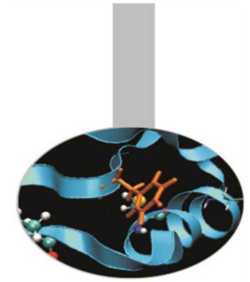
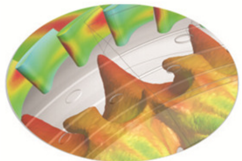
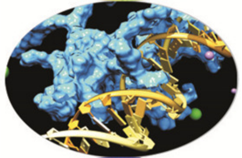
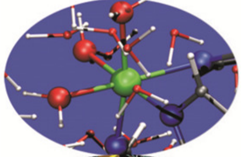
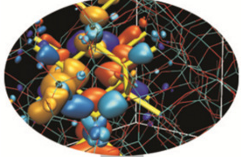
CINECA SCAI

SuperComputing Applications and Innovation



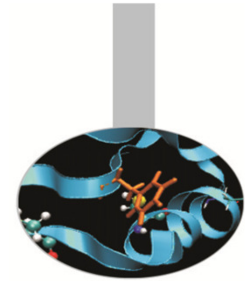
# MPI avanzato

## Introduzione al calcolo parallelo

CINECA

SCAI
SuperComputing Applications and Innovation



# Pack

Il sistema MPI permette di unire dati diversi in un unico buffer, che può essere usato per le comunicazioni. In questo modo l'applicazione può migliorare i tempi di trasferimento. Per raccogliere i dati in un unico buffer si utilizza l'istruzione `MPI_PACK`

fortran

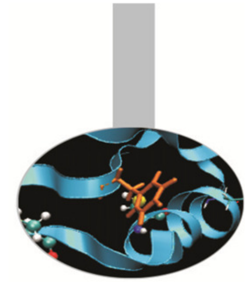
```
INTERFACE
  SUBROUTINE MPI_PACK(inbuf,incount,datatype,outbuf,outsize,position,comm,ierr)
    INTEGER, INTENT(IN) :: INCOUNT, DATATYPE, OUTSIZE, COMM
    <type>, INTENT(IN) :: INBUF(:)
    <type>, INTENT(OUT) :: OUTBUF(:)
    INTEGER, INTENT(INOUT) :: POSITION
    INTEGER, INTENT(OUT) :: IERR
  END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

dove INCOUNT elementi del tipo DATATYPE del buffer INBUF vengono copiati nel buffer OUTBUF partendo dalla posizione POSITION (in byte). In uscita POSITION assume il valore dell'indirizzo libero successivo.

CINECA



# Unpack

Viceversa la funzione `MPI_UNPACK` esegue l'operazione inversa

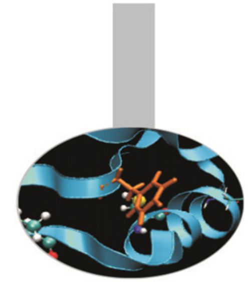
fortran

```
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
```

c/c++

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

SuperComputing Applications and Innovation



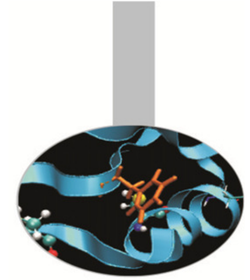
# Esercizi: pack e unpack

Esercizio 1:

modificare l'esempio get_data.f in modo da usare MPI_Pack e MPI_Unpack.

Esercizio 2:

si provi a scrivere un codice che raggruppa gli elementi di una riga di una matrice sparsa e li invia dal processo 0 al processo 1.



# Definizione di dati derivati

Il sistema MPI mette a disposizione del programmatore la possibilità di dichiarare il tipo di dati associato ad ogni comunicazione. Il sistema MPI definisce i dati primitivi seguenti:

fortran

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

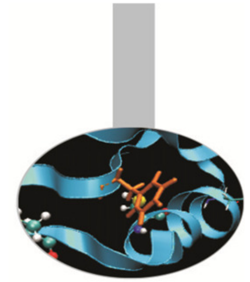
c/c++

```
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
```

Oltre a questi è possibile definire tipi personalizzati di dati, costruiti come una sequenza di tipi primitivi o tipi personalizzati definiti in precedenza:

```
Generico = [(prim_0 , pos_0), (prim_1 , pos_1), ..., (prim_n-1 , pos_n-1)]
```

CINECA SCAI
SuperComputing Applications and Innovation



# Definizione di dati derivati

La procedura di definizione avviene in due passi:

- Specifica della struttura del nuovo tipo di dato, in base ai tipi di dati definiti in precedenza.
- Registrazione ovvero "ufficializzazione" del nuovo tipo nei riguardi di MPI.

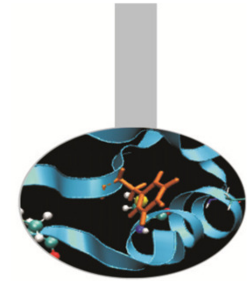
Solo dopo che il nuovo tipo è stato registrato è ammesso usarlo nello scambio di messaggi.

La subroutine di registrazione che conclude la procedura di creazione di un nuovo tipo di dato è la seguente:

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 )
```

