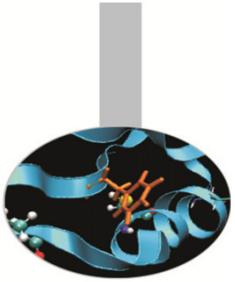
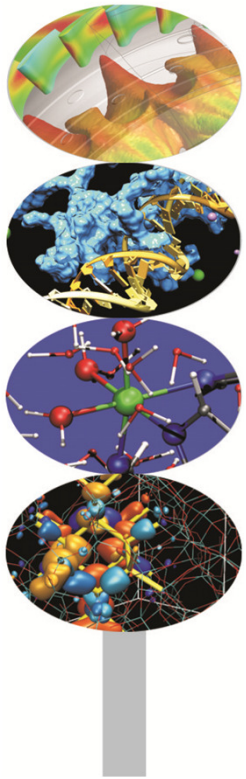
CINECA SCAI
SuperComputing Applications and Innovation

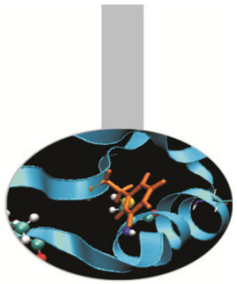


Advanced MPI



CINECA

Pack

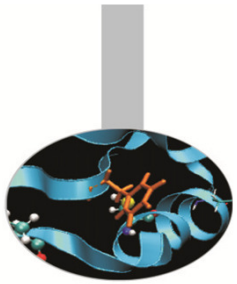


The MPI library enables packing different data in one buffer, that can be sent as a whole. Communication times can thus be reduced. To gather several data in a single buffer the function `MPI_PACK` may be used.

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

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.

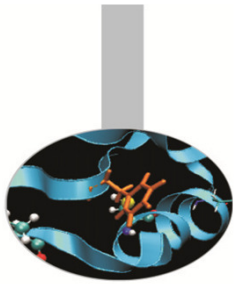
```fortran
INTERFACE
   SUBROUTINE MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype,
      comm, ierr)
      INTEGER, INTENT(IN) :: INSIZE, DATATYPE, OUTCOUNT, COMM
      <type>, INTENT(IN) :: INBUF(:)
      <type>, INTENT(OUT) :: OUTBUF(:)
      INTEGER, INTENT(INOUT) :: POSITION
   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

fortran

```c
int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf,
                  int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

c/c++

Derived data types



If the data to be communicated are structured it may be convenient to define a MPI derived data type. The basic MPI data types are:

fortran

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

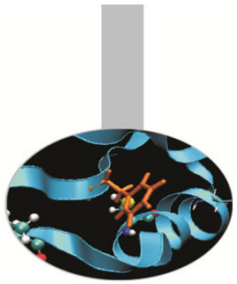
c/c++

```
MPI_CHAR
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
```

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

Starting backwards, to register a newly defined data type the following function is used:

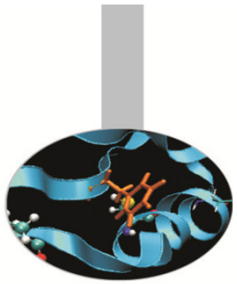
```fortran
interface
    subroutine mpi_type_commit (mpi_mytype, cod_err)
        integer, intent (in) :: mpi_mytype ! Il nome del nuovo tipo di dati
        integer, intent (out):: cod_err    ! codice di errore.
    end subroutine mpi_type_commit
end interface
```

fortran

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

c/c++

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



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
```

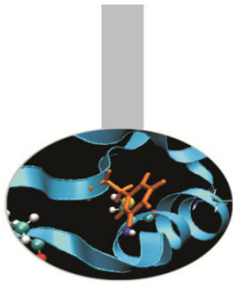
fortran

```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 data type.

