use mpi** or **include 'mpif.h'** is needed;
  - with **use mpi_f08** is optional.

Sintax



C/C++

```
int scount, rcount, dest, src, stag, rtag;
MPI_Status status;
MPI_Sendrecv(&sbuf, scount, sdtype, dest, stag,
             &rbuf, rcount, rdtype, src, rtag,
             MPI_COMM_WORLD, &status);
```

Fortran

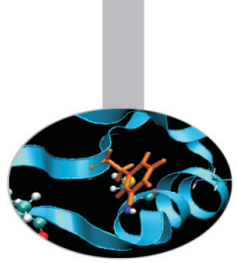
With mpif.h & mpi:

```
INTEGER ::  scount, rcount, dest, src, stag, rtag, ierr
INTEGER ::  status(MPI_STATUS_SIZE)

CALL MPI_SENDRECV(sbuf, scount, sdtype, dest, stag,
                  rbuf, rcount, rdtype, src, rtag,
                  MPI_COMM_WORLD, status, ierr)
```

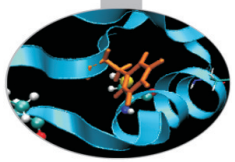
With mpi_f08:

```
INTEGER ::  scount, rcount, dest, src, stag, rtag
TYPE(MPI_Status) ::  status

CALL MPI_SENDRECV(sbuf, scount, sdtype, dest, stag,
                  rbuf, rcount, rdtype, src, rtag,
                  MPI_COMM_WORLD, status)
```

Outline



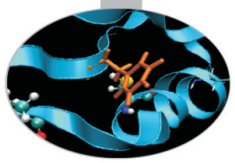
Circular periodic shift



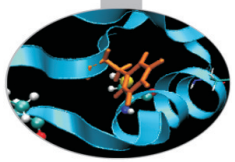
C

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

to = (rank + 1) %size;
from = (rank + size - 1) %size;

for(i = 1; i < MSIZE; i++)   A[i] = rank

MPI_Sendrecv(A, MSIZE, MPI_INT, to, 201,
             B, MSIZE, MPI_INT, from, 201,
             MPI_COMM_WORLD, &status);

printf("Proc %d sends %d integers to proc %d\n", rank, MSIZE,
to);
printf("Proc %d receives %d integers from proc %d\n", rank,
MSIZE, from);

MPI_Finalize();
```

Outline

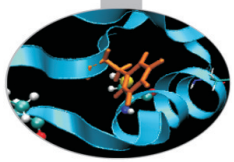


Overview

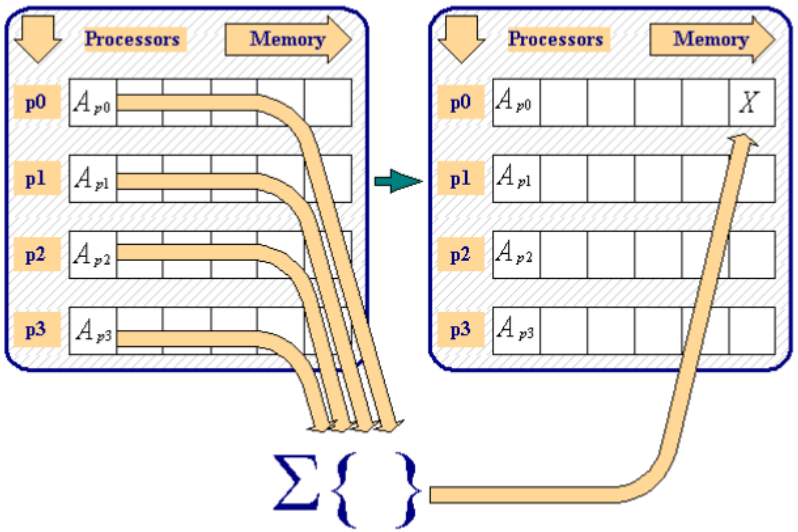


- Some communications patterns provide for the participation of all processes (of the communicator).
- MPI provide some functions for these patterns:
  - in this way the programmer doesn't need to implement these patterns from the point-to-point communications;
  - for these functions, the most efficient algorithms are implemented.
- We can classify these functions on the number of senders and receivers:
  - **`all-to-one`**: all processes send data to one process only;
  - **`one-to-all`**: one process sends data to all processes;
  - **`all-to-all`**: all processes send data to all processes.

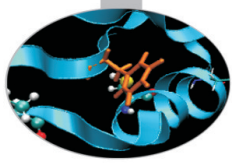
Reduction



- The *REDUCE* operation lets:
  - to collect from each process the data in the send buffer;
  - to *reduce* the data to an only value through an operator (in the figure: the sum operator);
  - to save the result in the receive buffer of the destination process, conventionally named root (in the figure: *p*0).
- The corresponding MPI function is MPI_Reduce:
  - it's in the **`all-to-one`** class.

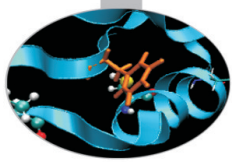


MPI_Op

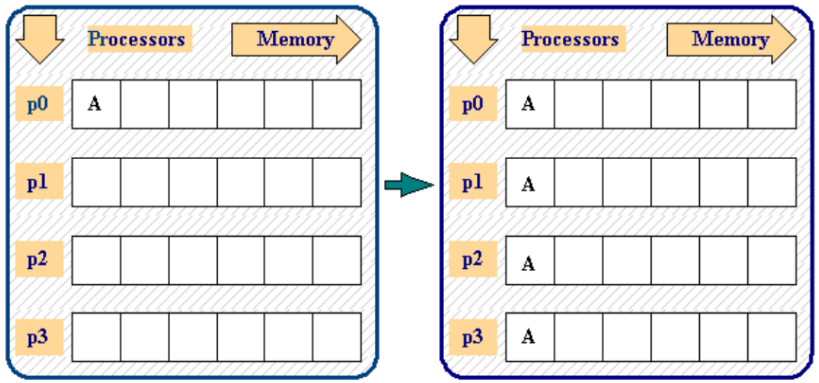


MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

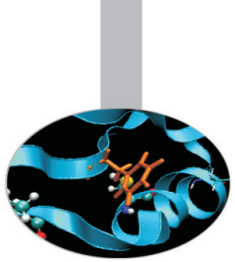
Broadcasting



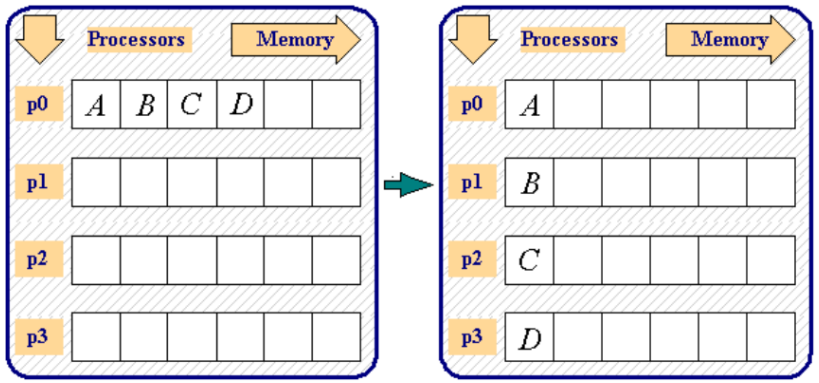
- The *BROADCAST* operation lets to copy data from the send buffer of root process (in the figure: *p*0) in the receive buffer of all processes living in the communicator (even process root).
- The corresponding MPI function is MPI_Bcast:
  - it's in the `one-to-all` class.



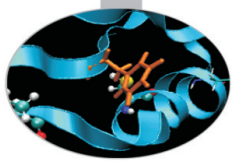
Scattering



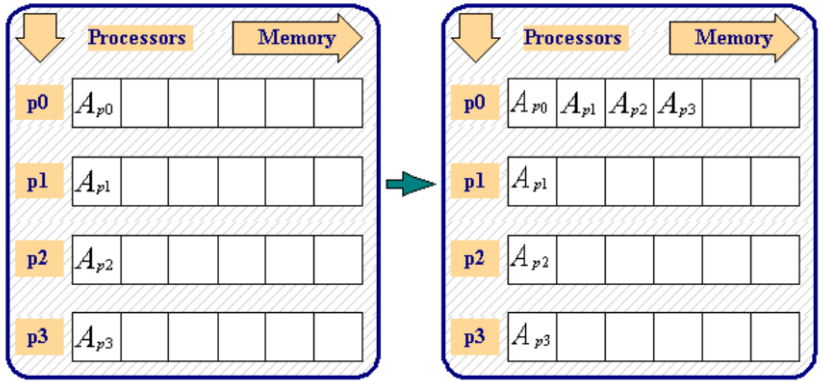
- The *SCATTER* operation lets to the root process (in the figure: *p*0):
  - to split in **`size`** equal portions a set of contiguous data in memory;
  - to send one portion to every process according to the order of **`rank`** (even process root).
- The corresponding MPI function is MPI_Scatter:
  - it's in the **`one-to-all`** class.



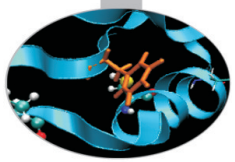
Gathering



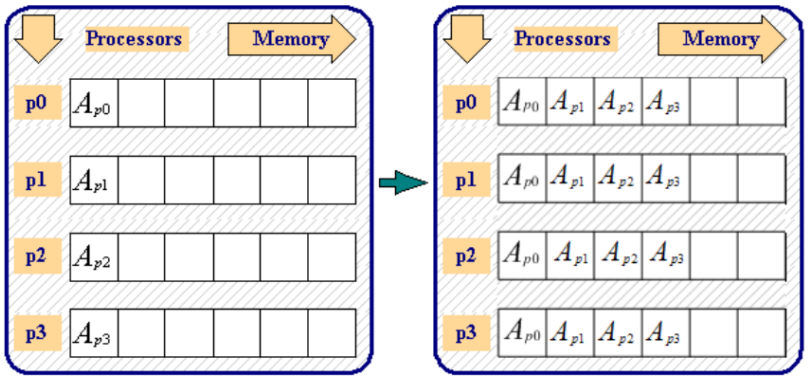
- The *GATHER* operation is the reverse of the *SCATTER* operation:
  - each process (even process root) sends data contained in the send buffer to root process;
  - the root process receives data and reorders them according the order of **`rank`**.
- The corresponding MPI function is MPI_Gather:
  - it's in the **`all-to-one`** class.



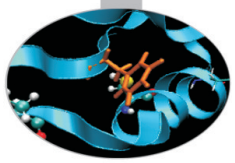
Gathering & Broadcasting



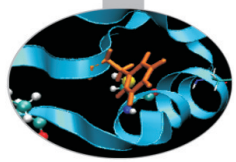
- The *GATHER + BROADCAST* operation is equal to a *GATHER* operation followed by a *BROADCAST* operation executed by root process.
- The corresponding MPI function is MPI_Allgather:
  - it's much more advantageous and efficient of the sequence *GATHER + BROADCAST*;
  - it's in the **`all-to-all`** class.



Outline



MPI_Reduce



C/C++

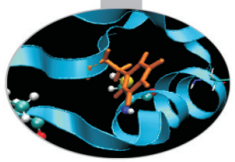
```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
               MPI_Datatype dtype, MPI_Op op, int root,
               MPI_Comm comm)
```

Fortran