SCAI
SuperComputing Applications and Innovation



# Vettore di dati contigui

Un vettore di dati contigui rappresenta la forma più semplice di tipo di dato derivato. Tra i dati non sono presenti spazi vuoti ovvero tra un elemento e l'altro non sono ammesse lacune.

fortran

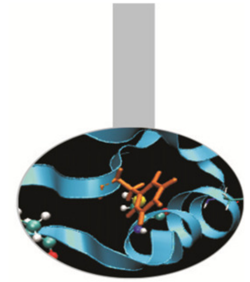
```
interface
    subroutine mpi_type_contiguous (quanti_el, tipo_el, tipo_vet, ierr)
        integer, intent(in) :: quanti_el ! quanti elem. ha il vettore
        integer, intent(in) :: tipo_el   ! tipo di ogni elemento
        integer, intent(out) :: tipo_vet ! identif. del nuovo tipo
    end subroutine mpi_type_contiguous
end interface
```

c/c++

```
int MPI_Type_contiguous ( int quanti_el, MPI_Datatype tipo_el,
                          MPI_Datatype *tipo_vet)
```

In queste istruzioni si definisce il nuovo tipo `TIPO_VET` a partire da `QUANTI_EL` ripetizioni del tipo `TIPO_EL`.

CINECA



# Vettore di dati contigui

Ad esempio, se

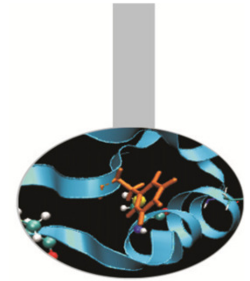
```
Tipo_el = {(double, 0), (char, 8)}
```

ed è un oggetto di estensione 16 e QUANTI_EL = 3, allora

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

Si noti che tipo_el, il secondo argomento, può coincidere con uno dei tipi predefiniti (ad es. MPI_DOUBLE_PRECISION) ma può anche essere un tipo derivato che è stato definito in precedenza

SCAI
SuperComputing Applications and Innovation



# Dati non contigui

Per inserire dati non contigui in memoria sono possibili varie soluzioni: se i vari blocchi di dati sono tutti della stessa grandezza e se lo spazio tra l'inizio di un blocco e l'inizio del blocco seguente è sempre lo stesso si deve utilizzare la subroutine `mpi_type_vector` mentre se ogni blocco contiene un numero di dati diverso e/o la distanza tra blocco e blocco è non uniforme bisogna usare la subroutine `mpi_type_indexed`.

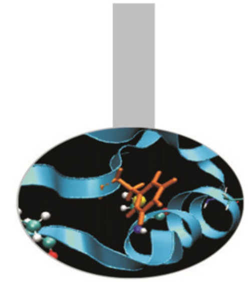
fortran

```
interface
   subroutine mpi_type_vector(num_blk,lun_blk,occ_blk,tipo_el,tipo_vet,cod_err)
       integer, intent(in) ::  num_blk  ! quanti blocchi di elementi
       integer, intent(in) ::  lun_blk  ! quanti elem. utili ha un blocco.
       integer, intent(in) ::  occ_blk  ! quanti elem., inclusi gli elem.
                                        ! saltati ha un blocco.
       integer, intent(in) ::  tipo_el  ! tipo di ogni elemento
                                        ! di ogni blocco.
       integer, intent(out) :: tipo_vet ! identif. del nuovo oggetto.
       integer, intent(out) :: cod_err  ! codice di errore
   end subroutine mpi_type_vector
end interface
```

c/c++

```
int MPI_Type_vector( int num_blk, int lun_blk, int occ_blk,
                     MPI_Datatype tipo_el, MPI_Datatype *tipo_vet )
```

SCAI
SuperComputing Applications and Innovation

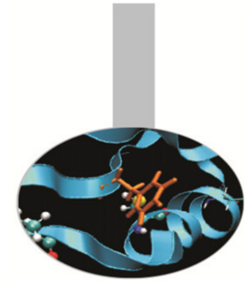


# Dati non contigui

Si noti che la lunghezza di un blocco e l'occupazione sono sempre misurate in numero di elementi. Se, ad esempio, il tipo di elemento è `MPI_INTEGER` e si è posto `lun_blk=7` e `occ_blk=10` allora ogni blocco occuperà 4 x 10 = 40 byte di cui 4 x 7 = 28 veramente usati e 4 x 3 = 12 di salto tra la fine dei dati utili del blocco e l'inizio dei dati utili del blocco successivo.

