Advanced MPI
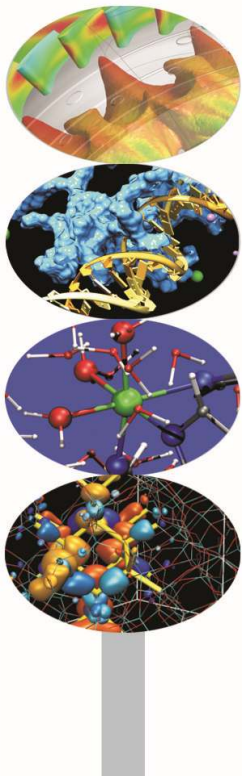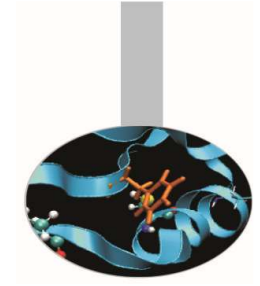
Maurizio Cremonesi

June 2017

Content

Packing data

Derived data types

MPI-IO introduction

Groups of processes

Communicators

Topologies

Pack/Unpack

Example: *0301MPIexample-pack*

Consider the problem of sending data of different kinds. For example, root should broadcast the following values:

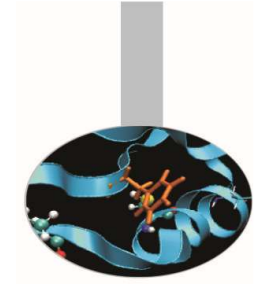MPI_DOUBLE_PRECISION: SWV(2), Range

MPI_INTEGER: XYdots, Niter

the broadcasting function should be called 4 times:

```
CALL MPI_BCAST (SWV, 2, MPI_DOUBLE_PRECISION , 0,
    MPI_COMM_WORLD, ierr)

CALL MPI_BCAST (XYdots, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

CALL MPI_BCAST (Range, 1, MPI_DOUBLE_PRECISION, 0,
    MPI_COMM_WORLD, ierr)

CALL MPI_BCAST (NITER, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

Pack/Unpack

A better solution is possible, minimizing the use of sending/receiving functions. The four objects:

MPI_DOUBLE_PRECISION: SWV(2), Range ! 8*3 = 24 bytes

MPI_INTEGER: XYdots, Niter ! 2*4 = 8 bytes

can be packed into a buffer for delivery and unpacked on receiving:

1. root process packs data into the buffer pckd_data, 32 bytes long:
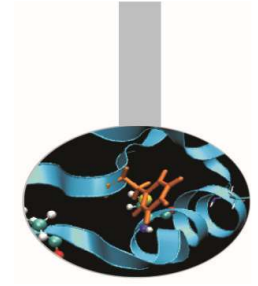
```
Pos = 0
CALL MPI_PACK (SWV, 2, MPI_DOUBLE_PRECISION, &
   &          pckd_data, 32, pos, MPI_COMM_WORLD, ierr)
    .   .   .
CALL MPI_PACK (Niter, 1, MPI_INTEGER, pckd_data, &
   & 32, pos, MPI_COMM_WORLD, ierr)
```

Pack/Unpack

2. buffered data are distributed:

```
CALL MPI_BCAST (pckd_data, 32, MPI_PACKED , 0, &
      &        MPI_COMM_WORLD, ierr)
```

3. receiving processes unpack data:
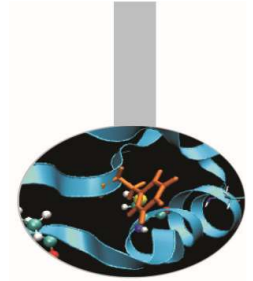
```
Pos = 0
CALL MPI_UNPACK (pckd_data, 32, pos, SWV, 2, &
      &        MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
 .   .   .
CALL MPI_UNPACK (pckd_data, 32, pos, Niter, 1, &
      &        MPI_INTEGER, MPI_COMM_WORLD, ierr)
```
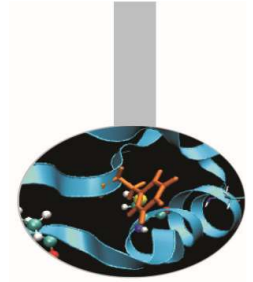
Pack

The MPI library enables packing different data in one buffer, that can be sent as a whole. Communication times can thus be reduced. To gather several data in a single buffer the function `MPI_PACK` may be used.

```fortran
INTERFACE                                                        fortran
  SUBROUTINE MPI_PACK(inbuf,incount,datatype,outbuf,outsize,position,comm,ierr)
    INTEGER, INTENT(IN) :: INCOUNT, DATATYPE, OUTSIZE, COMM
    <type>, INTENT(IN) :: INBUF(:)
    <type>, INTENT(OUT) :: OUTBUF(:)
    INTEGER, INTENT(INOUT) :: POSITION
    INTEGER, INTENT(OUT) :: IERR
  END SUBROUTINE MPI_PACK
END INTERFACE
```

```c
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf,
        int outsize, int *position, MPI_Comm comm);
                                                                    c/c++
```

INCOUNT elements of type DATATYPE of the buffer INBUF are copied in the buffer OUTBUF from position POSITION (in byte). On exit POSITION has the value of the next free address.

Unpack

The function `MPI_UNPACK` is used by the receiving processes to extract data from the buffer INBUF.

```fortran
INTERFACE
    SUBROUTINE MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype,
        comm, ierr)
        INTEGER, INTENT(IN) :: INSIZE, DATATYPE, OUTCOUNT, COMM
        <type>, INTENT(IN) :: INBUF(:)
        <type>, INTENT(OUT) :: OUTBUF(:)
        INTEGER, INTENT(INOUT) :: POSITION
        INTEGER, INTENT(OUT) :: IERR
    END SUBROUTINE MPI_UNPACK
END INTERFACE
```
fortran

```c/c++
int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm)
```
c/c++
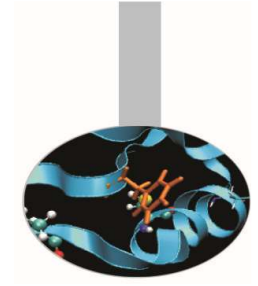
Basic data types

The basic MPI data types are:

```
MPI_INTEGER                              fortran
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_DOUBLE_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
MPI_BYTE
MPI_PACKED
```

```
MPI_CHAR                                 c/c++
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED
```

If the data to be communicated are structures with heterogeneous members, it may be convenient to define a MPI derived data type. Derived data types are defined using basic data types and formerly defined derived data types.