```
MPI_REDUCE(sbuf, rbuf, count, dtype, op, root, comm, ierr)
```

- *Input* arguments:
  - sbuf is the initial address of the *send* buffer;
  - count is the number of elements of the *send*/*receive* buffer (**integer**);
  - dtype is the type of every element of the *send*/*receive* buffer (**MPI_Datatype**);
  - op is the reference to the operator for the reduction (**MPI_Op**);
  - root is the *rank* of the process root for the reduction (**integer**);
  - comm is the communicator of the processes that contribute to the reduction (**MPI_Comm**);
- *Output* arguments:
  - rbuf is the initial address of the *receive* buffer;
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional;

Sintax



C/C++

```
int count, root;
MPI_Reduce(&sbuf, &rbuf, count, dtype, op, root,
           MPI_COMM_WORLD);
```

Fortran

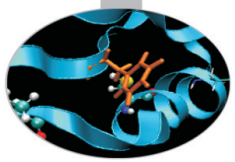
With mpif.h & mpi:

```
INTEGER ::  count, root, ierr

CALL MPI_REDUCE(sbuf, rbuf, count, dtype, op, root,
                MPI_COMM_WORLD, ierr)
```

With mpi_f08:

```
INTEGER ::  count, root

CALL MPI_REDUCE(sbuf, rbuf, count, dtype, op, root,
                MPI_COMM_WORLD)
```

MPI_Bcast



C/C++

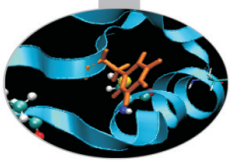
```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
              int root, MPI_Comm comm)
```

Fortran

```
MPI_BCAST(buf, count, dtype, root, comm, ierr)
```

- *Input* arguments:
  - count is the number of elements of the *send*/*receive* buffer (**integer**);
  - dtype is the type of every element of the *send*/*receive* buffer (**MPI_Datatype**);
  - root is the *rank* of the process root for the broadcasting (**integer**);
  - comm is the communicator for the broadcasting (**MPI_Comm**);
- *Input*/*Output* arguments:
  - buf is the initial address of the *send* and *receive* buffer;
- *Output* arguments:
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int count, root;
MPI_Bcast(&buf, count, dtype, root, MPI_COMM_WORLD);
```

Fortran

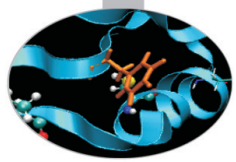
With mpif.h & mpi:

```
INTEGER ::  count, root, ierr
CALL MPI_BCAST(buf, count, dtype, root, MPI_COMM_WORLD, ierr)
```

With mpi_f08:

```
INTEGER ::  count, root
CALL MPI_BCAST(buf, count, dtype, root, MPI_COMM_WORLD)
```

MPI_Scatter



C/C++

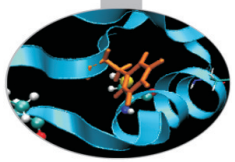
```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype sdtype,
                void *rbuf, int rcount, MPI_Datatype rdtype,
                int root, MPI_Comm comm)
```

Fortran

```
MPI_SCATTER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
            root, comm, ierr)
```

- *Input* arguments:
  - sbuf is the initial address of the *send* buffer.
  - scount is the number of elements of the *send* buffer (**integer**);
  - rcount is the number of elements of the *receive* buffer (**integer**);
  - sdtype is the type of every element of the *send* buffer (**MPI_Datatype**);
  - rdtype is the type of every element of the *receive* buffer (**MPI_Datatype**);
  - root is the *rank* of the process root for the scattering (**integer**);
  - comm is the communicator for the scattering (**MPI_Comm**);
- *Output* arguments:
  - rbuf is the initial address of the *receive* buffer;
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int scount, rcount, root;
MPI_Scatter(&sbuf, scount, sdtype, &rbuf, rcount, rdtype,
            root, MPI_COMM_WORLD);
```

Fortran

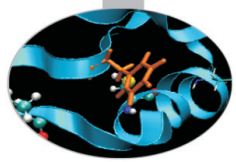
With mpif.h & mpi:

```
INTEGER ::  scount, rcount, root, ierr

CALL MPI_SCATTER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                 root, MPI_COMM_WORLD, ierr)
```

With mpi_f08:

```
INTEGER ::  scount, rcount, root

CALL MPI_SCATTER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                 root, MPI_COMM_WORLD)
```

MPI_Gather



C/C++

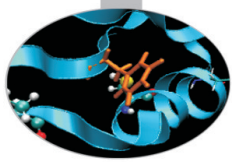
```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype sdtype,
               void *rbuf, int rcount, MPI_Datatype rdtype,
               int root, MPI_Comm comm)
```

Fortran