Vedere l'esempio Vettore a blocchi in Fortran o in C

CINECA



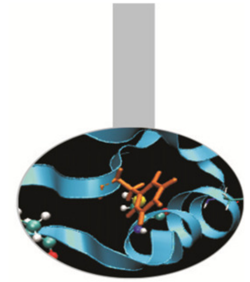
# Dati non contigui

Per i blocchi di dimensione non uniforme invece:

fortran
```
interface
 subroutine mpi_type_indexed(num_blk,v_lun_blk,v_monte,tipo_el,tipo_vet,cod_er)
   integer, intent(in) :: num_blk      ! quanti blocchi di elem.
   integer, intent(in), dimension(:) :: v_lun_blk  ! quanti elem. hanno i
                                                   ! vari blocchi
   integer, intent(in), dimension(:) :: v_monte    ! quanti elem. cisono prima
                                                   ! dell'inizio di ogni blocco
   integer, intent(in) :: tipo_el   ! tipo di ogni elemento di ogni blk.
   integer, intent(out) :: tipo_vet ! identif. del nuovo oggetto.
   integer, intent(out) :: cod_err  ! codice di errore
 end subroutine mpi_type_indexed
end interface
```

c/c++
```
int MPI_Type_indexed( int num_blk, int v_lun_blk[], int v_monte[],
                      MPI_Datatype tipo_el, MPI_Datatype *tipo_vet )
```

I due vettori usati in ingresso hanno `num_blk` dati ognuno poiché, per ogni singolo blocco, va indicato il numero di elementi di cui è costituito e il posizionamento del suo elemento iniziale ovvero quanti elementi stanno a monte del primo degli elementi del blocco.



# Dati non contigui

Se si vogliono tre blocchi, rispettivamente di 5, 13 e 7 elementi e si vuole che tra la fine di un blocco e l'inizio del successivo ci sia uno spazio di 3 blocchi i vettori v_lun_blk e v_monte dovranno essere così inizializzati:

```
v_lun_blk = (/ 5, 13,  7 /)
v_monte   = (/ 0,  8, 24 /)
```

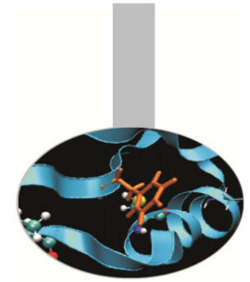
La funzione seguente ritorna la dimensione `DIM` del tipo di dati `DATATYPE`.

fortran

```
interface
    subroutine mpi_type_extent (datatype, dim, cod_err)
        integer, intent(in) :: datatype   ! Tipo primitivo o derivato
        integer, intent(out) :: dim       ! Occupazione in byte
        integer, intent(out) :: cod_err  ! codice di errore
    end subroutine mpi_type_extent
end interface
```

c/c++

```
int MPI_Type_extent( MPI_Datatype datatype, MPI_Aint *dim )
```



# Altre funzioni sui dati

La funzione `mpi_type_hvector` è simile alla `mpi_type_vector`, ma lo stride è misurato in byte. La funzione `mpi_type_hindexed` è simile alla `mpi_type_indexed`, ma la posizione è misurata in byte.

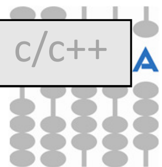
Il sistema MPI mette a disposizione la funzione `mpi_address` che costituisce un modo portabile per scoprire l'indirizzo di un dato.

fortran

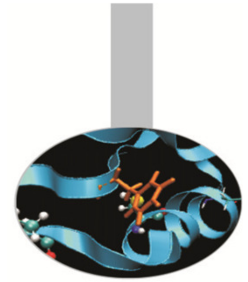
```
interface
    subroutine mpi_address (dato, address, ierr)
        integer, intent(in) :: dato
        integer, intent(out) :: address, ierr
    end subroutine mpi_address
end interface
```

c/c++

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



# Spedire diversi dati in un messaggio



La creazione di un tipo di dato costituito da dati di diverso tipo è l'operazione più generale e richiede un numero elevato di informazioni ovvero tre vettori di interi:

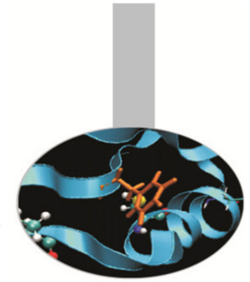
il primo dedicato alla specifica della dimensione del blocco;

il secondo per indicare il numero di byte a monte del primo elemento di ogni blocco;

il terzo per precisare il tipo di dato di ogni elemento di uno stesso blocco.

Si noti che l'unità di misura per indicare l'inizio del blocco non è più l'elemento perché non esiste un unico tipo di elemento ma ogni blocco può essere fatto di un suo particolare tipo. Pertanto l'unità di misura è il byte e occorre quindi conoscere la dimensione in byte di ogni tipo di elemento usato per definire il nuovo tipo.