```
Dtype = [(typ_0 , pos_0), (typ_1 , pos_1), ..., (typ_n-1 , pos_n-1)]
```

Derived data types
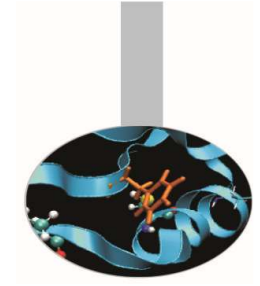
To define a derived data type it is required:

- To specify the structure of the new data type, on the basis of previously defined or basic data types.

- To register the new data type

The newly defined data type must be registered to MPI with the following function:

```fortran
interface
    subroutine mpi_type_commit (mpi_mytype, cod_err)
        integer, intent (in) :: mpi_mytype ! Il nome del nuovo tipo di dati
        integer, intent (out):: cod_err    ! codice di errore.
    end subroutine mpi_type_commit
end interface
```

```c/c++
int MPI_Type_commit ( MPI_Datatype *mpi_mytype )
```

Once committed the new data type becomes a recognized MPI data type.

Derived data types

Whenever a defined data type is of no use any more, the following function should be called:

```fortran
interface
    subroutine mpi_type_free (mpi_mytype, cod_err)
        integer, intent (in) :: mpi_mytype ! Data type handler
        integer, intent (out):: cod_err    ! Error code
    end subroutine mpi_type_commit
end interface
```
fortran

```c
int MPI_Type_free ( MPI_Datatype *mpi_mytype )
```
c/c++

Pending operations will complete normally.

Generic structures

Example: *0302MPIexample-struct*

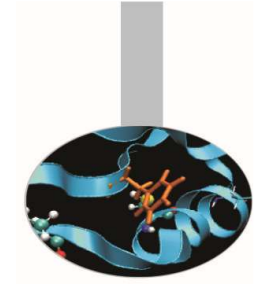Consider the problem of sending heterogeneous data contained in a structure:

```
TYPE input_data                    struct {
   REAL(8) :: SWV(2), Range            double SWV[2], Range;
   INTEGER :: XYdots, Niter            int Xydots, Niter;
END TYPE                           } input_data;
```

The sending function should be called 4 times:

```
CALL MPI_BCAST (values%SWV, 2, MPI_DOUBLE_PRECISION , 0, &
       &        MPI_COMM_WORLD, ierr)

              .       .        .

CALL MPI_BCAST (values%NITER, 1, MPI_INTEGER, 0, &
       &        MPI_COMM_WORLD, ierr)
```

Generic structures

A better solution is possible, minimising the communication calls.

A MPI derived data type is built from blocks of homogeneous elements. For example, if we are interested in communicating the following data structure:

```
TYPE input_data
    REAL(8) :: SWV(2), Range
    INTEGER :: XYdots, Niter
END TYPE
```

```
struct {
    double SWV[2], Range;
    int Xydots, Niter;
} input_data;
```
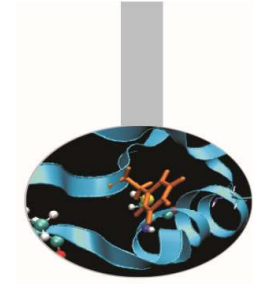
four blocks may be defined, one for each component of the structure:

Block #1: 2 doubles   Block #2: 1 double

Block #3: 1 integer   Block #4: 1 integer
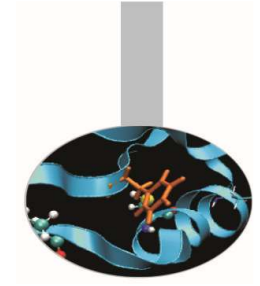
Generic structures

The MPI representation of a generic data structure requires three arrays to be defined:

`v_len_blk` – actual length of each block (in elements)

`v_head` – starting postion of each block (in bytes)

`v_el_typ` – data type of the elements in each block

It should be noted that the values in `v_head` must be given in *bytes* because the elements of each block may be of different type with different byte extensions.

Generic structures

The data structure can be described using the MPI_Type_struct function:

```
num_blk = 4
v_len_blk = [2,1,1,1]
v_head = [0,16,24,28]
v_el_typ = [MPI_DOUBLE_PRECISION, &
        & MPI_DOUBLE_PRECISION, &
        & MPI_INTEGER, MPI_INTEGER]
CALL MPI_Type_struct(num_blk, v_len_blk, v_head, &
        & v_el_typ, new_type, ierr)
CALL MPI_Type_commit(new_type,ierr)
```
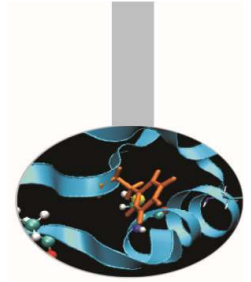
Generic structures

The function `mpi_type_struct` has the following interface:
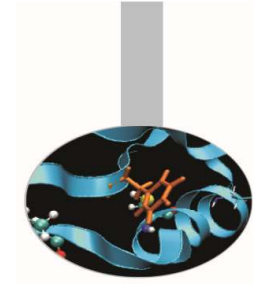
```fortran
interface
 subroutine mpi_type_struct(num_blk,v_len_blk,v_head,v_el_typ,new_typ,ierr)
   integer, intent(in) :: num_blk                   ! How many blocks
   integer,intent(in),dimension(:) :: v_len_blk    ! How many elements per block
   integer, intent(in), dimension(:) :: v_head     ! How many bytes before
                                                    !     each block
    integer, intent(in), dimension(:) :: v_el_typ ! Element type per block
    integer, intent(out) :: new_typ     ! Data type handler
    integer, intent(out) :: ierr     ! Error code
 end subroutine mpi_type_struct
end interface
```

```c/c++
int MPI_Type_struct( int num_blk, int v_len_blk[], MPI_Aint v_head[],
                     MPI_Datatype v_el_typ[], MPI_Datatype *new_typ )
```

Generic structures

C language structures and Fortran derived data types may be easily mapped into MPI derived data types.

The programmer must anyway be sure that the relative positions of the structure components are not modified by compiler optimizations.

Fortran derived types should contain the instruction SEQUENCE.
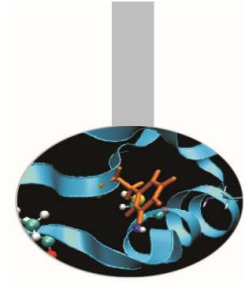
Contiguous elements

Example: *0303MPIexample-struct_gather*

Consider the problem of distributing a vector of structured data:

```
TYPE person

    SEQUENCE

    CHARACTER(80) :: Name, Surname

    INTEGER, DIMENSION(3) :: Birth_date

    INTEGER :: Position, Id

END TYPE person

TYPE(person), dimension(8) :: lteam, team
```

If we would like to distribute the *team* array, we could send it as a number of elements of type person (or whatever is called the object MPI_Datatype) or as a global object composed of elements of type person.

Contiguous elements

An array of contiguous and homogeneous elements is the simplest derived type to be defined. From element to element there must be no spaces.