```
MPI_GATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
           root, comm, ierr)
```

- *Input* arguments:
  - sbuf is the initial address of the *send* buffer.
  - scount is the number of elements of the *send* buffer (**integer**);
  - rcount is the number of elements of the *receive* buffer (**integer**);
  - sdtype is the type of every element of the *send* buffer (**MPI_Datatype**);
  - rdtype is the type of every element of the *receive* buffer (**MPI_Datatype**);
  - root is the *rank* of the process root for the gathering (**integer**);
  - comm is the communicator for the gathering (**MPI_Comm**);
- *Output* arguments:
  - rbuf is the initial address of the *receive* buffer;
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int scount, rcount, root;
MPI_Gather(&sbuf, scount, sdtype, &rbuf, rcount, rdtype,
               root, MPI_COMM_WORLD);
```

Fortran

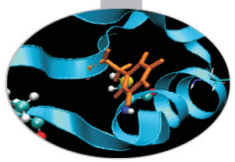
With mpif.h & mpi:

```
INTEGER ::  scount, rcount, root, ierr
CALL MPI_GATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                   root, MPI_COMM_WORLD, ierr)
```

With mpi_f08:

```
INTEGER ::  scount, rcount, root
CALL MPI_GATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                   root, MPI_COMM_WORLD)
```

MPI_Allgather



C/C++

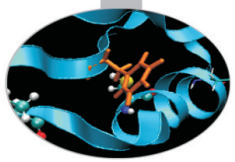
```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype,
                  void *rbuf, int rcount, MPI_Datatype rdtype,
                  MPI_Comm comm)
```

Fortran

```
MPI_ALLGATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
              comm, ierr)
```

- *Input* arguments:
  - sbuf is the initial address of the *send* buffer.
  - scount is the number of elements of the *send* buffer (**integer**);
  - rcount is the number of elements of the *receive* buffer (**integer**);
  - sdtype is the type of every element of the *send* buffer (**MPI_Datatype**);
  - rdtype is the type of every element of the *receive* buffer (**MPI_Datatype**);
  - comm is the communicator for the communication (**MPI_Comm**);
- *Output* arguments:
  - rbuf is the initial address of the *receive* buffer;
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int scount, rcount;
MPI_Allgather(&sbuf, scount, sdtype, &rbuf, rcount, rdtype,
              MPI_COMM_WORLD);
```

Fortran

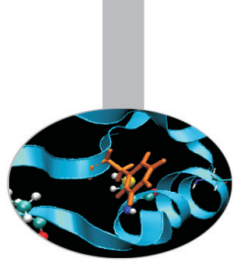
With mpif.h & mpi:

```
INTEGER ::  scount, rcount, ierr

CALL MPI_ALLGATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                   MPI_COMM_WORLD, ierr)
```

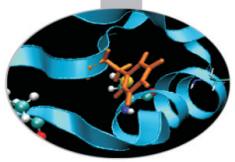
With mpi_f08:

```
INTEGER ::  scount, rcount

CALL MPI_ALLGATHER(sbuf, scount, sdtype, rbuf, rcount, rdtype,
                   MPI_COMM_WORLD)
```

Outline

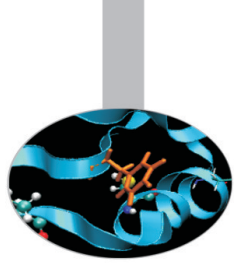


Hints

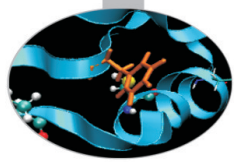


- **`MPI_Scatterv`**: as MPI_Scatter, but each message has different count and displacement.
- **`MPI_Gatherv`**: as MPI_Gather, but each message has different count and displacement.
- **`MPI_Barrier`**: synchronizes all processes.
- **`MPI_Alltoall`**: each process sends a message to all processes.
- **`MPI_Allreduce`**: as MPI_Reduce, but all processes receive the result.
- ...and others

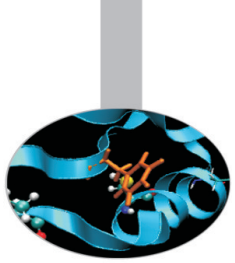
Outline



Outline



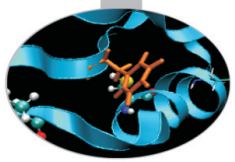
Overview (1/2)



A communication point-to-point can be:

- *BLOCKING*:
  - the control returns to the process which has invoked the primitive for the communication only when that has been completed;
  - every process must wait the completion of the operations of all others processes before to continue with the computation.
- *NON BLOCKING*:
  - the control returns to the process which has invoked the primitive for the communication only when that has been executed;
  - the control of the effective completion of the communication must be done later;
  - in the meanwhile, the process can execute other operations, without waiting the completion of the operations of others processes.

All collective communications are *BLOCKING*.

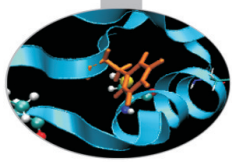


Overview (2/2)

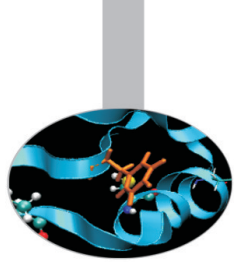
- In the MPI library there are several primitives for the point-to-point communications that combine communication modes with completion criterions.
- The criterion of completion of the communication is relevant.

Functionality	Completion criterions
Synchronous send	It's completed when the receiving of message is ended
Buffered send	It's completed when the writing of data on the buffer is ended (indipendent by receiver!)
Standard send	It can be implemented *synchronous* or *buffered send*
Receive	It's completed when the message arrives

Outline



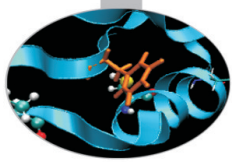
Synchronous Send



The MPI function is **`MPI_Ssend`**:

- the function arguments are the same of *MPI_Send*;
- to have the completion of the operation, the *sender* must be informed by the *receiver* that the message has been received;
- advantage:
  - is the point-to-point communication mode much more easy and dependable;
- disadvantage:
  - can cause a significant loss time for the processes that are waiting for the end of the communication.

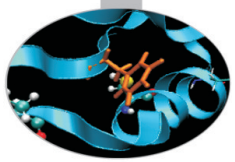
Buffered Send



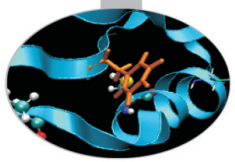
The MPI function is **`MPI_Bsend`**:

- the function arguments are the same of *MPI_Send*;
- the completion is immediate, as soon as the process has done the copy of the message in an appropriate transmitting buffer;
- the programmer can't suppose the existence of an allocated system buffer to execute the operation, and has to execute an operation of:
  - **`BUFFER_ATTACH`** to define a memory area, with the needed dimension, as buffer for the trasmission of the messages;
  - **`BUFFER_DETACH`** to free the memory areas of the buffers used.
- advantage:
  - immediate return by the primitive of communication;
- disadvantage:
  - explicit management of the buffer is needed;
  - entails an operation copy in memory of data that have to be transmitted.

Outline



MPI_Buffer_attach



C/C++

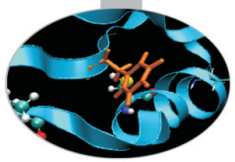
```
int MPI_Buffer_attach(void *buf, int bsize)
```

Fortran

```
MPI_BUFFER_ATTACH(buf, bsize, ierr)
```

- Allows the *sender* process to allocate the *send* buffer for a succesive call to MPI_Bsend.
- *Input* arguments:
  - buf is the initial address of the buffer that has to be allocated;
  - bsize is the dimension in byte of buf (**integer**);
- *Output* arguments:
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

MPI_Buffer_detach



C/C++

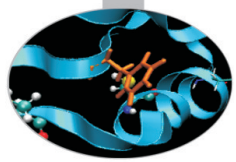
```
int MPI_Buffer_detach(void *buf, int bsize)
```

Fortran