# Spedire diversi dati in un messaggio

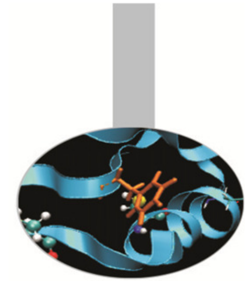


La subroutine mpi_type_struct ha la seguente interfaccia:

fortran
```
interface
  subroutine mpi_type_struct(num_blk,v_lun_blk,v_monte,v_tipo,tipo_vet,cod_err)
    integer, intent(in) :: num_blk      ! quanti blocchi di elem.
    integer,intent(in),dimension(:) :: v_lun_blk ! quanti elem. hanno i blocchi
    integer, intent(in), dimension(:) :: v_monte ! quanti byte ci sono prima di
                                                 ! ogni blocco
    integer, intent(in), dimension(:) :: v_tipo  ! tipo di ogni elemento per
                                                 ! ogni blocco
    integer, intent(out) :: tipo_vet    ! identif. del nuovo oggetto.
    integer, intent(out) :: cod_err     ! codice di errore
  end subroutine mpi_type_struct
end interface
```

c/c++
```
int MPI_Type_struct( int num_blk, int v_lun_blk[], MPI_Aint v_monte[],
                          MPI_Datatype v_tipo[], MPI_Datatype *tipo_vet )
```

SCAI SuperComputing Applications and Innovation



# Vettore di dati contigui

Ad esempio supponiamo di volere una struttura costituita da due MPI_LOGICAL (ipotizzando che un MPI_LOGICAL richieda 4 byte), da tre MPI_DOUBLE_PRECISION (ognuno di 8 byte) e da nove dati MPI_CHARACTER. Si suppone inoltre che tra un blocco e l'altro ci debba essere un intervallo vuoto di 10 byte. Allora i tre vettori, di tre elementi, avranno i seguenti valori:

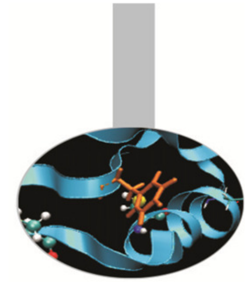
```
v_lun_blk = (/ 2, 3, 9 /)
v_monte   = (/ 0, 18, 52 /)
v_tipo    = (/ MPI_LOGICAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER /)
```

In F90 può risultare molto naturale definire un tipo derivato che combaci esattamente o quasi con l'oggetto rappresentato dal nuovo tipo MPI.

L'unico accorgimento da seguire è quello di usare l'istruzione SEQUENCE per imporre che l'ordine usato dal programmatore per specificare i componenti del tipo derivato F90 sia esattamente rispettato dal compilatore (che, in assenza di SEQUENCE può ottimizzare e quindi cambiare liberamente tale ordine).

CINECA

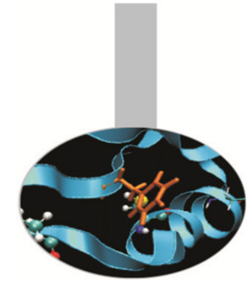
SuperComputing Applications and Innovation



# Esercizio

Esercizio:

modificare get_data in modo da utilizzare una struttura di dati MPI.



# Processi e gruppi di processi

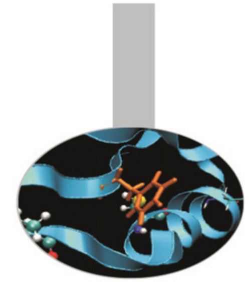
Nel linguaggio MPI il processo è l'unità fondamentale di calcolo. Ogni processo è indipendente dagli altri ed ha uno spazio di memoria autonomo. I processi MPI vengono eseguiti secondo il modello MIMD, tuttavia non esiste un meccanismo per assegnare processi ai singoli processori, nè per generare e terminare i processi.

Ogni processo MPI appartiene ad un gruppo ed ha un numero identificativo o rank (da 0 ad N-1), relativo al gruppo di appartenenza. I gruppi di processi possono essere generati e distrutti, ma finchè esistono sono statici. Ogni gruppo ha un identificativo (handle) ed è un oggetto opaco, ovvero il programmatore non ha accesso alla sua struttura interna. Per conoscere gli attributi del gruppo si devono utilizzare apposite funzioni, ad es.:

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

All'inizio tutti i processi appartengono al gruppo predefinito dal comunicatore `MPI_COMM_WORLD`: tutti gli altri gruppi devono essere generati esplicitamente. I singoli processi possono far parte di diversi gruppi.

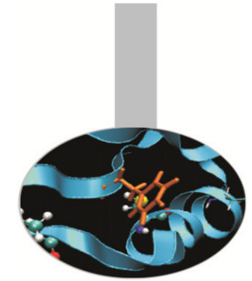
# Costruire i gruppi di processi



La funzione seguente permette di costruire il nuovo gruppo NEWGROUP partendo dal gruppo GROUP; il processo con indice RANKS(I) in GROUP assume indice I in NEWGROUP :