Contiguous elements



As an example, if

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

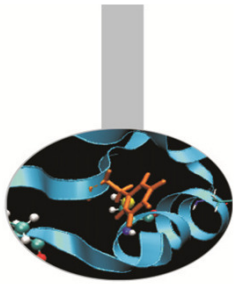
is a 16 bytes data type and 3 elements of that type are filed in an array, then

```
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.

Anyhow this function has been introduced first because of its simplicity.

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

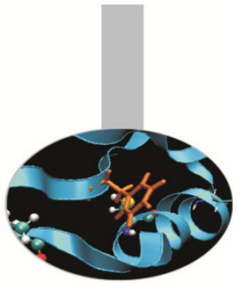
```fortran
interface
   subroutine mpi_type_vector(num_blk,len_blk,blk_siz,el_typ,new_typ,errcode)
       integer, intent(in) ::  num_blk  ! How many blocks
       integer, intent(in) ::  len_blk  ! How many useful elements per block
       integer, intent(in) ::  blk_siz  ! Total number of elements 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/c++
int MPI_Type_vector( int num_blk, int len_blk, int blk_siz,
                     MPI_Datatype el_typ, MPI_Datatype *new-typ )
```

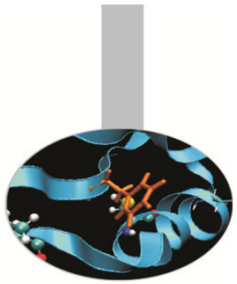
c/c++

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`, then the actual size of each block is 4 x 10 = 40 bytes but only 4 x 7 = 28 are communicated and 4 x (10-7) = 12 bytes are never sent.

Example: *block_array* (Fortran and C)



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
```

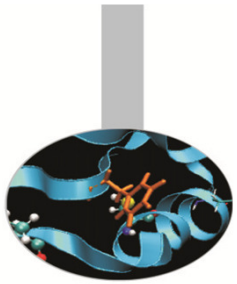
fortran

```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.

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 /)
```

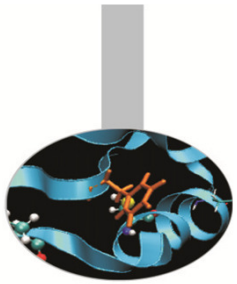
The following useful function may be used to know the extension of a MPI (either basic or derived) data type:

```
interface
    subroutine mpi_type_extent (datatype, dim, 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 )
```

c/c++



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.

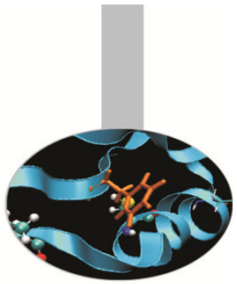
fortran

```
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
```

c/c++

```
int MPI_Address( void *obj, MPI_Aint *address)
```

Generic structures



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

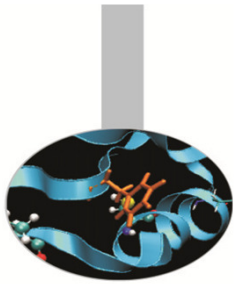
`v_len_blk` – actual length of each block

`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



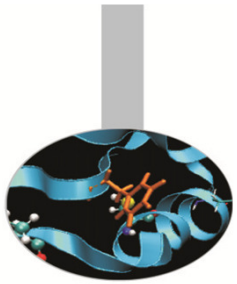
The function `mpi_type_struct` has the following interface:

```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
```

fortran

```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 )
```

c/c++



Generic structures

Example: the MPI description of the structure

2 x MPI_LOGICAL; 10 bytes space;

3 x MPI_DOUBLE_PRECISION; 10 bytes space;

9 x MPI_CHARACTER

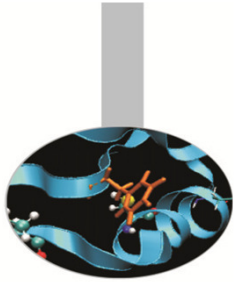
is defined assigning the values:

```
v_len_blk = (/ 2, 3, 9 /)
v_head    = (/ 0, 18, 52 /)
v_el_typ  = (/ MPI_LOGICAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER /)
```

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

The programmer must anyhow be sure that the relative positions of the structure components are not modified by compiler optimizations. For this reason Fortran derived types should contain the SEQUENCE instruction

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.

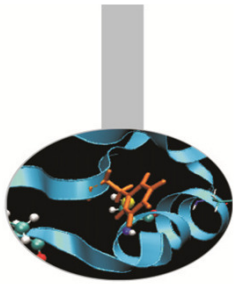
Every MPI process belongs to one or more MPI group and has its own identification number or rank. MPI ranks are always numbered starting from 0. The 0 process is often called the *master* and usually acts as the boss in master-slave programming model, but it is not mandatory. MPI groups may be generated and destroyed but they are otherwise static.