```
MPI_BUFFER_DETACH(buf, bsize, ierr)
```

- Allows to free the buffer allocated by MPI_Buffer_attach.
- All arguments are *output*:
  - buf is the initial address of the buffer that has to be deallocated;
  - bsize is the dimension in byte of buf (**integer**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

MPI_BSEND_OVERHEAD



- The value of *bsize* is given by the sum of *count*∗*dtype_size*) and MPI_BSEND_OVERHEAD.
- MPI_BSEND_OVERHEAD is a macro and gives the maximum amount of space that may be used in the buffer by *MPI_Bsend*.
- The value of MPI_BSEND_OVERHEAD is declared in MPI headers.
- *dtype_size* is returned by the MPI function MPI_Type_size.

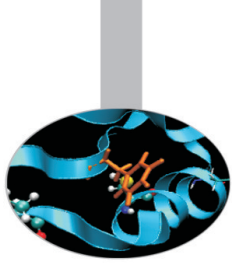
C/C++

```
int MPI_Type_size(MPI_Datatype dtype, int *dtype_size)
bsize = count*dtype_size + MPI_BSEND_OVERHEAD;
```

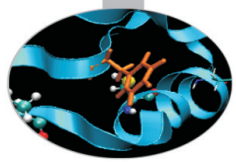
Fortran

```
MPI_TYPE_SIZE(dtype, dtype_size, ierr)
bsize = count*dtype_size + MPI_BSEND_OVERHEAD
```

Outline



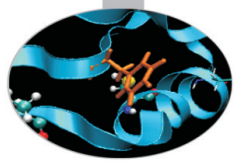
MPI_Ssend



C

```c
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

int main(int argc, char *argv[]) {
    MPI_Status status;
    int rank, size, i;
    double matrix[MSIZE];

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

    if (rank == 0) {
        for (i=0; i< MSIZE; i++) matrix[i] = (double) i;
        MPI_Ssend(matrix, MSIZE, MPI_DOUBLE 1, 123,
                  MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(matrix, MSIZE, MPI_DOUBLE, 0, 123,
                  MPI_COMM_WORLD, &status);
        printf("Process 1 receives an array of size %d
                from process 0.\n", MSIZE);
    }
    MPI_Finalize();
    return 0;
 }
```

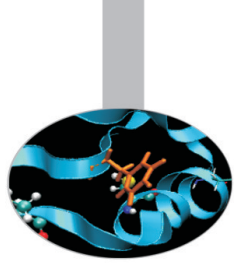
MPI_Bsend



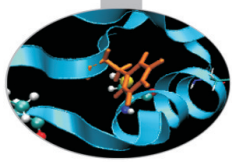
Fortran with mpi_f08

```
PROGRAM Main

USE mpi_f08
IMPLICIT NONE
INTEGER ::  rank, size, bsize, dtype_size, i
TYPE(MPI_Datatype) ::  status
INTEGER, PARAMETER ::  MSIZE = 10
REAL*8 ::  matrix(MSIZE)
REAL*8, DIMENSION(:), ALLOCATABLE ::  buf

    CALL   MPI_Init()
    CALL   MPI_Comm_rank(MPI_COMM_WORLD, rank)
    CALL   MPI_Comm_size(MPI_COMM_WORLD, size);

    IF (rank .eq.  0) THEN
      DO i=1,MSIZE
        matrix(i)=dble(i)
      ENDDO
      CALL MPI_Type_size(MPI_DOUBLE_PRECISION, dtype_size)
      bsize = dtype_size*msize + MPI_BSEND_OVERHEAD
      ALLOCATE(buf(bsize))
      CALL MPI_Buffer_attach(buf, bsize)
      CALL MPI_Bsend(matrix, MSIZE, MPI_DOUBLE_PRECISION, 1, 123, MPI_COMM_WORLD)
      CALL MPI_Buffer_detach(buf, bsize)
    ELSE IF (rank .eq.  1) THEN
      CALL MPI_Recv(matrix, MSIZE, MPI_DOUBLE_PRECISION, 0, 123, MPI_COMM_WORLD, &
                    status)
    ENDIF
    CALL MPI_Finalize()
END PROGRAM Main
```

Outline

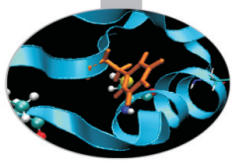


Overview

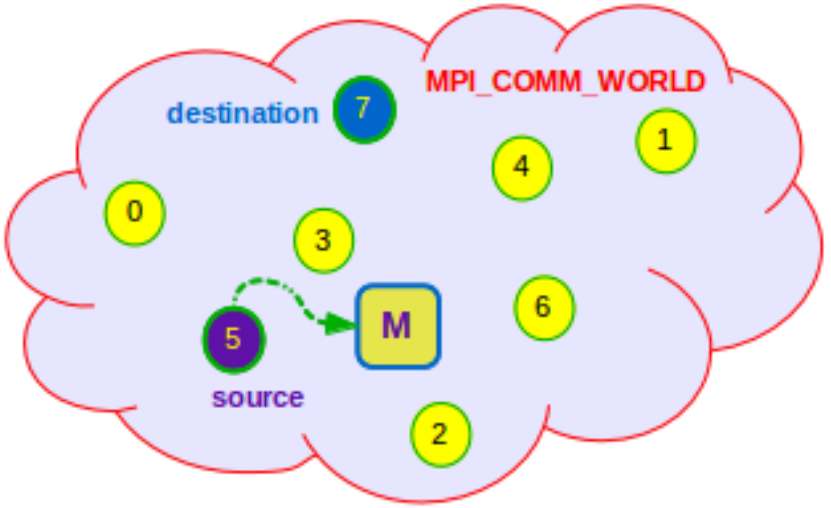


- A non-blocking communication usually consists in three successive phases:
  - the beginning of the operation *send* or *receive*;
  - a set of activities that doesn't need to access to the data interested by the communication;
  - the control or the waiting for the ending of the communication.
- advantage:
  - communication phases and computation can coexist;
  - the latency effects of communications are reduced;
  - deadlocks are prevented;
- disadvantage:
  - the programming with functions for the non blocking communication is a little bit more difficult.

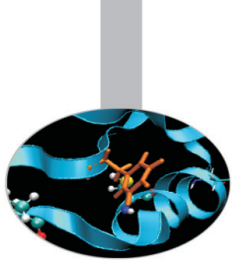
Non-blocking send



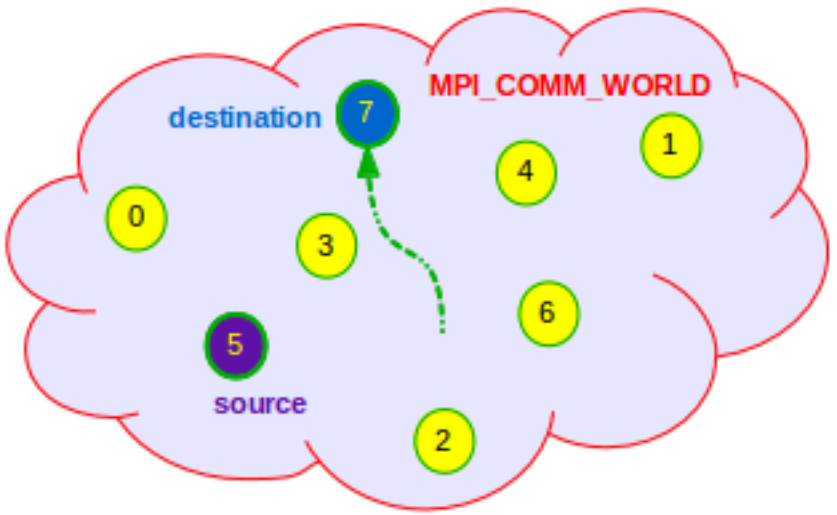
- The control returns to the sender process after the beginning of the sending.
- The sender process must check if the operation has been completed (by the use of some MPI functions) before to reuse the memory areas used by the communication.
- For the non-blocking send are available the same several completion modes shown before.



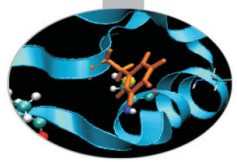
Non-blocking receive



- The control returns to the receiver process after the beginning of the receiving phase.
- The receiver process must check if the operation has been completed (by the use of some MPI functions) before to use in a secure way the received data.
- A non-blocking receive can be used to receive messages sent by a send blocking or non-blocking.



MPI_Isend



C/C++

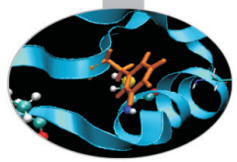
```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype,
              int dest, int tag, MPI_Comm comm,
              MPI_request *req)
```

Fortran

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

- *Input* arguments:
  - buf is the initial address of the *send* buffer;
  - count is the number of elements of the *send* buffer (**integer**);
  - dtype is the type of every element of the *send* buffer (**MPI_Datatype**);
  - dest is the *rank* of the receiver in the communicator *comm* (**integer**);
  - tag is the identity number of the message (**integer**);
  - comm is the communicator where is the *send* (**MPI_Comm**);
- *Output* arguments:
  - req is the non-blocking handler (**MPI_Request**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

MPI_Irecv



C/C++

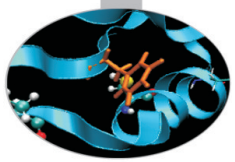
```
int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int src,
              int tag, MPI_Comm comm, MPI_Request *req)
```

Fortran

