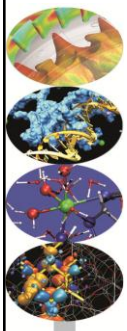
CINECA SCAI
SuperComputing Applications and Innovation

Calcolo Parallelo con MPI

(2ª parte)



Claudia Truini
c.truini@cineca.it

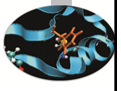
Luca Ferraro
l.ferraro@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Calcolo parallelo con MPI (2ª parte)



- **Approfondimento sulle comunicazioni *point-to-point***
- **La comunicazione non *blocking***
- **Laboratorio n° 3**
- **MPI virtual topologies**
- **Laboratorio n° 4**

CINECA

CINECA SCAI SuperComputing Applications and Innovation

Modalità di comunicazione

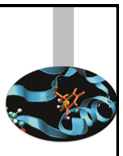
Una comunicazione tipo *point-to-point* può essere:

- **Blocking**:
  - il controllo è restituito al processo che ha invocato la primitiva di comunicazione solo quando la stessa è stata completata
- **Non blocking**:
  - il controllo è restituito al processo che ha invocato la primitiva di comunicazione quando la stessa è stata eseguita
  - il controllo sull'effettivo **completamento della comunicazione** deve essere fatto in seguito
  - nel frattempo il processo può eseguire altre operazioni

CINECA

CINECA SCAI SuperComputing Applications and Innovation

Criteri di completamento della comunicazione



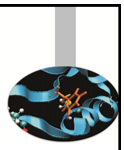
- Dal punto di vista non locale, ovvero di entrambi i processi coinvolti nella comunicazione, *è rilevante il criterio in base al quale si considera completata la comunicazione*

Funzionalità	Criterio di completamento
Syncronous send	è completa quando è terminata la ricezione del messaggio
Buffered send	è completa quando è terminata la scrittura dei dati da comunicare sul buffer predisposto (non dipende dal *receiver*!)
Standard send	Può essere implementata come una *syncronous* o una *buffered send*
Receive	è completa quando il messaggio è arrivato

- Nella libreria MPI esistono diverse primitive di comunicazione *point-to-point*, che combinano le modalità di comunicazione ed i criteri di completamento

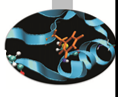
CINECA

Syncronous SEND blocking: MPI_Ssend



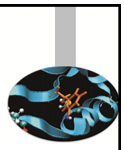
- Per il completamento dell'operazione il *sender* deve essere informato dal *receiver* che il messaggio è stato ricevuto
- Gli argomenti della funzione sono gli stessi della **MPI_Send**
- Pro: è la modalità di comunicazione *point-to-point* più semplice ed affidabile
- Contro: può comportare rilevanti intervalli di tempo in cui i processi coinvolti non "*hanno nulla di utile da fare*", se non attendere che la comunicazione sia terminata

Syncronous Send: un esempio



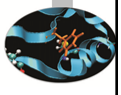
```
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

int main(int argc, char *argv[]) {

    MPI_Status status;
    int rank, size;

    int i;
    /* data to communicate */
    double  matrix[MSIZE];

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
                matrix[i] = (double) i;
        MPI_Ssend(matrix, MSIZE, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);

    } else if (rank == 1) {
        MPI_Recv(matrix, MSIZE, MPI_DOUBLE, 0, 666, MPI_COMM_WORLD, &status);
        printf("Process 1 receives an array of size %d from process 0.\n", MSIZE);
    }

    /* Quit MPI */
    MPI_Finalize();

    return 0;
}
```

Buffered SEND blocking: MPI_Bsend



- Una ***buffered send*** è completata immediatamente, non appena il processo ha copiato il messaggio su un opportuno *buffer* di trasmissione
- Il programmatore non può assumere la presenza di un *buffer* di sistema allocato per eseguire l'operazione, ma deve effettuare un'operazione di
  - **BUFFER_ATTACH** per definire un'area di memoria di dimensioni opportune come buffer per il trasferimento di messaggi
  - **BUFFER_DETACH** per rilasciare le aree di memoria di *buffer* utilizzate
- Pro
  - ritorno immediato dalla primitiva di comunicazione
- Contro
  - È necessaria la gestione esplicita del *buffer*
  - Implica un'operazione di copia in memoria dei dati da trasmettere