```fortran
interface
    subroutine mpi_type_contiguous (num_el, el_type, new_type, ierr)
        integer, intent(in) :: num_el ! How many elements in the array
        integer, intent(in) :: el_type  ! Element type
        integer, intent(out) :: new_type ! New data type handler
        integer, intent(out) :: ierr ! Error code
    end subroutine mpi_type_contiguous
end interface
```
fortran

```c
int MPI_Type_contiguous ( int num_el, MPI_Datatype el_type,
                          MPI_Datatype *new_type)
```
c/c++

This function defines the new data type starting from an array of `num_el` elements. All the elements must be of the same (derived) data type.
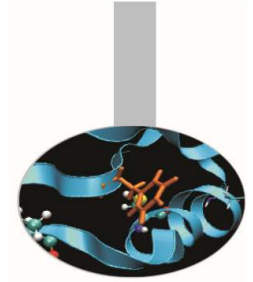
Contiguous elements

As an other simple example, if

```
El_type = {(double, 0), (char, 8)}
```

is a 16 bytes data type and 3 elements of that type are filed in an array, then

```
New_type = {(double,  0), (char,  8)
            (double, 16), (char, 24)
            (double, 32), (char, 40)}
```

Of course there is no point in *El_type* being a basic MPI type even if it may be as well.

Not contiguous elements

Making things a bit more complicated, the following function is used to define arrays with useful data separated by fixed strides. i.e. arrays may be seen as sequences of identical blocks containing elements to be communicated and elements to be discarded

```fortran
interface
   subroutine mpi_type_vector(num_blk,len_blk,blk_siz,el_typ,new_typ,errcode)
       integer, intent(in) ::  num_blk  ! How many blocks
       integer, intent(in) ::  len_blk  ! How many useful elements per block
       integer, intent(in) ::  blk_siz  ! Total number of elements per block
       integer, intent(in) ::  el_typ   ! Data type of the block elements
       integer, intent(out) :: new_typ  ! New data type handler
       integer, intent(out) :: ierr     ! Error code
   end subroutine mpi_type_vector
end interface
```
fortran

```c
int MPI_Type_vector( int num_blk, int len_blk, int blk_siz,
                MPI_Datatype el_typ, MPI_Datatype *new-typ )
```
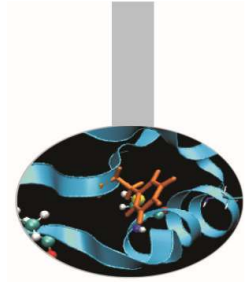*c/c++*

Not contiguous elements
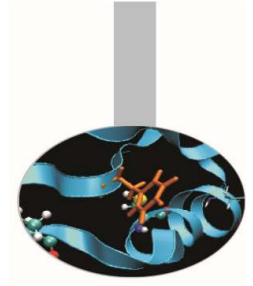
It can be noted that the size and the useful length of the blocks is given in number of elements.

As an example if blk_siz=10 and len_blk=7 and the elements are of type `MPI_INTEGER`, the actual size of each block is 4 x 10 = 40 bytes. But only 4 x 7 = 28 elements are communicated and 4 x (10-7) = 12 bytes are never sent.

More on not contiguous elements

The following function must be used to define arrays with blocks of different dimensions. Two vectors are needed to define the lengths because each block has its own number of useful and discarded elements.

```fortran
interface
 subroutine mpi_type_indexed(num_blk,v_len_blk,v_head,el_typ,new_typ,cod_er)
   integer, intent(in) :: num_blk      ! How many blocks
   integer, intent(in), dimension(:) :: v_len_blk  ! How many elements
                                                   ! in each block
   integer, intent(in), dimension(:) :: v_head    ! How many elements before
                                                  ! each block
   integer, intent(in) :: el_typ    ! Data type of elements in each block
   integer, intent(out) :: new_typ  ! New data type handler
   integer, intent(out) :: ierr     ! Error code
 end subroutine mpi_type_indexed
end interface
```
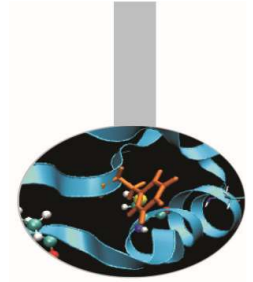*fortran*

```c
int MPI_Type_indexed( int num_blk, int v_len_blk[], int v_head[],
                MPI_Datatype el_typ, MPI_Datatype *new_typ )
```
c/c++

Please note that instead of specifying the total length of each block, the starting position of the blocks have to be passed to the function.

More on not contiguous elements

As an example, if we have to describe data structured in three blocks, 3 elements parted each other and containing 5, 13 and 7 elements, the arrays v_len_blk and v_head must be defined as follow:

```
v_len_blk = (/ 5, 13,  7 /)
v_head    = (/ 0,  8, 24 /)
```

The following function may be used to know the extension of a MPI (either basic or derived) data type:

```fortran
interface
    subroutine mpi_type_extent (datatype, ext, cod_err)
        integer, intent(in) :: datatype    ! MPI data type
        integer, intent(out) :: ext        ! Extension (in bytes)
        integer, intent(out) :: ierr       ! Error code
    end subroutine mpi_type_extent
end interface
```
fortran

```
int MPI_Type_extent( MPI_Datatype datatype, MPI_Aint *ext )
```
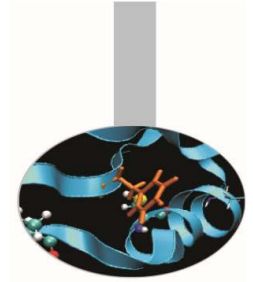
Useful functions

The function `mpi_type_hvector` is similar to `mpi_type_vector`, but `blk_siz` is given in bytes. The function `mpi_type_hindexed` is alike `mpi_type_indexed`, but `v_head` is measured in bytes.

The function `mpi_address` returns the starting address of an object. It is important for portability issues.

```fortran
interface
    subroutine mpi_address (obj, address, ierr)
        integer, intent(in) :: obj          ! Input object or variable
        integer, intent(out) :: address     ! Starting address
        integer, intent(out) :: ierr        ! Error code
    end subroutine mpi_address
end interface
```
fortran

```c
int MPI_Address( void *obj, MPI_Aint *address)
```
c/c++

MPI-IO basics

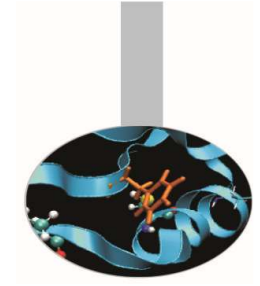Example: *0304MPIexample-struct_file*

The MPI derived data types may be used not only in communications but in I/O operations also.

As an example, imagine in the previous example to substitute collective file writing for the data distribution function:

```fortran
CALL MPI_File_open(MPI_COMM_WORLD, 'team.dat', &
    & MPI_MODE_WRONLY+MPI_MODE_CREATE, &
    & MPI_INFO_NULL, fh, ierr)

CALL MPI_File_write_ordered(fh, lteam, 1, &
    & pair_type, status, ierr)

CALL MPI_File_close(fh,ierr)
```
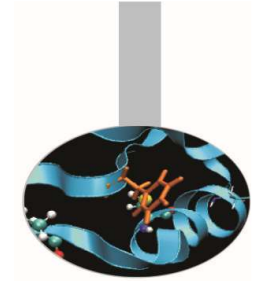
MPI-IO basics

After writing data is kept on disk and can be recovered when needed:

```fortran
IF ( my_rank == 0 ) THEN
    CALL MPI_FILE_OPEN(MPI_COMM_SELF, 'team.dat', &
        & MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
    DO i = 1, 8
        CALL MPI_File_read(fh, team(i), 1, &
            & person_type, status, ierr)
    ENDDO
    CALL MPI_File_close(fh,ierr))
ENDIF
```

MPI-IO basics

- basic MPI-IO operations are: open, seek, read, write, close

- open/close operations must be issued by all processes on the same file (collective operations)

- MPI read/write functions are similar to send/recv

- a local pointer to the file (individual file pointer) is kept for each process for seek, read, write operations

- MPI I/O is unformatted: data are saved on disk as they are stored in memory.

MPI-IO basics

Before beginning I/O operations the destination file must be connected to the MPI system. This is afforded by the MPI_File_open function. Remember that this function is collective: it must be called by all the processes in a communicator.