```
MPI_IRECV(buf, count, dtype, src, tag, comm, req, ierr)
```

- *Input* arguments:
  - count is the number of elements of the *receive* buffer (**integer**);
  - dtype is the type of every element of the *receive* buffer (**MPI_Datatype**);
  - src is the *rank* of the sender in the communicator *comm* (**integer**);
  - tag is the identity number of the message (**integer**);
  - comm is the communicator where is the *send* (**MPI_Comm**);
- *Output* arguments:
  - buf is the initial address of the *receive* buffer;
  - req is the non-blocking handler (**MPI_Request**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Sintax



C/C++

```
int count, dest, src, tag;
MPI_Request req;
MPI_Isend(&buf, count, dtype, dest, tag, MPI_COMM_WORLD, &req);
MPI_Irecv(&buf, count, dtype, src, tag, MPI_COMM_WORLD, &req);
```

Fortran

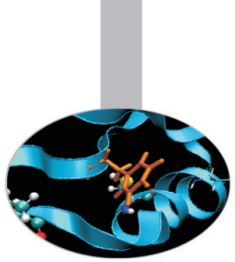
With mpif.h & mpi:

```
INTEGER ::  count, dest, src, req, ierr (,dtype, MPI_COMM_WORLD)

CALL MPI_ISEND(buf, count, dtype, dest, tag, MPI_COMM_WORLD,
               req, ierr)
CALL MPI_IRECV(buf, count, dtype, src, tag, MPI_COMM_WORLD,
               req, ierr)
```

With mpi_f08:

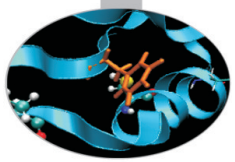
```
INTEGER ::  count, dest, src, tag   TYPE(MPI_Datatype) ::  dtype
TYPE(MPI_Comm) ::  MPI_COMM_WORLD   TYPE(MPI_Status) ::  status
TYPE(MPI_Request) ::  req

CALL MPI_ISEND(buf, count, dtype, dest, tag, MPI_COMM_WORLD,
               req)
CALL MPI_IRECV(buf, count, dtype, src, tag, MPI_COMM_WORLD,
               req)
```

Check the completion



- The request handle *req* is needed to check the completion of the non-blocking communication.
- MPI makes available two functions to do this check:
  - *MPI_Wait*: it allows to stop the process execution until the communication is complete;
  - *MPI_Test*: a logical *TRUE* or *FALSE* is returned to the process to let check the completion with a conditional statement.
- The handle *status* is returned by these two functions.

MPI_Wait & MPI_Test



C/C++

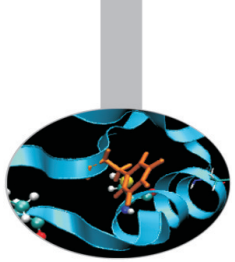
```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

Fortran

```
MPI_WAIT(req, status, ierr)
MPI_TEST(req, flag, status, ierr)
```

- *Input/Output* arguments:
  - req is the non-blocking handler (**MPI_Request**);
- *Output* arguments:
  - flag is the logical *TRUE/FALSE* (**logical**);
  - status contains the informations about the message (**MPI_Status**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

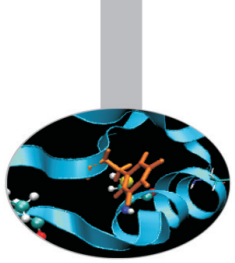
An example



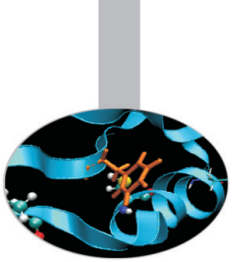
C/C++ (Only a portion of the code...)

```
...    ...    ...
MPI_Status status;
MPI_Request req;
int flag = 0;
double buffer[BIG_SIZE];
MPI_Isend(buffer, BIG_SIZE, MPI_DOUBLE, dest, tag,
          MPI_COMM_WORLD, &req);
while (!flag && have more work to do) {
    ...  do some work ...
   MPI_Test(&req, &flag, &status);
}
if (!flag)   MPI_Wait(&req, &status);
...    ...    ...
```

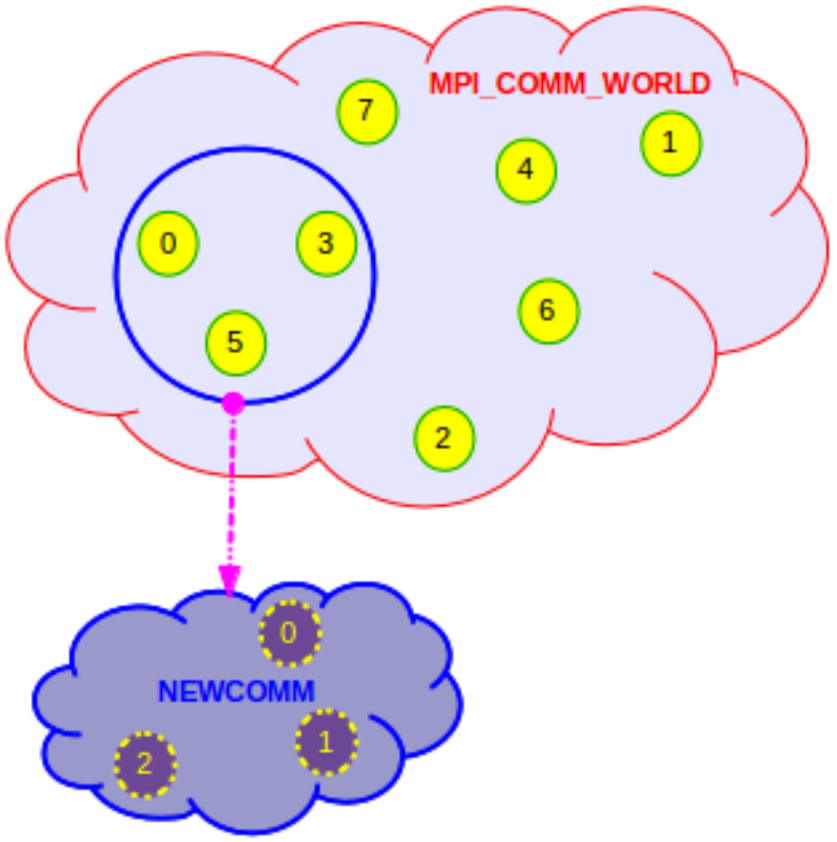
Outline



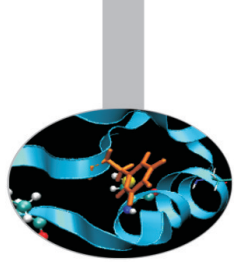
Beyond MPI_COMM_WORLD



- A communicator defines the *universe of communication* of a set of processes.
- In an MPI program is possible to define others communicators beyond MPI_COMM_WORLD to meet particular needs, as:
  - use collective functions only for a subset of processes;
  - use an identificative scheme of processes that is useful for a particular pattern communication.