Gestione dei buffer: MPI_Buffer_attach



In C

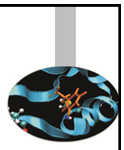
```
MPI_Buffer_attach(void *buf, int size)
```

In Fortran

```
MPI_BUFFER_ATTACH(BUF, SIZE, ERR)
```

- Consente al processo *sender* di allocare il *send* buffer per una successiva chiamata di **MPI_Bsend**
- Argomenti:
  - **buf** è l'indirizzo iniziale del buffer da allocare
  - **size** è un **int** (**INTEGER**) e contiene la dimensione in byte del buffer da allocare

Gestione dei buffer: MPI_Buffer_detach



In C

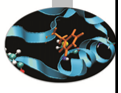
```
MPI_Buffer_detach(void *buf, int *size)
```

In Fortran

```
MPI_BUFFER_DETACH(BUF, SIZE, ERR)
```

- Consente di rilasciare il buffer creato con **MPI_Buffer_attach**
- Argomenti:
  - **buf** è l'indirizzo iniziale del buffer da deallocare
  - **size** è un **int*** (INTEGER) e contiene la dimensione in byte del buffer deallocato

Buffered Send: un esempio



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

int main(int argc, char *argv[]) {

    MPI_Status status;
    int rank, size, i;
    double *mpibuffer;
    int     mpibuffer_length;

    double  matrix[MSIZE];

    /* Start up MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
                matrix[i] = (double) i;

        mpibuffer_length = (MSIZE*sizeof(double) + MPI_BSEND_OVERHEAD);
        mpibuffer = (double *) malloc (mpibuffer_length);

        MPI_Buffer_attach(mpibuffer, mpibuffer_length);
        MPI_Bsend(matrix, MSIZE, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
    } else  if (rank == 1) {
        MPI_Recv(matrix, MSIZE, MPI_DOUBLE, 0, 666, MPI_COMM_WORLD, &status);
    }

    /* Quit MPI environment */
    MPI_Finalize();
    return 0;
}
```

Buffered Send: un esempio (FORTRAN)

```
program main
      implicit none
      include 'mpif.h'
      integer ierr, rank, nprocs
      integer i, status(MPI_STATUS_SIZE)
      integer, parameter :: MSIZE=10
      double precision matrix(MSIZE)

      INTEGER mpibuffer_length, typesize
      REAL(kind=8), DIMENSION(:), ALLOCATABLE :: mpibuffer

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

      if (rank .eq. 0) then
         do i=1,MSIZE
               matrix(i)= dble(i)
         enddo
         mpibuffer_length = msize + MPI_BSEND_OVERHEAD
         ALLOCATE(mpibuffer(mpibuffer_length))
         CALL MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, typesize, ierr)
         CALL MPI_BUFFER_ATTACH(mpibuffer, typesize*mpibuffer_length, ierr)