```fortran
interface
    subroutine MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)
        integer, intent(in) :: comm          ! Communicator
        character(*), intent(in) :: filename
         integer, intent(in) :: amode         ! Access mode
         integer, intent(in) :: info          ! Access details
         integer, intent(out) :: fh           ! File handle
         intent(out) :: ierr                  ! Error code
    end subroutine MPI_FILE_OPEN
end interface
```
fortran

```c
int MPI_File_open(MPI_Comm comm, const char* filename, int amode,
        MPI_Info info, MPI_File* fh)
```
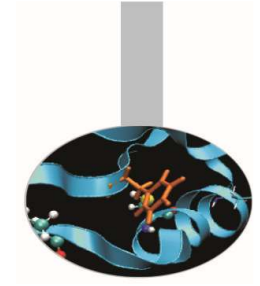*c/c++*

MPI-IO basics
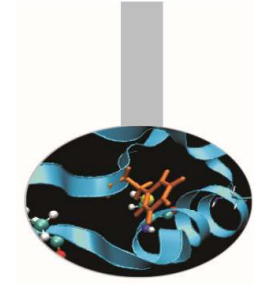
Few notes about the open function:

- the function is collective; would it be the case just one process has to call it, MPI_COMM_SELF should be used

- the filename must be the same for all involved processes

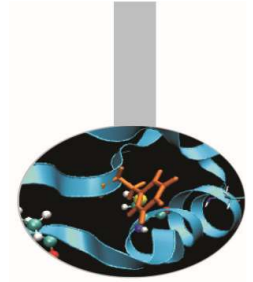- if the MPI_Info handler is not used, MPI_INFO_NULL value can be passed

MPI-IO basics

- the access mode value must be the same for all involved processes; some of the most common mode values are:

  MPI_MODE_RDONLY   read only

  MPI_MODE_RDWR   read/write

  MPI_MODE_WRONLY   write only

  MPI_MODE_CREATE   create if not existing

  MPI_MODE_DELETE_ON_CLOSE   delete on closing

  Mode values may be chained with + (plus, Fortran) and | (pipe, C, C++) characters

MPI-IO basics

After the destination file have been used, the MPI_File_close function should be called. Remember that this function also is collective: it must be called by all the processes in the communicator.
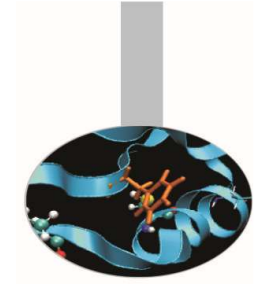
```fortran
interface
    subroutine MPI_FILE_CLOSE(fh, ierr)
        integer, intent(in) :: fh ! File handle
        intent(out) :: ierr                 ! Error code
    end subroutine MPI_FILE_CLOSE
end interface
```
fortran

```c
int MPI_File_close(MPI_File* fh)
```
c/c++

MPI-IO basics

There are several functions for storing and recovering data to/from disk.

Some functions are blocking, while others can overlap I/O with computation.

Besides operations can be collective or individually managed.

Positioning can be collectively or individually or explicitly managed.

In the previous example a writing collective function has been used to store data. In this case the operations is automatically managed by MPI and processes do not need to take care about data positioning in the file.

Instead data retrieving has been accomplished by calling a non collective function with individual pointer.
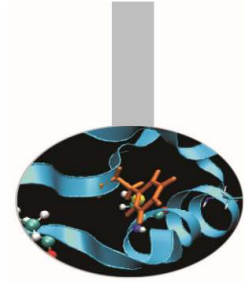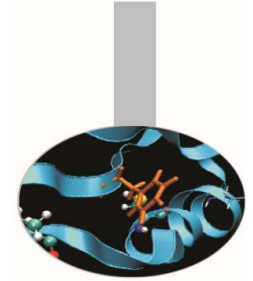
MPI-IO basics

The MPI_File_write_ordered function can be used to store distributed data in the process order.

```fortran
interface
    subroutine MPI_File_write_ordered(fh, buf, count, &
          & datatype, status, ierr)
        integer, intent(in) :: fh          ! File handle
        integer, intent(in) :: buf         ! Data (actual type may vary
        integer, intent(in) :: count       ! Elements in buffer
        integer, intent(in) :: datatype    ! MPI datatype of data
        integer, intent(out) :: status     ! Infos
        intent(out) :: ierr                ! Error code
    end subroutine MPI_File_write_ordered
end interface
```
fortran

```c
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
        MPI_Datatype datatype, MPI_Status *status)
```
c/c++

MPI-IO basics

The MPI_File_read function can be used to retrieve data individually, i.e. each process read data indipendently by all others.

```fortran
interface
    subroutine MPI_File_read(fh, buf, count, datatype, &
        &   status, ierr)
        integer, intent(in) :: fh          ! File handle
        integer, intent(out) :: buf        ! Data (actual type may vary)
        integer, intent(in) :: count       ! Elements in buffer
        integer, intent(in) :: datatype    ! MPI datatype of data
        integer, intent(out) :: status     ! Infos
        intent(out) :: ierr                ! Error code
    end subroutine MPI_File_read
end interface
```
fortran

```c
int MPI_File_read(MPI_File fh, void *buf, int count,
        MPI_Datatype datatype, MPI_Status *status)
```
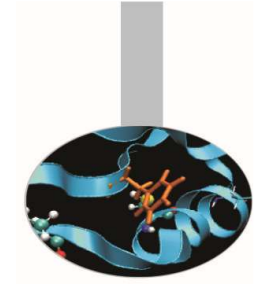*c/c++*

MPI-IO basics

The status object returned by I/O functions may be used to control data movements. In the example the MPI_Get_count has been used to show how many elements have been written in each writing operation.