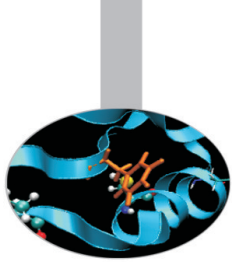


Components of communicator



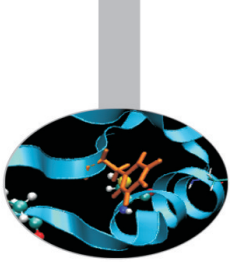
- *Group* of processes (an ordered set of processes):
  - is used to identify all processes;
  - to every process is assigned an index (*rank*), useful to identify the process;
- *Context*:
  - used by the communicator to manage the send/receive of messages;
  - contains several information regarding the message status;
- *Features* (others informations possibly linked to the communicator):
  - the rank of the process eligible to execute I/O operations;
  - the topology of communication.

Virtual topologies

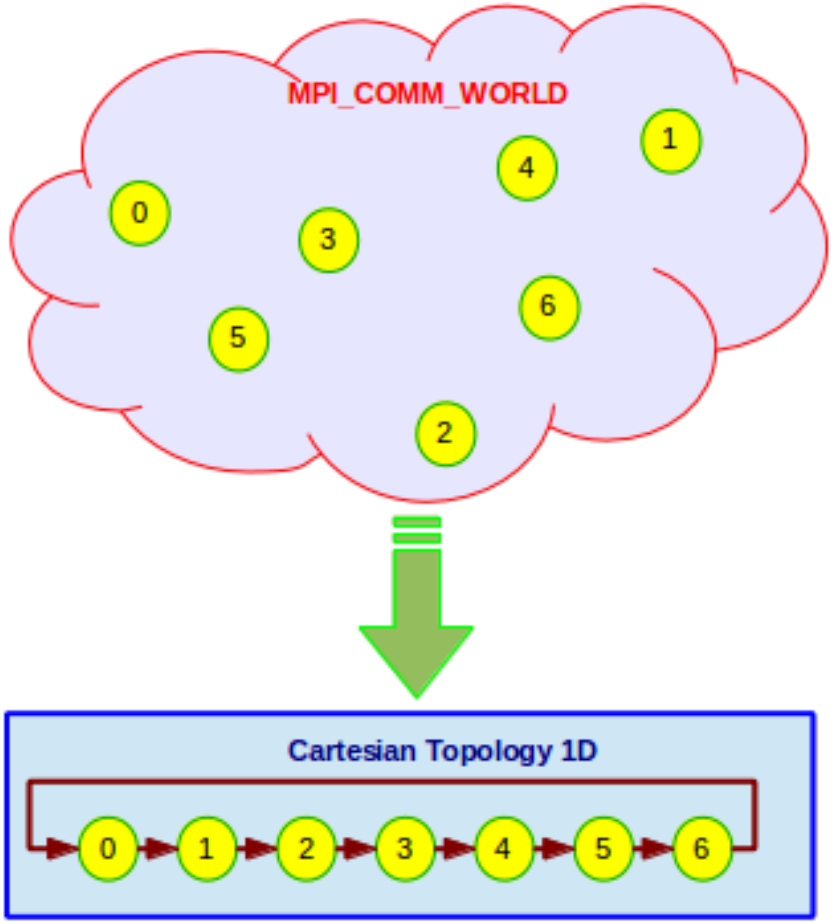


- Define a new identificative scheme for the processes is useful because:
  - simplifies the code writing;
  - allows MPI to optimize the communications.
- Create a virtual topology of processes in MPI means:
  - to define a new communicator with its own features.
- Topologies:
  - *Cartesian*:
    - every process is identified by a set of cartesian coordinates and connected to its neighbors by a virtual grid;
    - it's possible to set periodicity or not at the boundaries of the grid.
  - *Graph* (`not talked about in this course`).

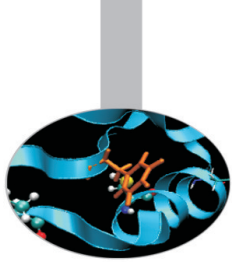
Topology cartesian 1D



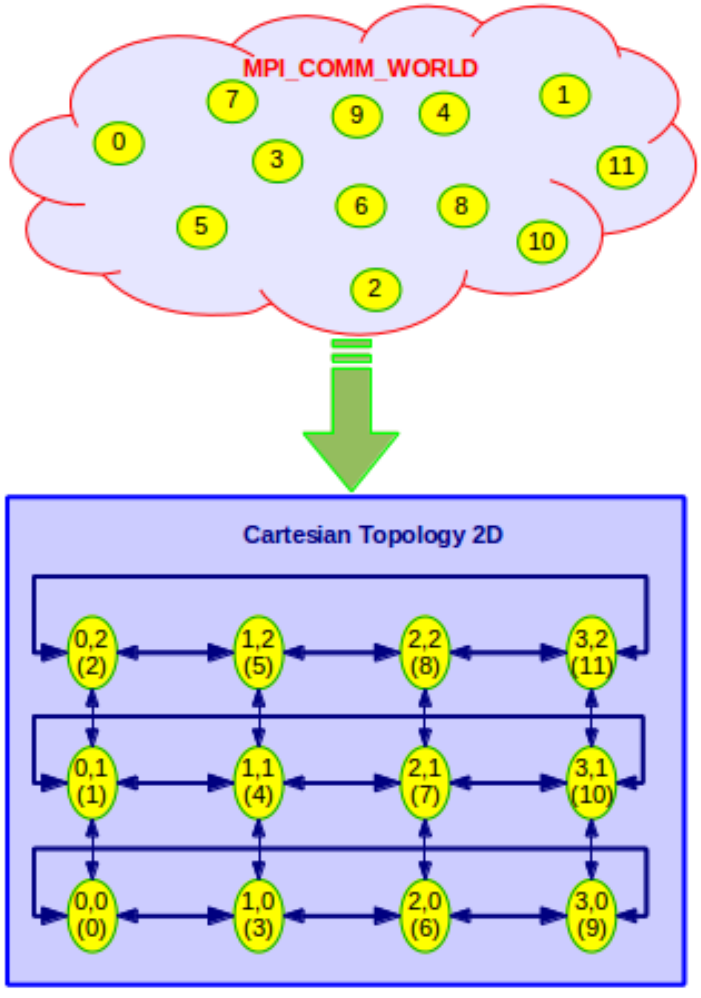
- An example:
  - every process sends data to right and receives data from left (*Circular shift*);
  - the last process sends data to the first one (*periodicity*).



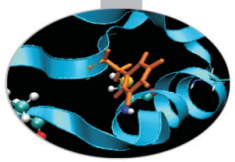
Topology cartesian 2D



- An example:
  - to every process is assigned a pair of index as its cartesian coordinates in a cartesian 2D (virtual) space;
  - the communications can happen between first neighbours:
    - with periodicity along X direction;
    - without periodicity along Y direction.