Each group has its own handle but it is an opaque object: the programmer can not access its details. Therefore 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

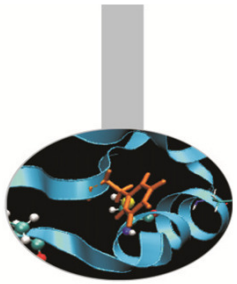


Given a communicator the following function returns the handle of the associated group:

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

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

Managing groups of processes

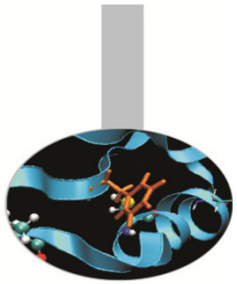


The following function enables generating a new group on the basis of an existing group. The process with rank `RANKS(I)` in the old group is given rank `I` in the new group:

```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
```

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

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

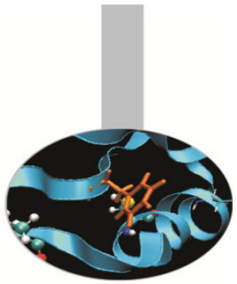
```fortran
call mpi_group_incl (group, 3, ranks, newgroup, ierr)
```

generates the new `NEWGROUP` with the three processes above re-numbered in the MPI conventional way.

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
```

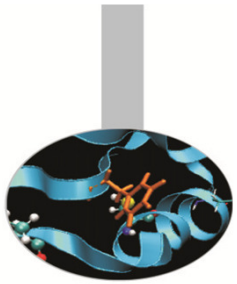
```c/c++
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

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 last index to be included and the stride.

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

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

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)=0, RANGES(1,2)=999, RANGES(1,3)=2`.

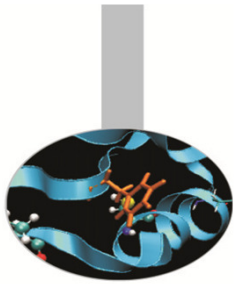
The processes should issue the following instruction:

```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.

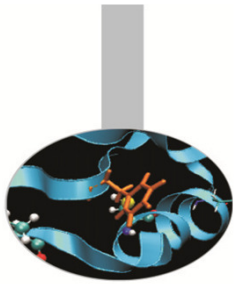
The following instruction may be used to know the relevant rank of the processes in two different groups.

`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
```

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

Managing groups of processes



It is possible to check the relation between two groups:

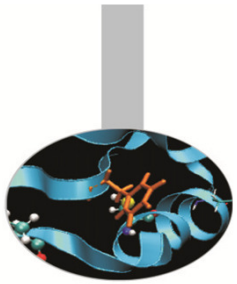
```fortran
interface
    subroutine mpi_group_compare(group1, group2, result, ierr)
        integer, intent(in) :: group1, group2
        integer, intent(out) :: result, ierr
    end subroutine mpi_group_compare
end interface
```

```c/c++
int MPI_Group_compare (group1, group2, result, ierr)
```

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

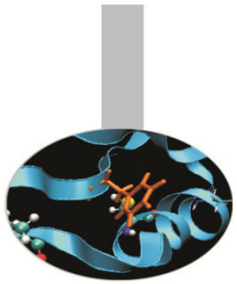


A communicator defines the processes that can communicate each other. Each communicator has its own handle, is an opaque object and can be managed by proper functions only.

The default communicator is named `MPI_COMM_WORLD`, but in a real program it is often useful to generate additional communicators. This may help managing communications among process subsets with increase of performance.

Communicator handles must always be specified in sending or receiving functions.

Managing communicators



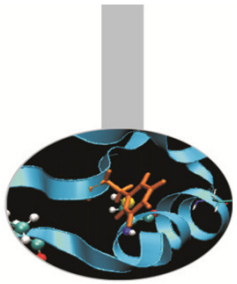
The following function may be used to generate a new communicator connected to an existing group:

```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
```