```fortran
interface
    subroutine MPI_Get_count(status, datatype, count, ierr)
        integer, intent(in) :: status      ! I/O infos
        integer, intent(in) :: datatype    ! MPI datatype of data
        intent(out) :: count               ! Elements moved
        intent(out) :: ierr                ! Error code
    end subroutine MPI_Get_count
end interface
```
fortran

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```
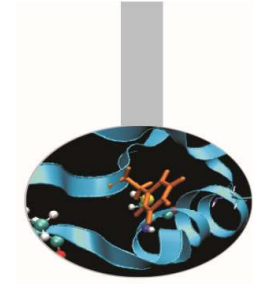*c/c++*

MPI-IO basics

There are many other MPI functions for saving and retrieving data on disk; here is just a list of some.

Blocking, individual file pointer:

- int **MPI_File_write**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

and the collective versions:

- int **MPI_File_write_all**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read_all**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
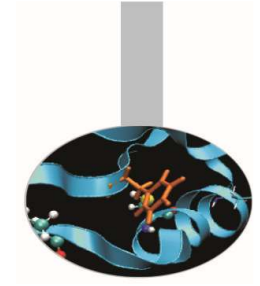
MPI-IO basics

Blocking, shared file pointer:

- int **MPI_File_write_shared**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read_shared**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

and the collective versions:

- int **MPI_File_write_ordered**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read_ordered**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

MPI-IO basics

Blocking, explicit file offset:

- int **MPI_File_write_at**(MPI_File fh, MPI_Offset offset, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read_at**(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
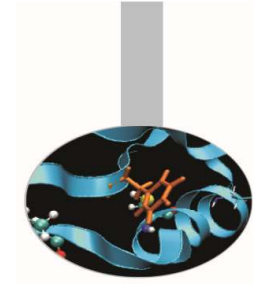
and the collective versions:

- int **MPI_File_write_at_all**(MPI_File fh, MPI_Offset offset, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- int **MPI_File_read_at_all**(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

Besides these functions there are the *non-blocking* versions.

Groups of processes
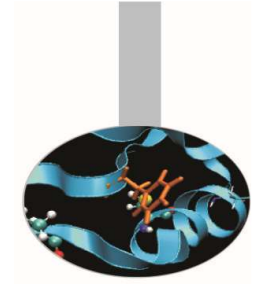
In MPI terminology the process is the computing unit. MPI processes behave following the MIMD model. Each process is an indipendent unit and has its own memory space; it should be thought of as running on its own computing machine.

Groups of processes

Every MPI process belongs to one or more MPI group and has its own identification number or rank. MPI ranks are always numbered starting from 0. The 0 process is often called the *master* and usually acts as the boss in master-slave programming model, but it is not mandatory: any other process can act the same.

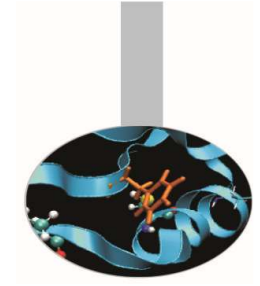MPI groups may be generated and destroyed but they are otherwise static.

Groups of processes

Example: *0305MPIExample_two_groups*

In this example the group related to the default communicator is splitted in two groups using an array of indices:

```
numproc0 = numproc/2;

for ( i = 0; i < numproc0; i++ ) ranks0[i] = i;

MPI_Group_incl(GlobalGroup, numproc0, ranks0, &Group0);

MPI_Group_excl(GlobalGroup, numproc0, ranks0, &Group1);

MPI_Group_size(Group1, &numproc1);
```

Groups of processes

Each group has its own handle but it is an opaque object: the programmer can not access its details. Proper functions must be used to manage group properties:

```
call mpi_group_size(group, size, ierr)
call mpi_group_rank(group, rank, ierr)
```

At the beginning all processes belong to the default group, the one associated to the default communicator `MPI_COMM_WORLD`. All other groups must be explicitly generated.

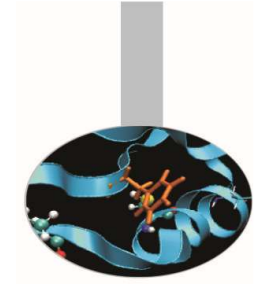MPI processes may belong to different groups.

Groups of processes

Given a communicator the following function returns the handle of the associated group:

```fortran
interface
    subroutine mpi_comm_group(comm,group, ierr)
        integer, intent(in) :: comm
        integer, intent(out) :: group, ierr
    end subroutine mpi_comm_group
end interface
```
fortran

```c
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```
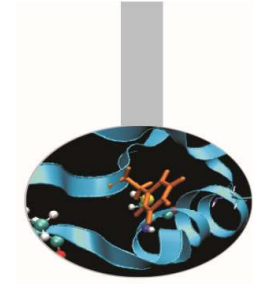*c/c++*

Managing groups of processes

The following function enables generating a new group on the basis of an existing group. The process with rank `RANKS(I)` in the old group is given rank `I` in the new group:

```fortran
interface
    subroutine mpi_group_incl(group, n, ranks, newgroup, ierr)
        integer, intent(in) :: group, n, ranks
        integer, intent(out) :: newgroup, ierr
    end subroutine mpi_group_incl
end interface
```
fortran

```c
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```
c/c++

Managing groups of processes

Example:

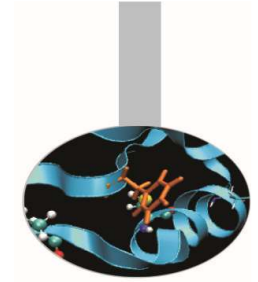if `GROUP` contains 8 processes (numbered from 0 to 7) and the array has values `RANKS(1:3)=(1,5,2)`, the instruction

```fortran
call mpi_group_incl (group, 3, ranks, newgroup, ierr)
```

generates the new `NEWGROUP` with the three processes above numbered.

The following table shows correspondence between the two groups:

Group	Newgroup
1	0
5	1
2	2

Managing groups of processes

On the contrary in the following function the array `RANKS(I)` specify the processes of `GROUP` to be eliminated for building `NEWGROUP`:

```fortran
interface
    subroutine mpi_group_excl(group, n, ranks, newgroup, ierr)
        integer, intent(in) :: group, n, ranks
        integer, intent(out) :: newgroup, ierr
    end subroutine mpi_group_excl
end interface
```
*fortran*

```c
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```
c/c++

It is also possible to specify a range of indexes, like `RANGES(1:N,1:3)`. In the following functions the second dimension of the array specifies the first and the last index to be included and the stride.

```fortran
call mpi_group_range_incl (group, n, ranges, newgroup, ierr)
```
fortran

```fortran
call mpi_group_range_excl (group, n, ranges, newgroup, ierr)
```
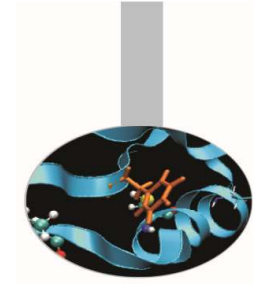*fortran*
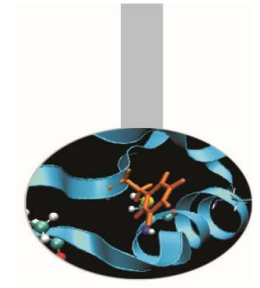
Managing groups of processes

Example:

If group contains 1000 processes and a new group is to be generated with half the number of the processes, taken from the odd positions, the array may be defined as `RANGES(1,1)=2, RANGES(1,2)=1000, RANGES(1,3)=2` and the program should issue the following instruction:

```fortran
call mpi_group_range_excl (group, n, ranges, newgroup, ierr)
```

The correspondence between the two groups would be:

Group	Newgroup
1	0
3	1
5	2
...	...

Managing groups of processes

The operations to manage groups are local and do not involve communications.

The following function may be used to know the relevant rank of the processes in two different groups.