         CALL MPI_BSEND(matrix, MSIZE, MPI_DOUBLE_PRECISION, 1, 666, &
                        MPI_COMM_WORLD,ierr)
      else if (rank .eq. 1) then
         CALL MPI_RECV(matrix, MSIZE, MPI_DOUBLE_PRECISION, 0, 666, &
                       MPI_COMM_WORLD, status, ierr)
      endif
      call MPI_FINALIZE(ierr)
end program main
```

11

Calcolo parallelo con MPI (2ª parte)

- Approfondimento sulle comunicazioni *point-to-point*
- **La comunicazione non *blocking***
- Laboratorio n° 3
- MPI virtual topologies
- Laboratorio n° 4

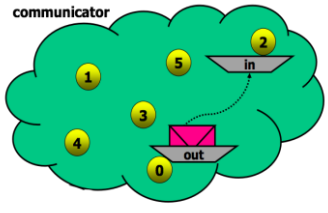
C. Truini, L. Ferraro, V.Ruggiero: MPI avanzato 12

Comunicazioni non-blocking

- Una comunicazione *non-blocking* è tipicamente costituita da tre fasi successive:
  1. L'inizio della operazione di *send* o *receive* del messaggio
  2. Lo svolgimento di un'attività che non implichi l'accesso ai dati coinvolti nella operazione di comunicazione avviata
  3. Controllo/attesa del completamento della comunicazione
- Pro
  - *Performance*: una comunicazione *non-blocking* consente di:
    - sovrapporre fasi di comunicazioni con fasi di calcolo
    - ridurre gli effetti della latenza di comunicazione
  - Le comunicazioni *non-blocking* evitano situazioni di *deadlock*
- Contro
  - La programmazione di uno scambio di messaggi con funzioni di comunicazione *non-blocking* è (leggermente) più complicata

SEND non-blocking: MPI_Isend

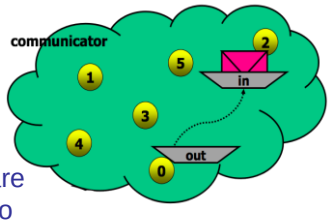
- Dopo che la spedizione è stata avviata il controllo torna al processo *sender*



- Prima di riutilizzare le aree di memoria coinvolte nella comunicazione, il processo *sender* deve controllare che l'operazione sia stata completata, attraverso opportune funzioni della libreria MPI
- Anche per la *send non-blocking, sono previste* le diverse modalità di completamento della comunicazione

RECEIVE non-blocking : MPI_Irecv

- Dopo che la fase di ricezione è stata avviata il controllo torna al processo *receiver*
- Prima di utilizzare in sicurezza i dati ricevuti, il processo *receiver* deve verificare che la ricezione sia completata, attraverso opportune funzioni della libreria MPI
- Una *receive non-blocking* può essere utilizzata per ricevere messaggi inviati sia in modalità *blocking* che non *blocking*



Binding di MPI_Isend e MPI_Irecv

In C

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

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

In Fortran

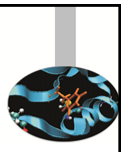
```
MPI_ISEND(buf, count, dtype, dest, tag, comm, req, err)
```

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

Le funzione `MPI_Isend` e `MPI_Irecv` prevedono un argomento aggiuntivo rispetto alle `MPI_Send` e `MPI_Recv`:

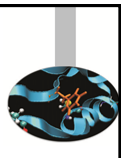
- un handler di tipo MPI_Request (INTEGER) è necessario per referenziare l'operazione di *send/receive* nella fase di controllo del completamento della stessa

Comunicazioni non blocking: completamento



- Quando si usano comunicazioni *point-to-point* non blocking è essenziale assicurarsi che la fase di comunicazione sia completata per:
  - utilizzare i dati del buffer (dal punto di vista del *receiver*)
  - riutilizzare l'area di memoria coinvolta nella comunicazione (dal punto di vista del *sender*)
- La libreria MPI mette a disposizione dell'utente due tipi di funzionalità per il test del completamento di una comunicazione:
  - tipo *WAIT* : consente di fermare l'esecuzione del processo fino a quando la comunicazione in argomento non sia completata
  - tipo *TEST*: ritorna al processo chiamante un valore *TRUE* se la comunicazione in argomento è stata completata, *FALSE* altrimenti

Binding di `MPI_Wait`



In C

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

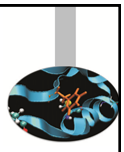
In Fortran

```
MPI_WAIT (REQUEST, STATUS, ERR)
```

Argomenti:

- [IN] **`request`** è l'*handler* necessario per referenziare la comunicazione di cui attendere il completamento (`INTEGER`)
- [OUT] **`status`** conterrà lo stato (*envelope*) del messaggio di cui si attende il completamento (`INTEGER`)

Binding di `MPI_Test`



In C

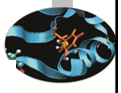
```
int MPI_Test(MPI_Request *request,
             int *flag, MPI_Status *status)
```

In Fortran

```
MPI_TEST (REQUEST, FLAG, STATUS, ERR)
```

- Argomenti
  - [IN] **`request`** è l'*handler* necessario per referenziare la comunicazione di cui controllare il completamento (`INTEGER`)
  - [OUT] **`flag`** conterrà **`TRUE`** se la comunicazione è stata completata, **`FALSE`** altrimenti (`LOGICAL`)
  - [OUT] **`status`** conterrà lo stato (*envelope*) del messaggio di cui si attende il completamento (`INTEGER(*)`)

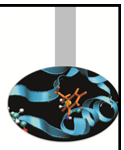
La struttura di un codice con *send non blocking*



