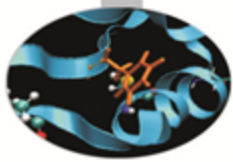
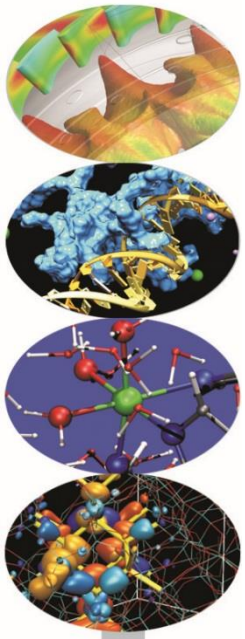
CINECA SCAI
SuperComputing Applications and Innovation



# Calcolo Parallelo con MPI

## (2° parte)

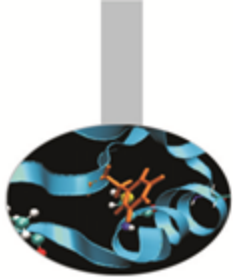



Claudia Truini
c.truini@cineca.it

Mariella Ippolito
m.ippolito@cineca.it


CINECA

# Calcolo Parallelo con MPI
(2° parte)



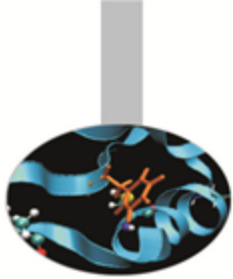
**Approfondimento sulle comunicazioni *point-to-point***

- Pattern di comunicazione *point-to-point: sendrecv*
- Synchronous Send
- Buffered Send

**La comunicazione *non-blocking***

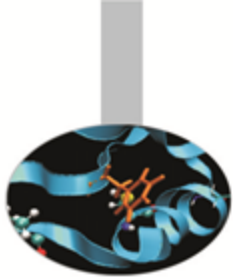
**Laboratorio 2**

# Cosa abbiamo imparato di MPI



- Concetto di “aritmetica parallela”:
  - gruppo di collaboratori che lavorano indipendentemente su parti del problema
  - sulla base dei risultati parziali, si ottiene il risultato complessivo: questa fase richiede la comunicazione tra collaboratori
- MPI come strumento per implementare la comunicazione tra processi (l’equivalente informatico dei collaboratori)
- Le 6 funzioni di base ed alcune costanti di MPI che ci permettono di implementare lo scambio di messaggi tra due processi:
  - Communication Environment:
    - MPI_Init e MPI_Finalize
    - MPI_Comm_rank e MPI_Comm_size
  - Communication point-to-point:
    - MPI_Send e MPI_Recv
  - Comunicatore di default MPI_COMM_WORLD ed alcuni MPI_Datatype

# Pattern di comunicazione



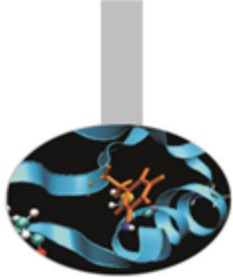
- Nella parallelizzazione di programmi reali, alcuni schemi di invio/ricezione del messaggio sono largamente diffusi:

  **pattern di comunicazione**

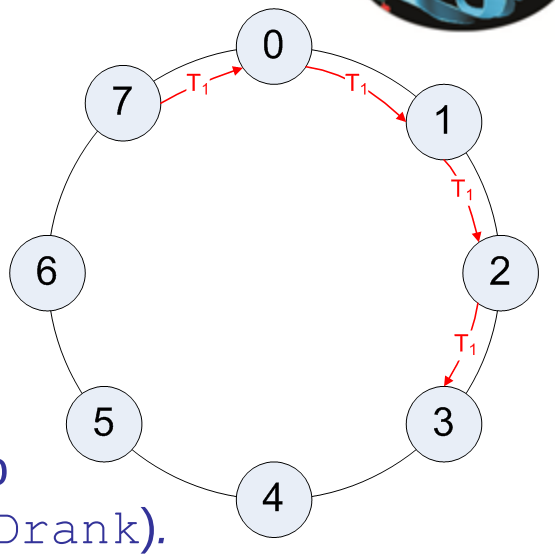
- I pattern di comunicazione possono essere di tipo
  - *point-to-point* , coinvolgono solo due processi
  - collettivi, coinvolgono più processi

- MPI mette a disposizione strumenti (funzioni MPI) per implementare alcuni pattern di comunicazione in modo corretto, robusto e semplice
  - il corretto funzionamento NON deve dipendere dal numero di processi