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)
```



# Costruire i gruppi di processi

Per esempio, se GROUP è un gruppo di 8 processi (numerati da 0 a 7) e RANKS(1:3)=(1,5,2) è il vettore che identifica i 3 processi da includere nel nuovo gruppo, l'istruzione

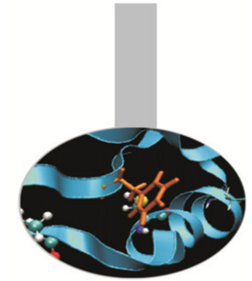
```fortran
call mpi_group_incl (group, 3, ranks, newgroup, ierr)
```

genera il gruppo NEWGROUP costituito dai 3 processi indicati sopra.

La corrispondenza tra gli indici nei 2 gruppi è la seguente:

| Group | Newgroup |
|---|---|
| 1 | 0 |
| 5 | 1 |
| 2 | 2 |

CINECA SCAI
SuperComputing Applications and Innovation



# Costruire i gruppi di processi

Nella funzione seguente viceversa `RANKS(I)` indica i processi di `GROUP` da eliminare per costruire `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
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

E' anche possibile aggiungere e togliere i processi specificando un'estensione di indici; negli esempi seguenti `RANGES(1:N,1:3)` è una matrice i cui elementi nella seconda dimensione specificano il primo e l'ultimo indice da considerare, ed il passo:

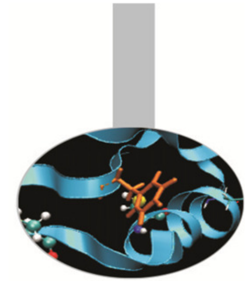
```fortran
call mpi_group_range_incl (group, n, ranges, newgroup, ierr)
```

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

CINECA

SCAI
SuperComputing Applications and Innovation

# Lavorare con i gruppi di processi



Le operazioni compiute sui gruppi sono locali, ovvero non coinvolgono comunicazioni.

L'istruzione seguente permette di conoscere quale rango, relativamente al gruppo `GROUP2`, hanno i processi di rango `RANKS1(:)` relativamente al gruppo `GROUP1`:

fortran

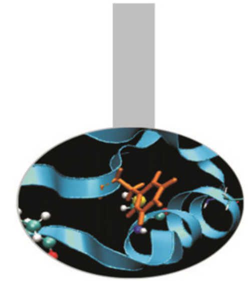
```
interface
    subroutine mpi_group_translate(group1, n, ranks1, group2, ranks2, ierr)
        integer, intent(in) :: group1, n, ranks1(:), group2
        integer, intent(out) :: ranks2(:), ierr
    end subroutine mpi_group_translate
end interface
```

c/c++

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

CINECA

CINECA SCAI
SuperComputing Applications and Innovation



# Lavorare con i gruppi di processi

E’ possibile controllare l’uguaglianza di due gruppi:

```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
```

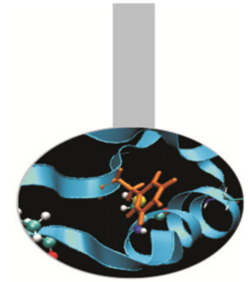
```c
int MPI_Group_compare (group1, group2, result, ierr)
```

I valori ritornati in `RESULT` sono 3:

- `MPI_IDENT` se i gruppi hanno gli stessi processi e gli stessi indici
- `MPI_SIMILAR` se i gruppi hanno gli stessi processi ma numerati differentemente
- `MPI_UNEQUAL` se i processi dei 2 gruppi sono diversi.

CINECA

CINECA SCAI
SuperComputing Applications and Innovation



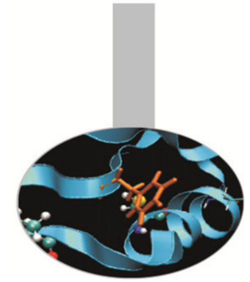
# Comunicazioni

Un messaggio MPI è identificato da un'etichetta, costituita da un contesto ed un indice relativo al contesto. Come i gruppi sono usati per partizionare i processi, così i contesti sono usati per partizionare i messaggi. Tuttavia i contesti non sono visibili a livello applicativo.

Un comunicatore definisce l'ambito di operabilità di un'operazione di comunicazione. I comunicatori definiscono ambienti di comunicazione indipendenti, che comprendono gruppi di processi e contesti comunicativi. Un comunicatore è un oggetto "opaco", cui si fa riferimento con un identificativo o "handle".

CINECA

# Comunicazioni

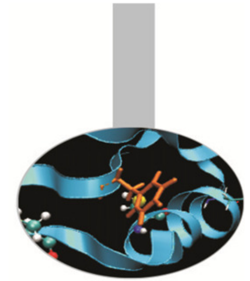


Esiste sempre un comunicatore di default, ma può essere utile utilizzare comunicatori diversi per poter sfruttare il potenziale del sistema MPI. In caso contrario l’applicazione deve provvedere a gestire le comunicazioni tra i singoli processi, con perdita di prestazione e maggior probabilità di errori.

In fasi diverse di un’applicazione possono essere usati contesti diversi per evitare confusioni tra i messaggi.

I comunicatori (i loro identificativi) devono essere passati alle funzioni di comunicazione.

CINECA SCAI
SuperComputing Applications and Innovation



# Costruire i comunicatori

L'istruzione seguente permette di generare un nuovo comunicatore:

```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

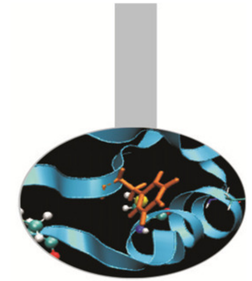
```c
int MPI_Comm_create ( MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm )
```

c/c++

E' importante ricordare che:

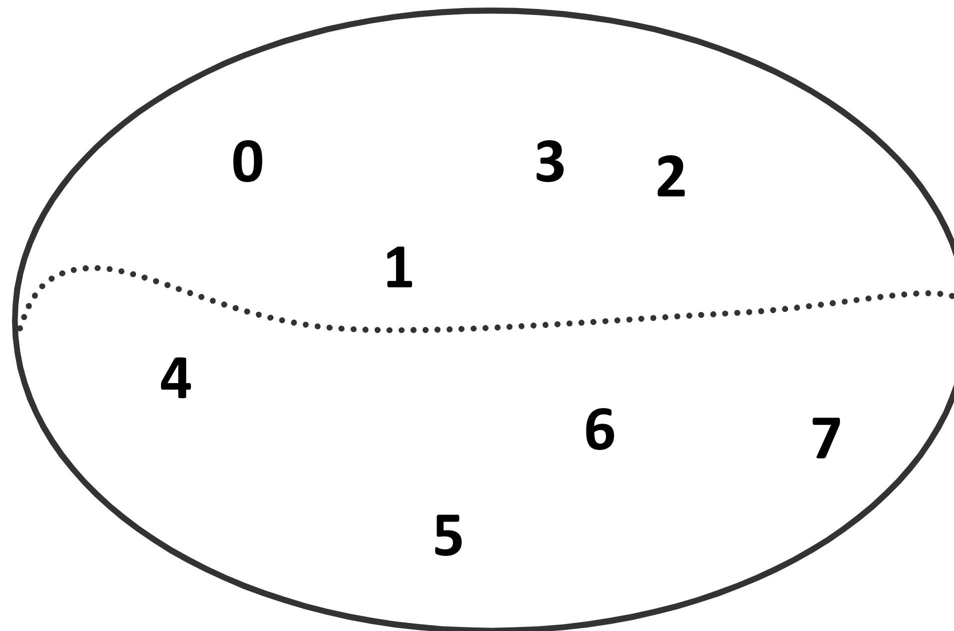
- `COMM` è un comunicatore pre-esistente, associato ad un gruppo di processi
- L'istruzione dev'essere eseguita da tutti i processi del gruppo associato al comunicatore `COMM`
- `GROUP` è un sottogruppo del gruppo di processi associato a `COMM`
- `NEWCOMM` è il comunicatore generato

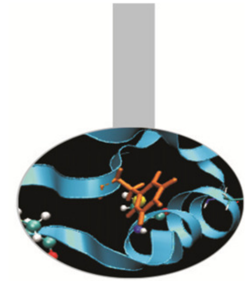
CINECA



# Dividere i comunicatori

E’ possibile dividere un comunicatore in più parti; per esempio, si supponga di avere un comunicatore associato ad un gruppo di 8 processi, da dividere in 2 parti cosiffatte:





# Dividere i comunicatori

Questo è il codice che ogni processo del gruppo deve eseguire: ogni processo riceve il proprio comunicatore `NEWCOMM`.

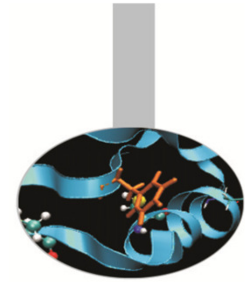
```
call mpi_comm_rank (comm, rank, ierr)
call mpi_comm_size (comm, size, ierr)
color = 2*rank/size
key   = size - rank - 1
call mpi_comm_split (comm, color, key, newcomm, ierr)
```

Gli indici che vengono associati ai processi risultanti sono i seguenti:

| 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 |

Se `COLOR=MPI_UNDEFINED`, la funzione `MPI_COMM_SPLIT` ritorna `NEWCOMM=MPI_COMM_NULL`

SuperComputing Applications and Innovation



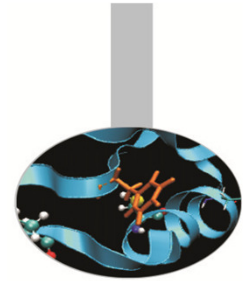
# Esercizio: i comunicatori

Esercizio:

si provi a generare un nuovo comunicatore di Q processi partendo da un gruppo di Q^2 processi.

Una possibile risposta è l'esempio comm_create.

SuperComputing Applications and Innovation



# Comunicazioni tra gruppi di processi

MPI permette di far comunicare gruppi di processi che non si intersecano, ovvero non hanno processi in comune; è possibile così far dialogare porzioni di programma distinte o realizzare sistemi client-server.

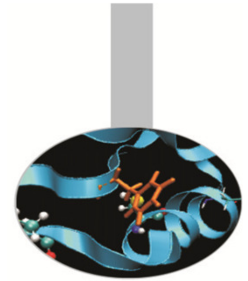
Le comunicazioni tra gruppi distinti di processi possono avvenire solo tra processi singoli: non sono realizzabili comunicazioni collettive.