```
/* ... Only a portion of the code */

MPI_Status status;
MPI_Request request;

int flag = 0;
double buffer[BIG_SIZE];

/* Send some data */
MPI_Isend(buffer, BIG_SIZE, MPI_DOUBLE, dest, tag,
          MPI_COMM_WORLD, &request);

/* While the send is progressing, do some useful work */
while (!flag && have_more_work_to_do) {

        /* ...do some work... */

        MPI_Test(&request, &flag, &status);
}

/* If we finished work but the send is still pending, wait */
if (!flag)
        MPI_Wait(&request, &status);
/* ... */
```

CINECA SCAI SuperComputing Applications and Innovation

Calcolo parallelo con MPI (2ª parte)

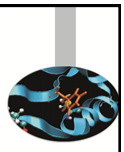


- Approfondimento sulle comunicazioni *point-to-point*
- La comunicazione non *blocking*
- **Laboratorio n° 3**
- MPI virtual topologies
- Laboratorio n° 4

CINECA

CINECA SCAI SuperComputing Applications and Innovation

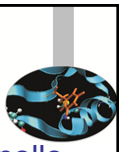
Programma della 3° sessione di laboratorio



- Modalità di completamento della comunicazione
  - Uso della syncronous send, della *buffered send* e delle funzioni di gestione dei buffer (Esercizio 12)
- Uso delle funzioni di comunicazione non-blocking
  - *Non blocking circular shift* (Esercizio 13)
  - *Array Smoothing* (Esercizio 14)

CINECA

Funzioni per il timing: MPI_Wtime e MPI_Wtick



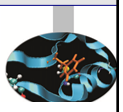
- È importante conoscere il tempo impiegato dal codice nelle singole parti per poter analizzare le performance
- `MPI_Wtime`: fornisce il numero floating-point di secondi intercorso tra due sue chiamate successive

```
double starttime, endtime;
starttime = MPI_Wtime();
.... stuff to be timed ...
endtime = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

- `MPI_Wtick`: ritorna la precisione di MPI_Wtime, cioè ritorna 10^{-3} se il contatore è incrementato ogni millesimo di secondo.
- NOTA: anche in FORTRAN sono funzioni

Esercizio 12

Syncronous Send blocking

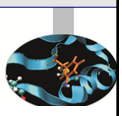


- Modificare il codice dell'Esercizio 2, utilizzando la funzione `MPI_Ssend` per la spedizione di un *array* di *float*
- Misurare il tempo impiegato nella `MPI_Ssend` usando la funzione `MPI_Wtime`

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

int main(int argc, char *argv[]) {

    MPI_Status status;
    int rank, size;
    int i, j;

    /* data to communicate */
    float  matrix[MSIZE];

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
           matrix[i] = (float)i;
        MPI_Send(matrix, MSIZE, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    } else if (rank ==1) {
        MPI_Recv(matrix, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
        printf("\nProcess 1 receives the following array from process 0.\n");
        for (i = 0; i < MSIZE; i++)
           printf("%6.2f\n", matrix[i]);
    }

    /* Quit MPI */
    MPI_Finalize();
    return 0;
}
```

Esercizio 12

Send buffered blocking

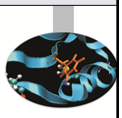


- Modificare l'Esercizio 2, utilizzando la funzione **`MPI_Bsend`** per spedire un *array* di *float*
- N.B.: la **`MPI_Bsend`** prevede la gestione diretta del *buffer* di comunicazione da parte del programmatore
- Misurare il tempo impiegato nella **`MPI_Bsend`** usando la funzione **`MPI_Wtime`**

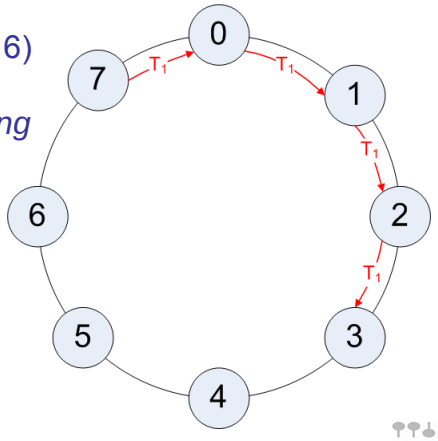
```
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

int main(int argc, char *argv[]) {

    MPI_Status status;
    int rank, size;
    int i, j;