RANKS1(:) are the known ranks of the processes in GROUP1

RANKS2(:) are the related ranks in GROUP2

```fortran
interface
    subroutine mpi_group_translate_ranks(group1, n, ranks1, group2, &
        ranks2, ierr)
        integer, intent(in) :: group1, n, ranks1(:), group2
        integer, intent(out) :: ranks2(:), ierr
    end subroutine mpi_group_translate
end interface
```
fortran

```c
int MPI_Group_translate (group1, n, ranks1, group2, ranks2, ierr)
```
c/c++

Managing groups of processes

It is possible to check similarity of two groups:

```fortran
interface
    subroutine mpi_group_compare(group1, group2, result, ierr)
        integer, intent(in) :: group1, group2
        integer, intent(out) :: result, ierr
    end subroutine mpi_group_compare
end interface                                                    fortran
```
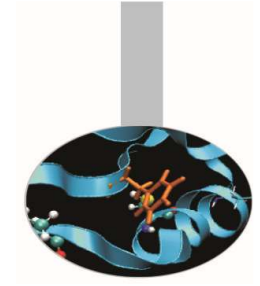
```c
int MPI_Group_compare (group1, group2, result, ierr)            c/c++
```

The returned values may be one out of the following:

- `MPI_IDENT` if the groups have the same processes with identical ranks

- `MPI_SIMILAR` if the groups have the same processes but unequal ranks

- `MPI_UNEQUAL` if the groups are different

Communicators

A communicator defines the processes that can communicate each other. Each communicator has its own handle, is an opaque object and can be managed by proper functions only.

The default communicator is named `MPI_COMM_WORLD`, but in a real program it is often useful to generate additional communicators, to be able to directly manage communications among process subsets.

Communicator handles must always be specified in sending and receiving functions.
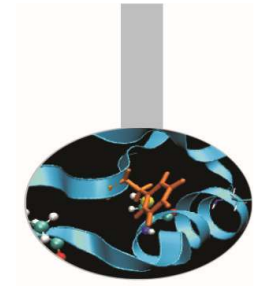
Communicators

Example: *0306MPIExample_two_comms*

The default communicator is splitted in two sets of processes and an intercommunicator is created:

```
colour = id % 2;
MPI_Comm_split(GlobalComm, colour, id, &LocComm);
if ( colour == 0 ) {
    MPI_Intercomm_create(LocComm, 0, GlobalComm, 1,
        01, &InterComm);
} else {
    MPI_Intercomm_create(LocComm, 0, GlobalComm, 0,
        01, &InterComm);
}
```
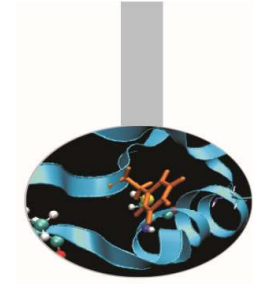
Managing communicators

The following function may be used to generate a new communicator connected to an existing group:

```fortran
interface
    subroutine mpi_comm_create(comm, group, newcomm, ierr)
        integer, intent(in) :: comm, group
        integer, intent(out) :: newcomm, ierr
    end subroutine mpi_comm_create
end interface
```
*fortran*

```c
int MPI_Comm_create ( MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm )
```
c/c++

- COMM is an existing communicator related to a wider process group; the function must be called by all processes in the COMM communicator

- GROUP is a sub-group of the process group related to COMM

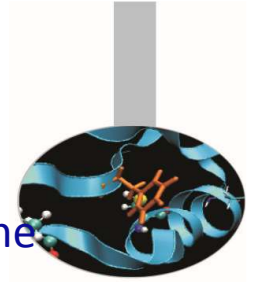- NEWCOMM is the handle of the newly generated communicator

Managing communicators

Suppose there is a communicator connected to a group of 8 processes and 2 new communicators are required by dividing the communicator in two parts as follow:
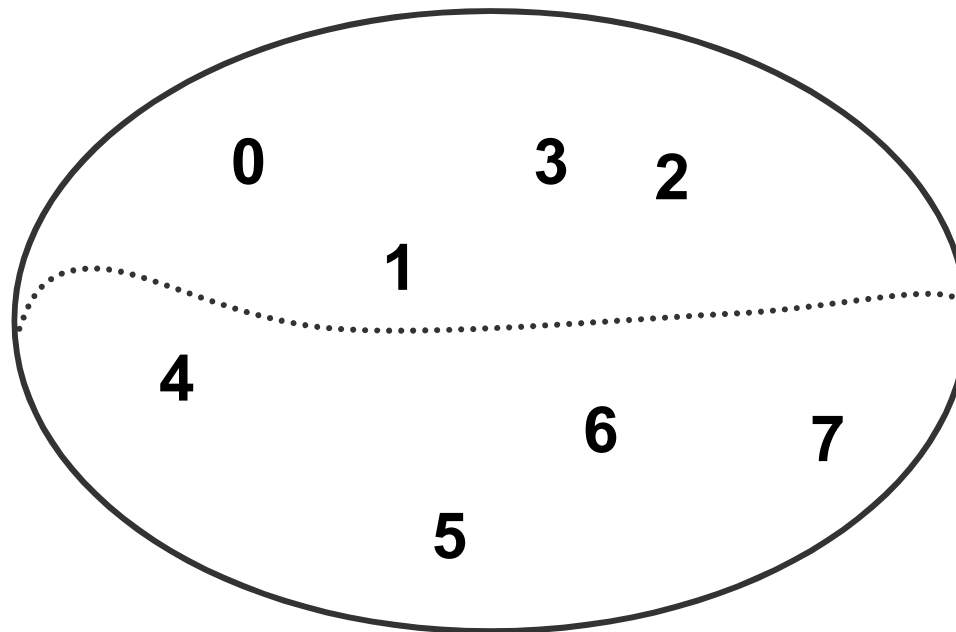
Managing communicators

To accomplish this task all the processes of the existing communicator may issue the following instructions:

```fortran
call mpi_comm_rank (comm, rank, ierr)
call mpi_comm_size (comm, size, ierr)
color = 2*rank/size
key   = size - rank - 1
call mpi_comm_split (comm, color, key, newcomm, ierr)
```
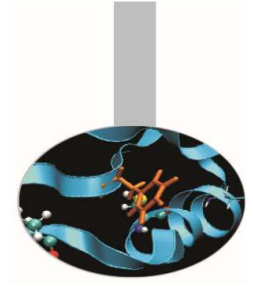
Each process receives a new communicator handle and will have the rank:

Communicator 1		Communicator 2	
Rank in new group	Rank in old group	Rank in new group	Rank in old group
0	3	0	7
1	2	1	6
2	1	2	5
3	0	3	4

If for some process COLOR=MPI_UNDEFINED, the function MPI_COMM_SPLIT returns NEWCOMM=MPI_COMM_NULL
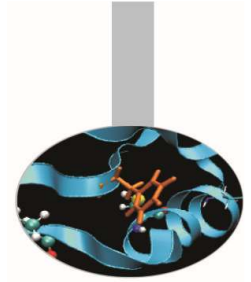
Communications between groups

Once the processes have been separated in several groups it is possible to realize client-server connections by connecting disjoined groups.