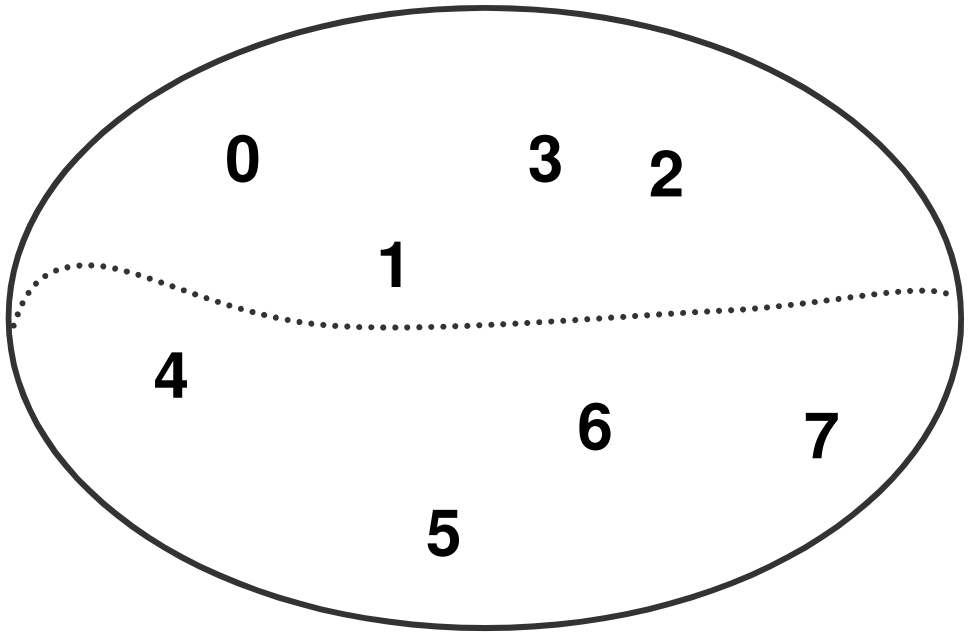
```c/c++
int MPI_Comm_create ( MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm )
```

- 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

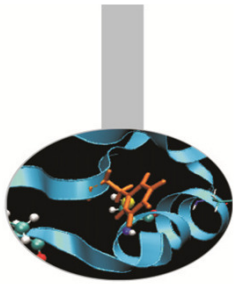
Managing communicators



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 should 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)
```

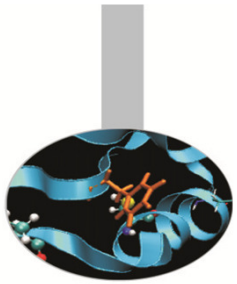
Each process receives a new communicator handle and will have the rank:

Communicator 1	
Rank in new group	Rank in old group
0	3
1	2
2	1
3	0

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

If for some process `COLOR=MPI_UNDEFINED`, the function `MPI_COMM_SPLIT` returns `NEWCOMM=MPI_COMM_NULL`

Communications between groups



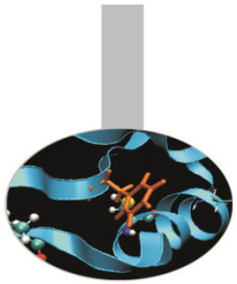
Once the processes have been separated in discrete groups it is possible to realize client-server connections by connecting disjoined groups.

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

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` instead return informations on the disjoined intercommunicator group.

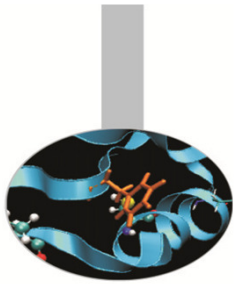
Communications between groups



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 the 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:

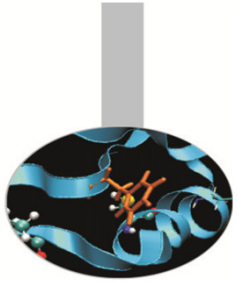
fortran

```
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
```

c/c++

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

Communications between groups



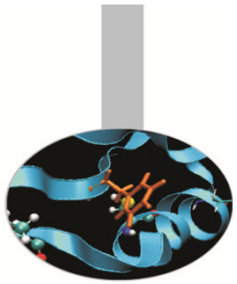
The intra-communicator NEWINTRACOMM may be generated from an inter-communicator INTERCOMM calling the function:

```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
```

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

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 merged group the processes are ordered starting from group 1; i.e. the processes in group 2 have a higher rank.

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:

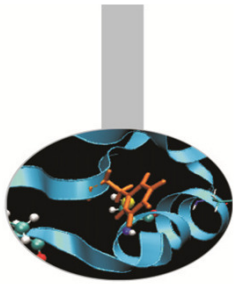
```fortran
interface
    subroutine mpi_cart_create(comm_old, ndims, ldims, periods, reorder,
                                    comm_cart, ierr)
            integer, intent(in) :: comm_old, ndims
            integer, dimension(:), intent(in) :: ldims
            logical, dimension(:), intent(in) ::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