    /* data to communicate */
    float  matrix[MSIZE];

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
            matrix[i] = (float)i;
        MPI_Send(matrix, MSIZE, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    } else if (rank ==1) {
        MPI_Recv(matrix, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
        printf("\nProcess 1 receives the following array from process 0.\n");
        for (i = 0; i < MSIZE; i++)
            printf("%6.2f\n", matrix[i]);
    }

    /* Quit MPI */
    MPI_Finalize();
    return 0;
}
```


Esercizio 13

Circular Shift non-blocking



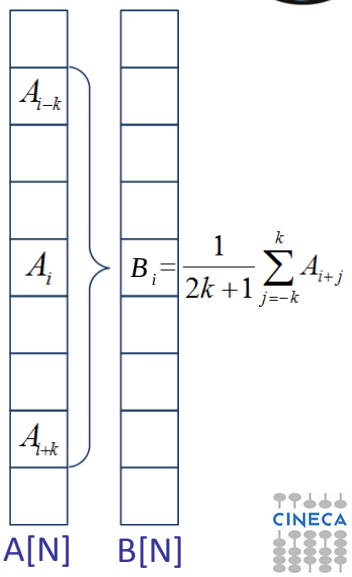
- Modificare il codice dello *shift* circolare periodico in "versione *naive*" (Esercizio 6) utilizzando le funzioni di comunicazione *non-blocking* per evitare la condizione *deadlock* per N=2000



Esercizio 14

Array smoothing

- dato un *array* A[N]
- stampare il vettore A
- per ITER volte:
  - calcolare un nuovo *array* B in cui ogni elemento sia uguale alla media aritmetica del suo valore e dei suoi K primi vicini al passo precedente
    - nota: l'array è periodico, quindi il primo e l'ultimo elemento di A sono considerati primi vicini
  - Stampare il vettore B
  - Copiare B in A e continuare l'iterazione



Esercizio 14

Array smoothing: algoritmo parallelo

- Il processo di *rank* 0
  - genera l'*array* globale di dimensione N , divisibile per il numero P di processi
  - inizializza il vettore A con A[i] =i
  - distribuisce il vettore A ai P processi i quali riceveranno N_{loc} elementi nell'array locale
- Ciascun processo ad ogni passo di *smoothing*:
  - Avvia le opportune *send non-blocking* verso i propri processi primi vicini per spedire i suoi elementi
  - Avvia le opportune *receive non-blocking* dai propri processi primi vicini per ricevere gli elementi dei vicini
  - Effettua lo *smoothing* dei soli elementi del vettore che non implicano la conoscenza di elementi di A in carico ad altri processi
  - Dopo l'avvenuta ricezione degli elementi in carico ai processi vicini, effettua lo *smoothing* dei rimanenti elementi del vettore
- Il processo di *rank* 0 ad ogni passo raccoglie i risultati parziali e li stampa

SCAI SuperComputing Applications and Innovation

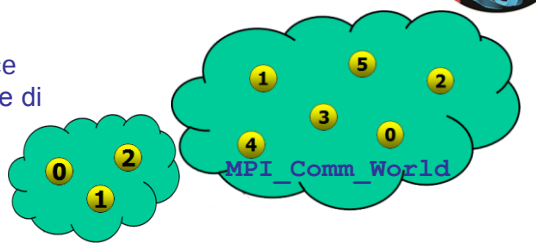
Calcolo parallelo con MPI (2ª parte)

- Approfondimento sulle comunicazioni *point-to-point*
- La comunicazione non *blocking*
- Laboratorio n° 3
- **MPI virtual topologies**
- Laboratorio n° 4

SCAI SuperComputing Applications and Innovation

Oltre `MPI_Comm_World`: i comunicatori