Communications between separated groups can only be of point-to-point type: no collective communications are available.

Communications between groups

Whenever a new inter-communicator has been created, the sending process must specify the rank of the receiving process (relevant to the other group); the receiving process must specify the rank of the sender (relevant to the other group).

To enable this, while the functions `mpi_comm_size`, `mpi_comm_rank`, `mpi_comm_group` return informations relevant to the local communicator, the functions `mpi_comm_remote_size`, `mpi_comm_remote_group` return informations on the disjoined intercommunicator group instead.

Communications between groups
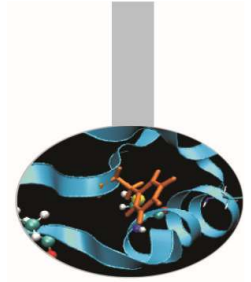
A communicator connecting disjoined groups is called an inter-communicator and can be generated by calling the function `mpi_intercomm_create`. This function requires:

- A leading process for each one of two disjoined groups

- An intra-communicator between the two leading processes

- A tag for safe communications between the two leading processes
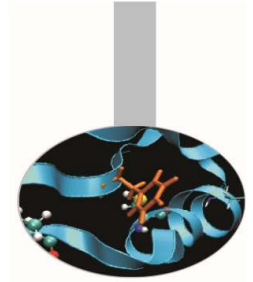
Communications between groups
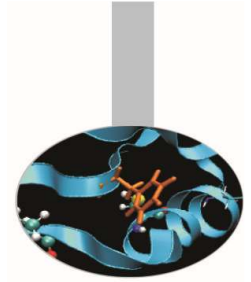
The following function generates an inter-communicator `NEWINTERCOMM` between the processes `LOCALLEADER` and `REMOTELEADER` of the intra-communicator `LOCALCOMM`, using `TAG` and the point-to-point communicator `PEERCOMM`. It should be noted that `REMOTELEADER` and `PEERCOMM` are referred to the local process, while `TAG` must have the same value for both the processes:

```fortran
interface
    subroutine mpi_intercomm_create(localcomm, localleader, peercomm, &
                                    remoteleader, tag, newintercomm, ierr)
        integer, intent(in) :: localcomm, localleader, peercomm
        integer, intent(in) :: remoteleader, tag
        integer, intent(out) :: newintercomm, ierr
    end subroutine mpi_intercomm_create
end interface
```
fortran

```c/c++
int MPI_Intercomm_create ( MPI_Comm localcomm, int localleader,
                           MPI_Comm peercomm, int remoteleader, int tag,
                           MPI_Comm *newintercomm )
```

Communications between groups

The intra-communicator `NEWINTRACOMM` may be generated from an inter-communicator `INTERCOMM` calling the function:
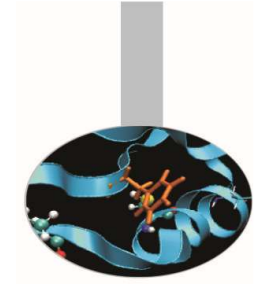
```fortran
interface
    subroutine mpi_intercomm_merge(intercomm, high, newintracomm, ierr)
            integer, intent(in) :: intercomm, high
            integer, intent(out) :: newintracomm, ierr
     end subroutine mpi_intercomm_merge
end interface
```
fortran

```
int MPI_Intercomm_merge(MPI_Comm intercomm,int high,MPI_Comm *newintracomm)
```
c/c++

This way two separated groups may be joined. The value of `HIGH` must be the same for all the processes belonging to the same group. If `HIGH = .FALSE.` for group 1 and `HIGH = .TRUE.` for group 2, in the new communicating group the processes are ordered starting from group 1; i.e. the processes in group 2 have a higher rank.

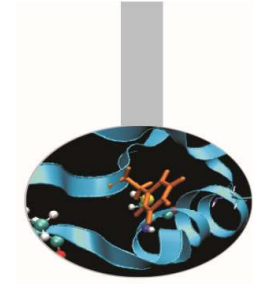Example: *0307MPIExample_comms_merge* (Fortran and C)

Topologies

In many programs it may be important to arrange the processes in a given topology. MPI enables the definition of topologies, with an explicit support for cartesian topology. This topology may be defined by calling the function:

```fortran
interface
    subroutine mpi_cart_create(comm_old, ndims, ldims, periods, reorder,
                                comm_cart, ierr)
            integer, intent(in) :: comm_old, ndims
            integer, dimension(:), intent(in) :: ldims
            logical, dimension(:), intent(in) :: periods
            logical, intent(in) :: reorder
            integer, intent(out) :: comm_cart, ierr
        end subroutine mpi_cart_create
    end interface
```
fortran

```c
int MPI_Cart_create ( MPI_Comm comm_old, int ndims, int *ldims, int *periods,
                      int reorder, MPI_Comm *comm_cart )
```
*c/c++*

Topologies

Example: *0308MPIExample_cart_create*

In the example a 2D cartesian topology is created to send a message along horizontal and vertical «bands».

```fortran
ldims(:) = q
periods(:) = .TRUE.
reorder = .FALSE.
call MPI_Cart_Create (MPI_COMM_WORLD, 2, ldims, periods, &
        & reorder, cart_comm, ierr)
!   Get process coordinates
call MPI_COMM_RANK(cart_comm,  cart_rank, ierr )
call mpi_cart_coords(cart_comm, cart_rank, 2, coords, &
        & ierr)
CALL MPI_CART_SHIFT(cart_comm, 0, 1, source, dest, ierr)
CALL MPI_SENDRECV_REPLACE(cval, 1, MPI_INTEGER, dest, 0, &
        & source, 0, cart_comm, status, ierr)
```
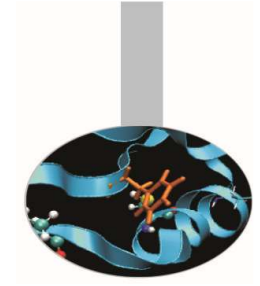
Topologies
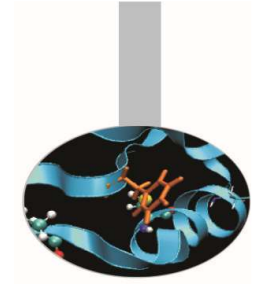
The `MPI_CART_CREATE` function returns the new communicator `COMM_CART`, connected to a grid with `NDIMS` dimensions. The extent of each dimension must be defined in `LDIMS(1:NDIMS)` and it is possible to specify periodicity for each dimension. The `REORDER` variable is used to allow reordering of the processes.

In cartesian topologies the processes are ordered by rows.

Functions dealing with informations and details about the topology associated to a communicator are available.