MPI_Cart_create



C/C++

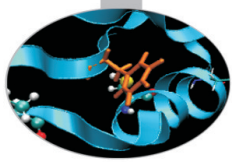
```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                    int *periods, int reorder, MPI_Comm *comm_cart)
```

Fortran

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
                comm_cart, ierr)
```

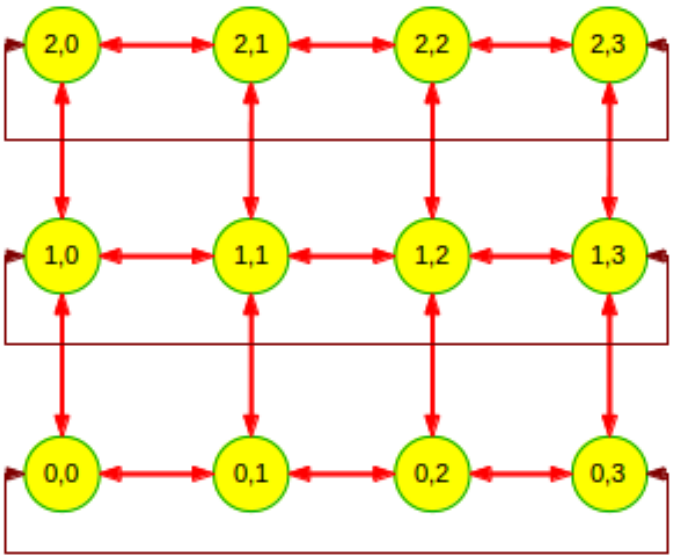
- *Input* arguments:
  - comm_old is the old communicator (**MPI_Comm**);
  - ndims is the dimension of cartesian space (**integer**);
  - dims is the vector whose elements are the number of processes along the space directions (**integer**);
  - periods is the vector whose elements are the logical *true*/*false* to define the periodicity along the space directions (**logical**);
  - reorder is the logical *true*/*false* to reorder the rank of the processes(**logical**);
- *Output* arguments:
  - comm_cart is the new communicator(**MPI_Comm**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

A C example

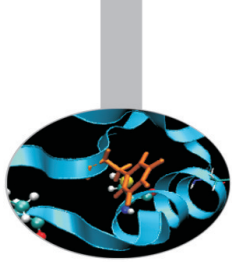


C/C++

```
int main(int argc, char **argv)
 {
...     ...     ...
    int dim[2], period[2], reorder;
...     ...     ...
    dim[0] = 4;
    dim[1] = 3;

    period[0] = 1;
    period[1] = 0;
    reorder = 1;

    MPI_Cart_create(MPI_COMM_WORLD, 2, dim,
                    period, reorder, &cart);

...     ...     ...
    return 0;
 }
```

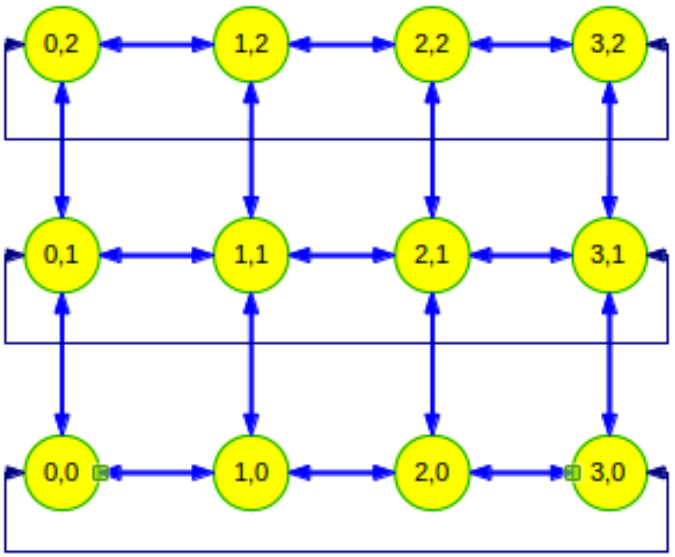


A Fortran example

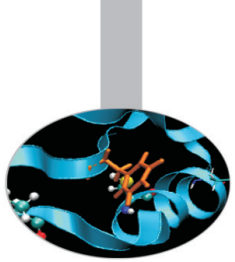


Fortran

```
PROGRAM Main
...    ...    ...
integer ::  dim[2], ierr
logical ::  period[2], reorder
...    ...    ...
dim[0] = 3
dim[1] = 4

period[0] = .false.
period[1] = .true.
reorder = .true.

call MPI_Cart_create(MPI_COMM_WORLD, 2, &
          dim, period, reorder, cart, &
          ierr);

...    ...    ...
return
END PROGRAM Main
```

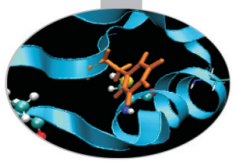


Some useful features



- MPI_Dims_create:
  - calculates the dimensions of the balanced optimal grid in relation of the number of processes and the cartesian dimension of the grid;
  - is useful to set the vector *dims* for the function MPI_Cart_create.
- MPI_Cart_coords:
  - returns the coordinates corresponding to the process defined by an established *rank*, in respect to the defined topology in the communicator.
- MPI_Cart_rank:
  - returns the rank corresponding to the process linked to an established set of cartesian *coordinates*, in respect to the defined topology in the communicator.

MPI_Dims_create



C/C++

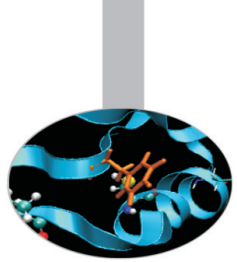
```
int MPI_Dims_create(int size, int ndims, int *dims)
```

Fortran

```
MPI_DIMS_CREATE(size, ndims, dims, ierr)
```

- *Input* arguments:
  - size is the total number of processes (**integer**);
  - ndims is the dimension of cartesian space (**integer**);
- *Input*/*Output* arguments:
  - dims is the vector whose elements are the number of processes along the space directions (**integer**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

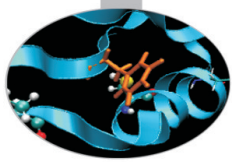
Notes to MPI_Dims_create



- The entries in the array dims are set to describe a cartesian grid with *ndims* dimensions and a total of *size* nodes.
- The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm.
- The caller may further constrain the operation of this routine by specifying elements of array *dims*.
- If *dims*[*i*] is set to a positive number, the routine will not modify the number of nodes in dimension *i*; only those entries where *dims*[*i*] $= 0$ are modified by the call.
- Negative input values of *dims*[*i*] are erroneous.
- An error will occur if nnodes is not a multiple of:

$$\prod dims[i].$$

MPI_Cart_coords



C/C++

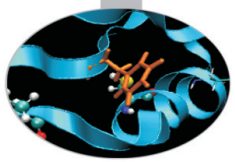
```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
                    int *coords)
```

Fortran

```
MPI_CART_COORDS(coom, rank, ndims, coords, ierr)
```

- *Input* arguments:
  - comm is the communicator with cartesian topology (**`MPI_Comm`**);
  - rank is the identificative number of the process about which we want to know the cartesian coordinates (**`integer`**);
  - ndims is the dimension of cartesian space (**`integer`**);
- *Output* arguments:
  - coords is the vector of coordinates assigned to the process *rank* (**`integer`**);
  - ierr [*only Fortran*] is the error handler (**`integer`**):
    - with **`use mpi`** or **`include 'mpif.h'`** is needed;
    - with **`use mpi_f08`** is optional.

MPI_Cart_rank



C/C++

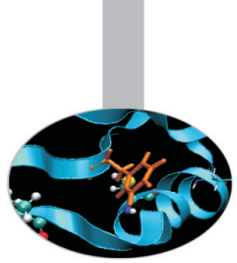
```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Fortran

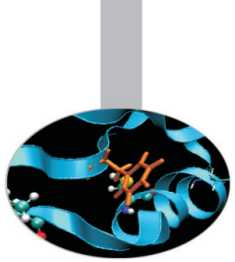
```
MPI_CART_RANK(coom, coords, rank, ierr)
```

- *Input* arguments:
  - comm is the communicator with cartesian topology (**MPI_Comm**);
  - coords is the vector of coordinates assigned to the process (**integer**);
- *Output* arguments:
  - rank is the identificative number of the process with cartsian coordinates *coords* (**integer**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