- Un **comunicatore** definisce l'universo di comunicazione di un insieme di processi

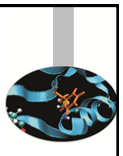


- Oltre ad **`MPI_Comm_World`**, in un programma MPI possono essere definiti altri comunicatori per specifiche esigenze, quali:
  - utilizzare funzioni collettive solo all'interno di un sotto insieme dei processi del comunicatore di default
  - utilizzare uno schema identificativo dei processi conveniente per un particolare pattern di comunicazione

Le componenti del comunicatore

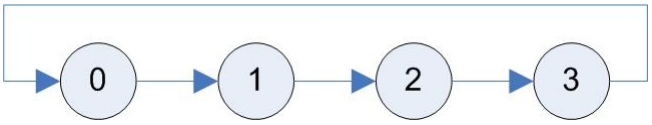
- ***Gruppo*** di processi: *un set ordinato di processi*
  - il gruppo è usato per identificare i processi
  - ad ogni processo all'interno di un gruppo è assegnato un indice (*rank*), utile per identificare il processo stesso
- ***Contesto***: utilizzato dal comunicatore per gestire l'invio/ricezione di messaggi
  - Contiene, ad esempio, l'informazione sullo stato di un messaggio da ricevere inviato con una `MPI_Isend`
- ***Attributi***: ulteriori informazioni eventualmente associate al comunicatore
  - Il *rank* del processo in grado di eseguire operazioni di I/O
  - La topologia di comunicazione sottesa

Topologie virtuali di processi



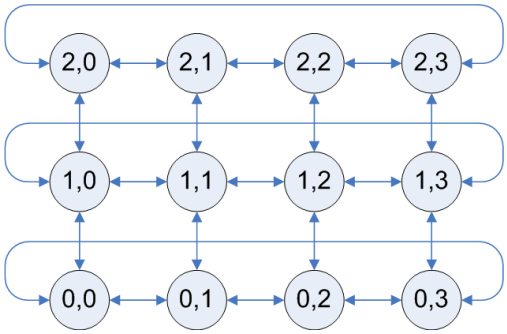
- Definizione di un nuovo schema identificativo dei processi *conveniente* per lavorare con uno specifico pattern di comunicazione:
  - semplifica la scrittura del codice
  - può consentire ad MPI di ottimizzare le comunicazioni
- Creare una topologia virtuale di processi in MPI significa definire un nuovo comunicatore, con attributi specifici
- Tipi di topologie:
  - **Cartesiane**:
    - ogni processo è identificato da un set di coordinate cartesiane ed è connesso ai propri vicini da una griglia virtuale
    - Ai bordi della griglia può essere impostata o meno la periodicità
  - **Grafo** (al di fuori di questo corso)

Circular shift: una topologia cartesiana 1D



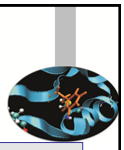
- Ogni processo comunica alla sua destra un dato
- L'ultimo processo del gruppo comunica il dato al primo

Topologia cartesiana 2D



- Ad ogni processo è associata una coppia di indici che rappresentano le sue coordinate in uno spazio cartesiano 2D
- Ad esempio, le comunicazioni possono avvenire
  - tra primi vicini *con periodicità* lungo la direzione X
  - tra primi vicini *senza periodicità* lungo la direzione Y

Creare un comunicatore con topologia cartesiana



In C

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

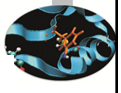
In Fortran

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
                REORDER, COMM_CART, IERROR)
```