Il processo mittente deve specificare l'indice (relativo all'altro gruppo) del destinatario; viceversa il processo ricevente deve specificare l'indice (sempre relativo all'altro gruppo) del mittente.

Le funzioni `mpi_comm_size`, `mpi_comm_rank`, `mpi_comm_group` ritornano informazioni sul gruppo locale.

Le funzioni `mpi_comm_remote_size`, `mpi_comm_remote_group` ritornano informazioni sui gruppi remoti.

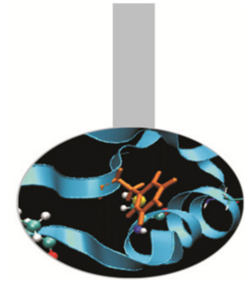
# Comunicazioni tra gruppi di processi



Per generare un comunicatore tra gruppi diversi, o inter-comunicatore, si richiede:

- un processo "leader" per ognuno dei 2 gruppi
- un intra-comunicatore per le comunicazioni tra i 2 processi leader
- un’etichetta o tag, per comunicazioni sicure tra i 2 processi leader

SCAI SuperComputing Applications and Innovation



# Intercomunicatori

L'istruzione seguente genera un inter-comunicatore `NEWINTERCOMM` tra i processi `LOCALLEADER` e `REMOTELEADER` dell'intra-comunicatore `LOCALCOMM`, utilizzando l'etichetta TAG e il comunicatore punto-punto `PEERCOMM` (al quale i 2 leader sono già associati). Si deve notare che `REMOTELEADER` e `PEERCOMM` sono riferiti al processo locale, mentre TAG deve avere lo stesso valore per entrambi i processi, locale e remoto:

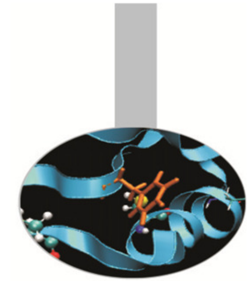
fortran

```
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 )
```

SuperComputing Applications and Innovation



# Intercomunicatori

Si può generare un intra-comunicatore `NEWINTRACOMM` partendo da un inter-comunicatore `INTERCOMM` con l'istruzione

fortran

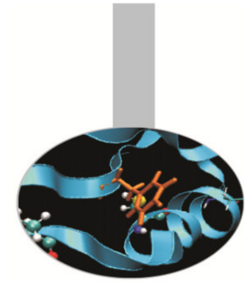
```
interface
    subroutine mpi_intercomm_merge(intercomm, high, newintracomm, ierr)
            integer, intent(in) :: intercomm, high
            integer, intent(out) :: newintracomm, ierr
     end subroutine mpi_intercomm_merge
end interface
```

c/c++

```
int MPI_Intercomm_merge(MPI_Comm intercomm,int high,MPI_Comm *newintracomm)
```

che permette quindi di unire 2 gruppi separati. Il valore di HIGH dev'essere lo stesso per tutti i processi dello stesso gruppo. Se HIGH =.FALSE. per il gruppo 1 e HIGH =.TRUE. per il gruppo 2, nel nuovo intra-comunicatore i processi vengono ordinati partendo dal gruppo 1, ovvero quelli del gruppo 2 hanno indice più "alto".

SuperComputing Applications and Innovation



# Topologie

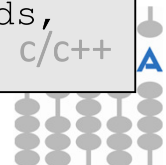
Per molte applicazione è utile che i processi siano ordinati secondo una topologia specifica. MPI permette di definire topologie mediante un grafo nel quale i processi sono collegati con un arco. Esiste anche un supporto esplicito per topologie a griglia cartesiana, definibile con la funzione:

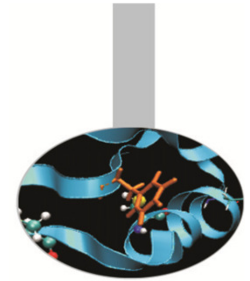
fortran

```
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
```

c/c++

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





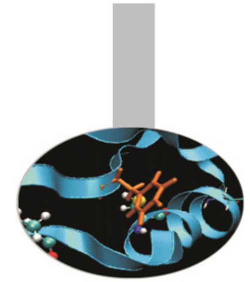
# Topologie

La funzione `MPI_CART_CREATE` ritorna il nuovo comunicatore `COMM_CART`, associato alla griglia a `NDIMS` dimensioni. E’ necessario indicare in `LDIMS(1:NDIMS)` la lunghezza di ogni dimensione ed è possibile dare la periodicità in ogni singola dimensione. La variabile `REORDER` serve ad indicare se l’indice dei processi dev’essere cambiato.

Nelle topologie cartesiane i processi sono ordinati per righe.

Sono disponibili funzioni che ritornano la topologia associata ad un comunicatore.