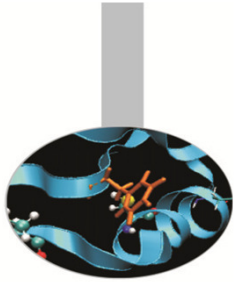


The `MPI_CART_CREATE` function returns the new communicator `COMM_CART`, connected to a grid with `NDIMS` dimensions. The extent of each dimension must be defined in `LDIMS(1:NDIMS)` and it is possible to specify periodicity for each dimension. The `REORDER` variable is used to allow reordering of the 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



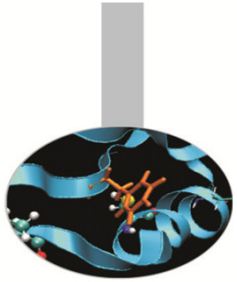
Given a communicator `COMM`, the function `MPI_TOPO_TEST` returns the associated topology:

- `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
```

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

Topologies

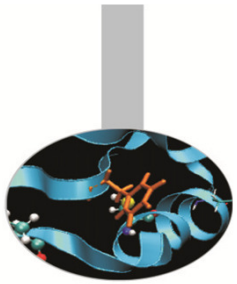


Given a communicator `COMM`, with cartesian topology, the function `MPI_CARTDIM_GET` returns the number of dimensions

```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
```

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

Topologies



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

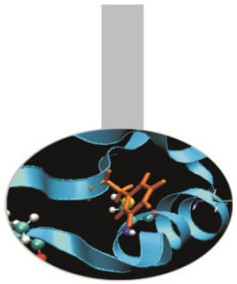
fortran

```
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
```

c/c++

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

Topologies



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

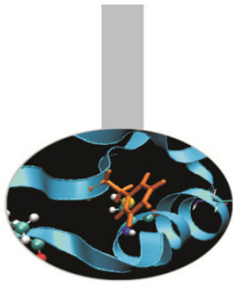
```fortran
interface
    subroutine mpi_cart_rank(comm, coords, rank, ierr)
         integer, intent(in) :: comm
         integer, dimension(:), intent(in) :: coords
         integer, intent(out) :: rank, ierr
    end subroutine mpi_cart_rank
end interface
```

fortran

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

c/c++

Topologies

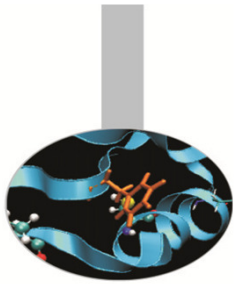


The following function returns the coordinates of a process in a cartesian topology:

```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
```

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

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 in the `DIM` dimension to a `DELTA` distance. The following function returns the ranks of the processes `SOURCE` and `DEST`…

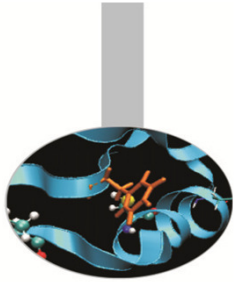
```fortran
interface
    subroutine mpi_cart_shift(comm, dim, delta, source, dest, ierr)
         integer, intent(in) :: comm, dim, delta
         integer, intent(out) :: source, dest, ierr
    end subroutine mpi_cart_shift
end interface
```

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

… that may be passed to the function

```fortran
CALL MPI SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, &
                  SENDTAG, RECVBUF,RECVCOUNT, RECVTYPE, &
                  SOURCE, RECVTAG, COMM, STATUS, IERROR)
```

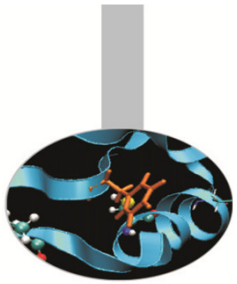
Example: MPI_CART_SHIFT



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

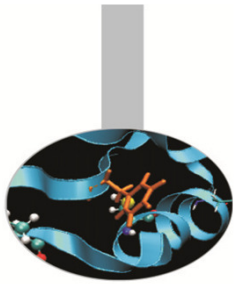
```c
int MPI_Cart_sub( MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```

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*

IBM XL compilers: mpi*xxx*_r –qsmp=omp –qnosave -O3 \
-o nomefile.exe nomefile.*xxx*

Execution on x86_64 platforms:

export OMP_NUM_THREADS=*threads*

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