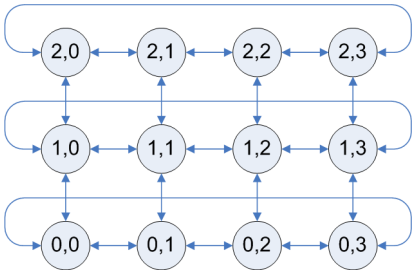
- [IN] **comm_old**: comunicatore dal quale selezionare il gruppo di processi (INTEGER)
- [IN] **ndims**: numero di dimensioni dello spazio cartesiano (INTEGER)
- [IN] **dims**: numero di processi lungo ogni direzione dello spazio cartesiano (INTEGER(*))
- [IN] **periods**: periodicità lungo le direzioni dello spazio cartesiano (LOGICAL(*))
- [IN] **reorder**: il ranking dei processi può essere riordinato per utilizzare al meglio la rete di comunicazione (LOGICAL)
- [OUT] **comm_cart**: nuovo comunicatore con l'attributo topologia cartesiana (INTEGER)

C. Truini, L. Ferraro, V.Ruggiero: MPI avanzato 35

Come usare MPI Cart create



```
int main(int argc, char **argv)
{
    ...

    int dim[2], period[2], reorder;

    ...

    dim[0]=4;
    dim[1]=3;

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

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

    ...

    return 0;
}
```

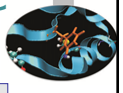


C. Truini, L. Ferraro, V.Ruggiero: MPI avanzato 36

Alcune utili funzionalità

- **MPI_Dims_Create**:
  - Calcola le dimensioni della griglia bilanciata ottimale rispetto al numero di processi e la dimensionalità della griglia dati in input
  - Utile per calcolare un vettore `dims` di input per la funzione `MPI_Cart_Create`
- Mapping tra coordinate cartesiane e *rank*
  - **MPI_Cart_coords**: sulla base della topologia definita all'interno del comunicatore, ritorna le coordinate corrispondenti al processo con un fissato *rank*
  - **MPI_Cart_rank**: sulla base della topologia definita all'interno del comunicatore, ritorna il *rank* del processo con un fissato set di coordinate cartesiane

Binding di MPI_Dims_Create



In C

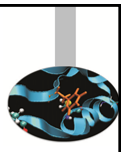
```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

In Fortran

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
```

- `[IN]` **nnodes**: numero totale di processi (`INTEGER`)
- `[IN]` **ndims**: dimesionalità dello spazio cartesiano (`INTEGER`)
- `[IN]/[OUT]` **dims**: numero di processi lungo le direzioni dello spazio cartesiano (`INTEGER(*)`)
  - Se una entry = 0 MPI_Dims_create calcola il numero di processi lungo quella direzione
  - Se una entry ≠ 0 MPI_Dims_create calcola il numero di processi lungo le direzioni "libere" compatibilmente col numero totale di processi e i valori definiti dall'utente

Rank -> Coordinate: MPI_Cart_coords



In C

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

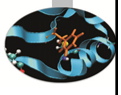
In Fortran

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS,
                IERROR)
```

- Dato il *rank* del processo, ritorna le coordinate cartesiane associate
- Argomenti:
  - [IN] **comm**: comunicatore con topologia cartesiana (INTEGER)
  - [IN] **rank**: rank del processo del quale si vogliano conoscere le coordinate (INTEGER)
  - [IN] **maxdims**: dimensionalità dello spazio cartesiano (INTEGER)
  - [OUT] **coords**: coordinate del processo rank (INTEGER(*))

C. Truini, L. Ferraro, V.Ruggiero: MPI avanzato 39

Coordinate -> Rank: MPI_Cart_rank



In C

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

In Fortran

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
```

- Date le coordinate cartesiane del processo, ritorna il *rank* associato
- Argomenti:
  - [IN] **comm**: comunicatore con topologia cartesiana (INTEGER)
  - [IN] **coords**: coordinate del processo (INTEGER(*))
  - [OUT] **rank**: *rank* del processo di coordinate coords (INTEGER)

C. Truini, L. Ferraro, V.Ruggiero: MPI avanzato 40

Sendrecv in topologia cartesiana

In C

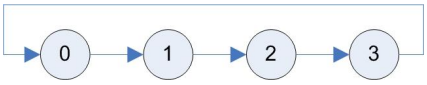
```
int MPI_Cart_shift(MPI_Comm comm, int direction,
      int disp, int *rank_source, int *rank_dest)
```

In Fortran

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, SOURCE,
      DEST, IERROR)
```

- Determina il rank del processo al/dal quale inviare/ricevere dati
- Argomenti:
  - [IN] **comm**: comunicatore con topologia cartesiana (INTEGER)
  - [IN] **direction**: indice della coordinata lungo la quale fare lo shift (INTEGER) [la numerazione degli indici parte da 0]
  - [IN] **disp**: entità dello shift (> 0: upward , < 0: downward) (INTEGER)
  - [OUT] **source**: rank del processo dal quale ricevere i dati (INTEGER)
  - [OUT] **dest**: rank del processo al quale inviare i dati (INTEGER)

Shift Circolare con topologia cartesiana 1D in C



- *Circular shift* senza topologia