CINECA SCAI
SuperComputing Applications and Innovation



# MPI_TOPO_TEST

Dato il comunicatore COMM, la funzione MPI_TOPO_TEST ritorna la topologia associata:

- MPI_GRAPH: grafo
- MPI_CART: cartesiana
- MPI_UNDEFINED: nessuna topologia

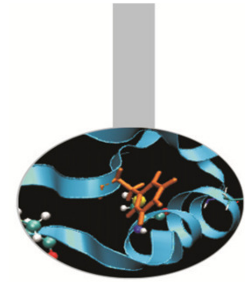
fortran
```
interface
    subroutine mpi_topo_test(comm, topol, ierr)
            integer, intent(in) :: comm
            integer, intent(out) :: topol, ierr
    end subroutine mpi_topo_test
end interface
```

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

CINECA

SuperComputing Applications and Innovation

# MPI_CARTDIM_GET



Dato il comunicatore `COMM`, con topologia cartesiana, la funzione `MPI_CARTDIM_GET` ritorna il numero di dimensioni.

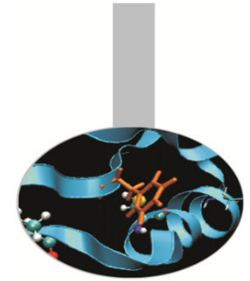
```fortran
interface
    subroutine mpi_cartdim_get(comm, ndims, ierr)
         integer, intent(in) :: comm
         integer, intent(out) :: ndims, ierr
    end subroutine mpi_cartdim_get
end interface
```

fortran

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

c/c++

SCAI CINECA SuperComputing Applications and Innovation



# MPI_CART_GET

La funzione `MPI_CART_GET`, più generale, ritorna il numero `DIMS(:)` di processi per ogni dimensione, la periodicità per ogni dimensione, le coordinate del processo.

fortran

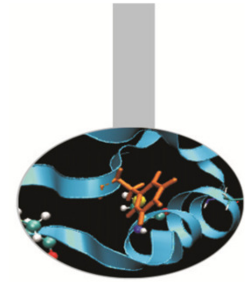
```
interface
    subroutine mpi_cart_get(comm, maxdims, dims, periods, coords, ierr)
        integer, intent(in) :: comm, maxdims
        integer, intent(out) :: ierr
        integer, dimension(:), intent(out) :: dims, coords
        logical, dimension(:), intent(out) :: periods
    end subroutine mpi_cart_get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

c/c++

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

CINECA

SCAI
SuperComputing Applications and Innovation



# MPI_CART_RANK

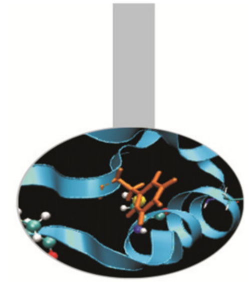
Dati un comunicatore con topologia cartesiana e le coordinate del processo, la funzione `MPI_CART_RANK` ritorna l'indice associato al processo.

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
```

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

CINECA

SCAI
SuperComputing Applications and Innovation



# MPI_CART_COORDS

La funzione `MPI_CART_COORDS` ritorna le coordinate in `COORDS(1:MAXDIMS)`.

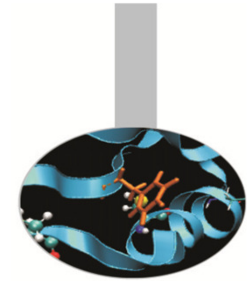
fortran

```
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)
```

CINECA



# MPI_CART_SHIFT

Le topologie possono essere utili a mappare i processi sui processori o realizzare comunicazioni lungo direzioni specifiche. Si supponga che, in una topologia di tipo cartesiano, ogni processo debba inviare dati a distanza `DELTA` lungo la dimensione DIM. L'istruzione

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++

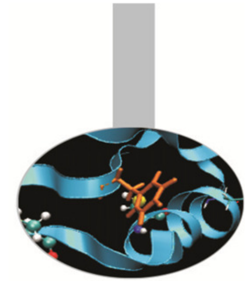
```
int MPI_Cart_shift(MPI_Comm comm,int dim,int delta,int *source,int *dest)
```

ritorna gli indici dei processi `SOURCE` e `DEST` da passare all'istruzione

fortran

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

SCAI
SuperComputing Applications and Innovation



# Esempio: MPI_CART_SHIFT

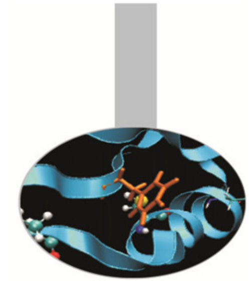
fortran

```
    ....
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)
```

CINECA

SCAI
SuperComputing Applications and Innovation

# MPI_CART_SUB



L'istruzione `MPI_CART_SUB` permette di generare una nuova topologia cartesiana, "ritagliando" un certo numero di dimensioni REMAIN_DIMS da una topologia più ampia, associata al comunicatore COMM:

```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
```

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

Ad esempio, se COMM è associato ad una topologia 2 x 3 x 4 e `REMAIN_DIMS=(.T., .T., .F.)`, vengono generate 4 nuove topologie di dimensione 2 x 3. Chiaramente per ogni singolo processo 1 solo `NEWCOMM` viene ritornato, perché ogni processo appartiene ad una sola nuova topologia.

CINECA