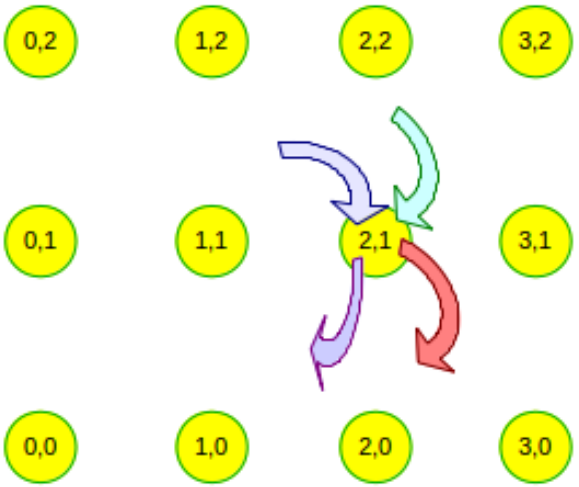
Outline



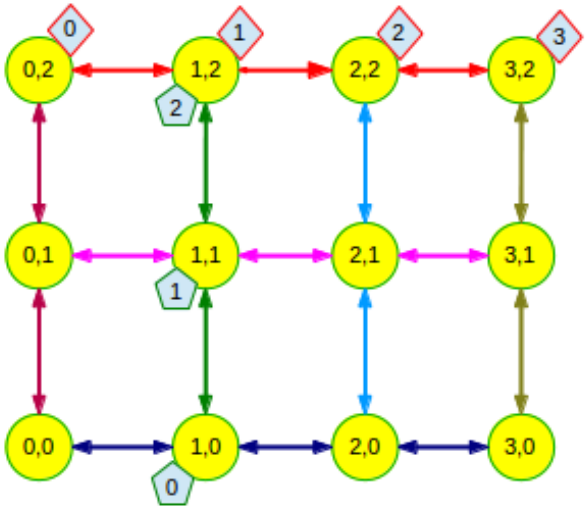
Other useful functions



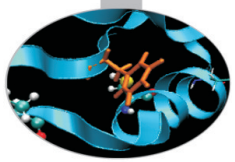
- MPI_Cart_shift:
  - locates the rank of the process to which send/from which receive data, for the function MPI_Sendrecv on a cartesian topology.



- MPI_Comm_split:
  - creates a sub-communicator for a subset of processes.



MPI_Cart_shift



C/C++

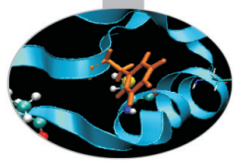
```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *source, int *dest)
```

Fortran

```
MPI_CART_SHIFT(coom, direction, disp, source, dest, ierr)
```

- *Input* arguments:
  - comm is the communicator with cartesian topology (**MPI_Comm**);
  - direction is the index of the coordinate along which do the shift (**integer**);
    - the numbering of indices starts from 0;
  - disp is the displacement of the shift (**integer**);
    - > 0: upwards shift, < 0: downwards shift;
- *Output* arguments:
  - source is the *rank* of the process from which receive data (**integer**).
  - dest is the *rank* of the process to which send data (**integer**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

MPI_Comm_split



C/C++

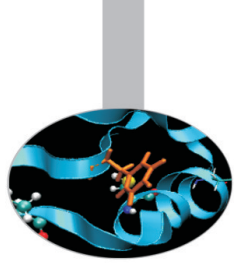
```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *subcomm)
```

Fortran

```
MPI_CART_SHIFT(coom, color, key, subcomm, ierr)
```

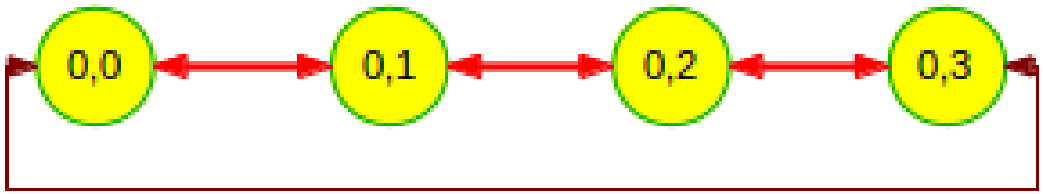
- *Input* arguments:
  - comm is the communicator to be split (**MPI_Comm**);
  - color is the control value of subset assignment (**integer**);
    - can't be negative;
  - key is the control value of rank assignment (**integer**);
- *Output* arguments:
  - subcomm is the sub-communicator (**MPI_Comm**);
  - ierr [*only Fortran*] is the error handler (**integer**):
    - with **use mpi** or **include 'mpif.h'** is needed;
    - with **use mpi_f08** is optional.

Circular shift on a cartesian topology 1D

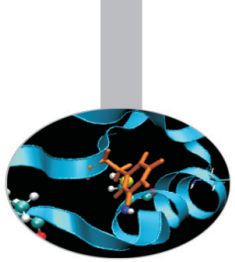


C/C++

```
MPI_Cart_create(MPI_COMM_WORLD, 1, &size, periods, 0, &comm_cart);
MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
MPI_Sendrevc(A, MSIZE, MPI_INT, dest, tag, B, MSIZE, MPI_INT, source, tag,
             comm_cart, &status);
```

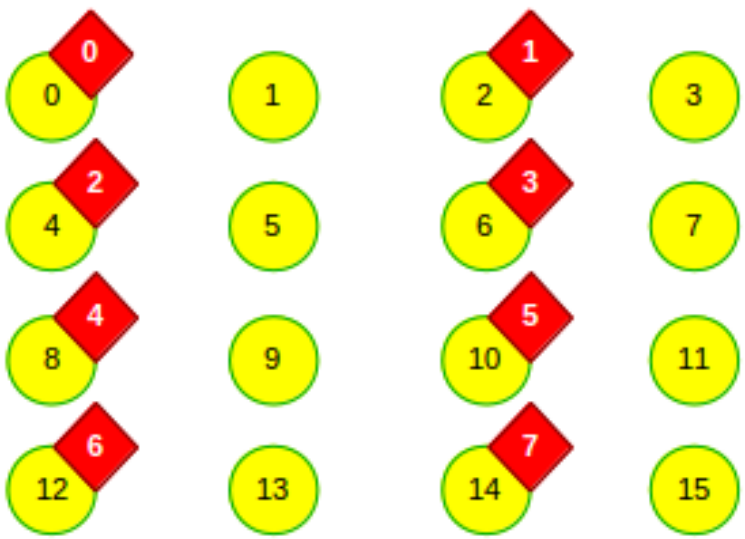


Sub-communicator for even numbers

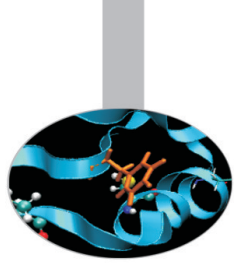


C/C++

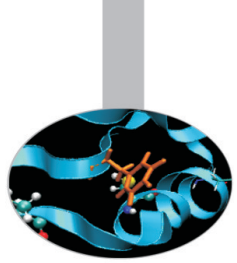
```
color = rank%2;
key = rank;
MPI_Comm_split(MPI_COMM_WORLD, color, key, &comm_even);
MPI_Comm_rank(comm_even, &rank_even);
```



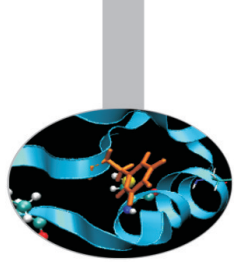
Outline



Outline

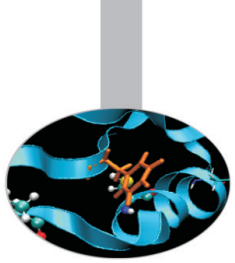


Other MPI functions exist!



- MPI_Win_create, MPI_Win_free;
- MPI_Win_allocate (`MPI-3.0`);
- MPI_Win_attach, MPI_Win_detach (`MPI-3.0`);
- MPI_Get, MPI_Put;
- MPI_Accumulate;
- MPI_Win_fence;
- ...

Thanks to ...



- Giorgio Amati
- Luca Ferraro
- Federico Massaioli
- Sergio Orlandini
- Marco Rorro
- Claudia Truini
- ... a lot of people!!!

Cheers

Thanks!