```
dest = (rank + 1) % size;
source = (rank + size - 1) % size;
MPI_Sendrecv(A, MSIZE, MPI_INT, dest, tag, B, MSIZE, MPI_INT, source,
             tag, MPI_COMM_WORLD, &status);
```

- *Circular shift* con topologia cartesiana

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

CINECA SCAI SuperComputing Applications and Innovation

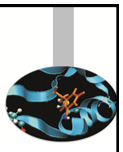
Calcolo parallelo con MPI (2ª parte)

- Approfondimento sulle comunicazioni *point-to-point*
- La comunicazione non *blocking*
- Laboratorio n° 3
- MPI virtual topologies
- **Laboratorio n° 4**

CINECA

CINECA SCAI SuperComputing Applications and Innovation

Programma della 4° sessione di laboratorio



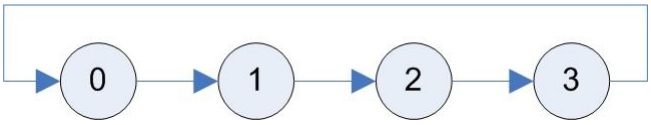
- Topologie virtuali
  - *Circular shift* con topologia cartesiana 1D (Esercizio 15)
  - Media aritmetica sui primi vicini in una topologia cartesiana 2D (Esercizio 16)

CINECA

SCAI SuperComputing Applications and Innovation

Esercizio 15

Circular shift con topologia cartesiana 1D



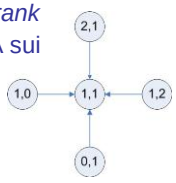
- Modificare il codice *Shift* Circolare con `MPI_Sendrecv` (Esercizio 7), in modo che siano utilizzate le funzionalità di MPI per le *virtual topologies* per determinare gli argomenti della funzione `MPI_Sendrecv`
- Nota: usare la funzione `MPI_Cart_shift` per determinare i processi sender /receiver della `MPI_Sendrecv`

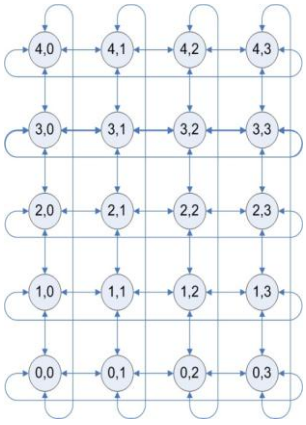
SCAI SuperComputing Applications and Innovation

Esercizio 16

Media aritmetica sui primi vicini in topologia 2D

- I processi sono distribuiti secondo una griglia rettangolare
- Ogni processo:
  - Inizializza una variabile intera A con il valore del proprio *rank*
  - calcola la media di A sui primi vicini



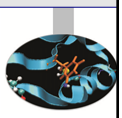


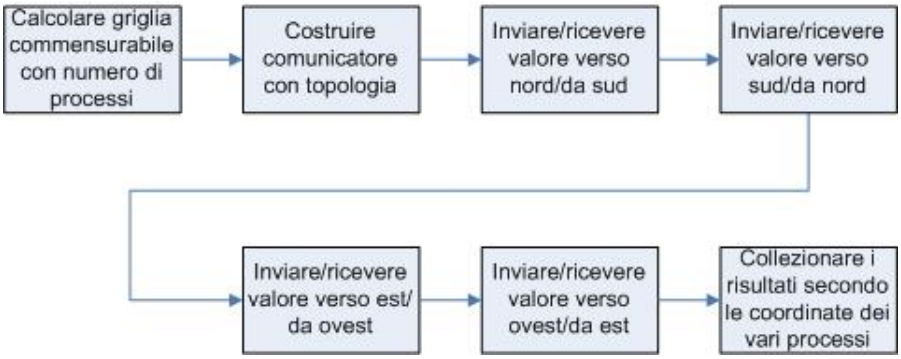
- Il processo di rank 0:
  - Raccoglie i risultati dagli altri processi
  - Riporta in output i risultati raccolti utilizzando un formato tabella organizzato secondo le coordinate dei vari processi

CINECA SCAI SuperComputing Applications and Innovation

Esercizio 16

Media aritm. sui primi vicini: flowchart delle comunicazioni





CINECA