Topologies

Given a communicator `COMM`, the function `MPI_TOPO_TEST` returns the associated topology:

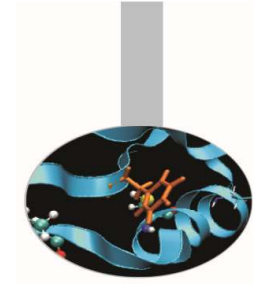`MPI_GRAPH`: graph topology

`MPI_CART`: cartesian topology

`MPI_UNDEFINED`: no topology

```fortran
interface
    subroutine mpi_topo_test(comm, topol, ierr)
            integer, intent(in) :: comm
            integer, intent(out) :: topol, ierr
    end subroutine mpi_topo_test
end interface
```
fortran

```c
int MPI_Topo_test ( MPI_Comm comm, int *topol )
```
c/c++

Topologies

Given a communicator `COMM`, with cartesian topology, the function `MPI_CARTDIM_GET` returns the number of dimensions

```fortran
interface
    subroutine mpi_cartdim_get(comm, ndims, ierr)
        integer, intent(in) :: comm
        integer, intent(out) :: ndims, ierr
    end subroutine mpi_cartdim_get
end interface
```
fortran

```c
int MPI_Cartdim_get ( MPI_Comm comm, int *ndims )
```
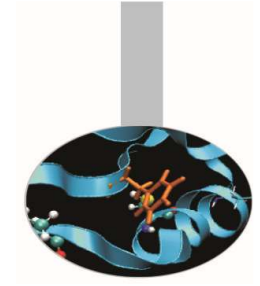*c/c++*

Topologies

The function `MPI_CART_GET`, returns the number `DIMS(:)` of processes in each dimension, the periodicity for each dimension, the process coordinates.

```fortran
interface
    subroutine mpi_cart_get(comm, maxdims, dims, periods, coords, ierr)
        integer, intent(in) :: comm, maxdims
        integer, intent(out) :: ierr
        integer, dimension(:), intent(out) :: dims, coords
        logical, dimension(:), intent(out) :: periods
    end subroutine mpi_cart_get
end interface
```
fortran

```c
int MPI_Cart_get ( MPI_Comm comm, int maxdims, int *dims,
                   int *periods, int *coords )
```
c/c++

Topologies

Given a communicator associated to a cartesian topology and the process coordinates, the following function returns the process rank:

```fortran
interface
    subroutine mpi_cart_rank(comm, coords, rank, ierr)
        integer, intent(in) :: comm
        integer, dimension(:), intent(in) :: coords
        integer, intent(out) :: rank, ierr
    end subroutine mpi_cart_rank
end interface
```
fortran

```c
int MPI_Cart_rank( MPI_Comm comm, int *coords, int *rank)
```
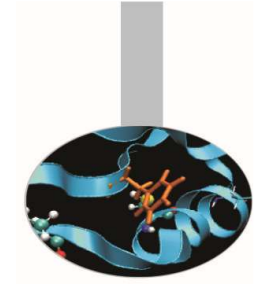*c/c++*

Topologies

The following function returns the coordinates of a process in a cartesian topology:

```fortran
interface
    subroutine mpi_cart_coords(comm, rank, maxdims, coords, ierr)
        integer, intent(in) :: comm, rank, maxdims
        integer, dimension(:), intent(out) :: coords
        integer, intent(out) :: ierr
    end subroutine mpi_cart_coords
end interface
```
fortran

```c
int MPI_Cart_coords( MPI_Comm comm, int rank, int maxdims, int *coords)
```
c/c++

Topologies

The following function helps defining the length in each dimension for a cartesian topology:

```fortran
interface
        subroutine MPI_Dims_create(nnodes, ndims, dims)
                integer, intent(in) :: nodes, dims
                integer, dimension(:), intent(inout) :: dims
                integer, intent(out) :: ierr
        end subroutine MPI_Dims_create
end interface
```
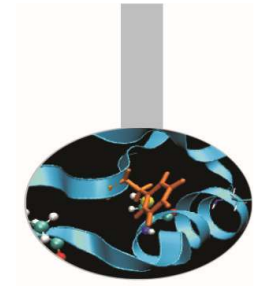*fortran*

```c/c++
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```
c/c++

The dims(:) array is filled by the function only where value is 0. As an example if nnodes=6, ndims=3, dims=[0,3,0], the output is dims=[2,3,1], otherwise the output is dims=[3,2,1]

Topologies

Topologies may be useful to send messages along specific directions.

As an example, suppose that every process in a cartesian topology has to send data toward the DIM dimension to a DELTA distance. The following function returns the ranks of the processes SOURCE and DEST

```fortran
interface
    subroutine mpi_cart_shift(comm, dim, delta, source, dest, ierr)
        integer, intent(in) :: comm, dim, delta
        integer, intent(out) :: source, dest, ierr
    end subroutine mpi_cart_shift
end interface
```
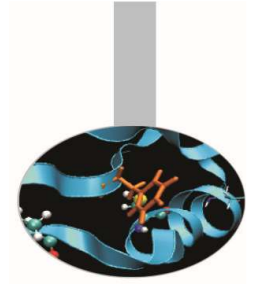*fortran*

```
int MPI_Cart_shift(MPI_Comm comm,int dim,int delta,int *source,int *dest)
```
c/c++

to be passed to the function
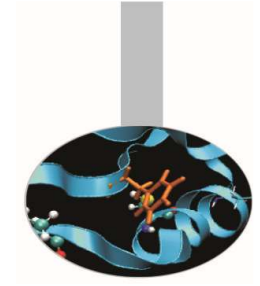
```
CALL MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, &
                  SENDTAG, RECVBUF,RECVCOUNT, RECVTYPE, &
                  SOURCE, RECVTAG, COMM, STATUS, IERROR)
```

Example: MPI_CART_SHIFT

```fortran
      ....
C find process rank
        CALL MPI_COMM_RANK(comm, rank, ierr))
C find cartesian coordinates
        CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
        CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
        CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
                status, ierr)
```

Topologies

The following function generates new cartesian topologies by cutting a wider cartesian space along the given dimensions:

```fortran
interface
    subroutine mpi_cart_sub(comm, remain_dims, newcomm, ierr)
        integer, intent(in) :: comm
        logical, dimension(:), intent(in) :: remain_dims
        integer, intent(out) :: newcomm, ierr
    end subroutine mpi_cart_sub
end interface
```
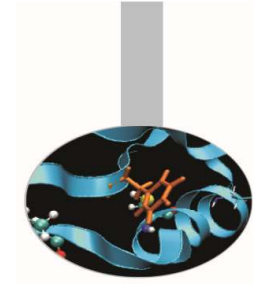*fortran*

```c
int MPI_Cart_sub( MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```
c/c++

Example: if `COMM` is associated to a cartesian topology with extensions 2x3x4 and `REMAIN_DIMS=(.T.,.T.,.F.)`, four new topologies are generated with extension 2x3.

Each process is returned one communicator handle; the former group is divided in 4 new groups with 6 processes each.

MPI+OpenMP

It is possible to develop parallel programs mixing MPI calls and OpenMP directives.

Intel compilers: mpi*xxx* -openmp -O3 -o nomefile.exe nomefile.*xxx*

PGI compilers: mpi*xxx* -mp -O3 -o nomefile.exe nomefile.*xxx*

GNU compilers: mpi*xxx* -fopenmp -O3 -o nomefile.exe nomefile.*xxx*

Execution:

export OMP_NUM_THREADS=*threads*

mpirun -np 2 -machinefile *mc* -x OMP_NUM_THREADS nomefile.exe