# Pattern di comunicazione point-to-point: shift

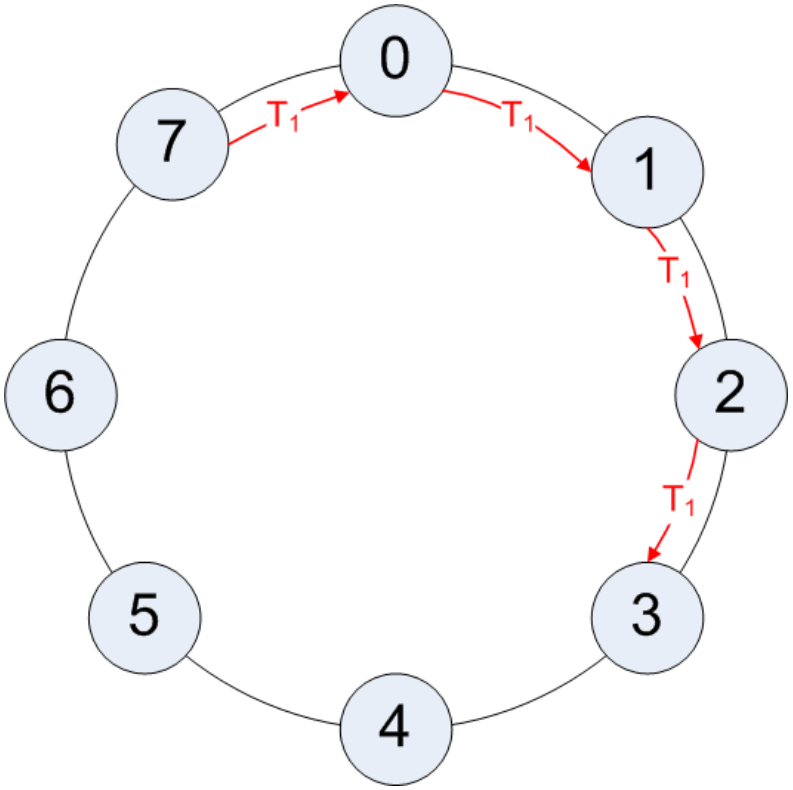


- Molti algoritmi paralleli richiedono la comunicazione tra ciascun processo ed uno (o più) dei suoi vicini, con *rank* maggiore o minore.
  Questo tipo di pattern di comunicazione è lo ***shift***



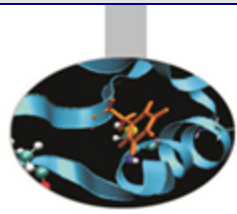

- Lo ***shift*** è un pattern *point-to-point*:
  - Ogni processo invia/riceve un set di dati in un verso (positivo/negativo) con una certa distanza di *rank* (`Drank`).
    *Per esempio:*
    - se `Drank=1` con verso positivo: il processo `i` comunica al processo `i+1`
    - se `Drank=3` con verso positivo: il processo `i` comunica al processo `i+3`
    - se `Drank=1` con verso negativo:il processo `i` comunica al processo `i-1`
  - Se lo *shift* è **periodico**:
    - Il processo con `rank=size-Drank` invia il set di dati al processo `0`
    - …

# Shift Circolare periodico

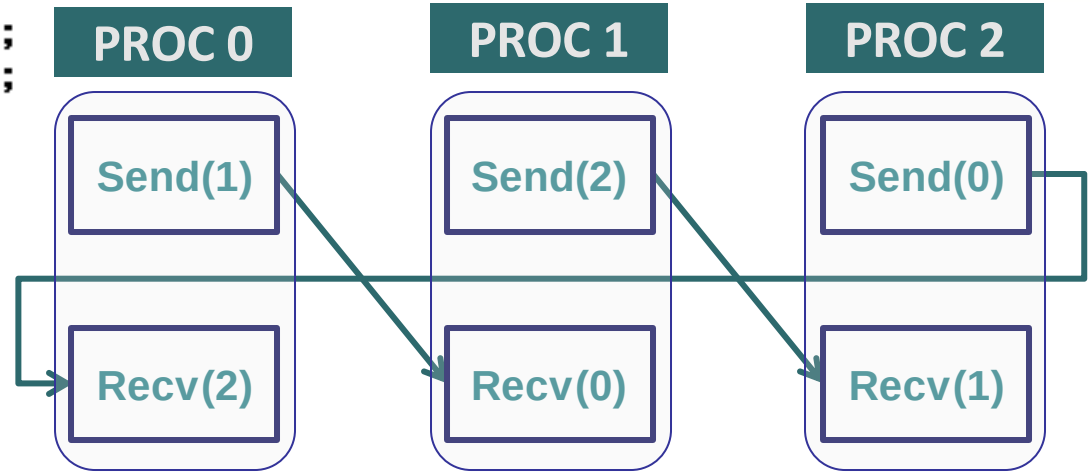



- Ogni processo genera un array `A`, popolandolo con interi pari al proprio rank
- Ogni processo invia il proprio array `A` al processo con rank immediatamente successivo
  - Periodic Boundary: l'ultimo processo invia l'array al primo processo
- Ogni processo riceve l'array `A` dal processo immediatamente precedente e lo immagazzina in un altro array `B`.
  - Periodic Boundary: il primo processo riceve l'array dall'ultimo processo
- Provare il programma dimensionando l'array A a 2000, 4000 e 5000 elementi.

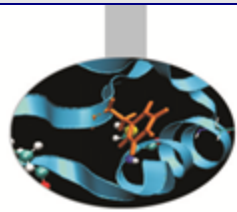
# *Shift* circolare periodico: versione *naive*



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

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

for (i = 0; i < MSIZE; i++)
    A[i] = rank;

/* starting send of array A */
MPI_Send(A, MSIZE, MPI_INT, to, tag, MPI_COMM_WORLD);
printf("Proc %d sends %d integers to proc %d\n",
    rank, MSIZE, to);

/* starting receive of array A in B */
MPI_Recv(B, MSIZE, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
printf("Proc %d receives %d integers from proc %d\n",
    rank, MSIZE, from);

/* print first content of arrays A and B */
printf("Proc %d has A[0] = %d, B[0] = %d\n\n", rank, A[0], B[0]);

/* Quit */
MPI_Finalize();
```



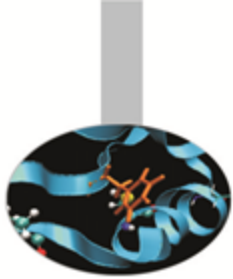

# *Shift* circolare periodico: versione *naive*



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

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

for (i = 0; i < MSIZE; i++)
    A[i] = rank;

/* starting send of array A */
MPI_Send(A, MSIZE, MPI_INT, to, tag, MPI_COMM_WORLD);
printf("Proc %d sends %d integers to proc %d\n",
    rank, MSIZE, to);

/* starting receive of array A in B */
MPI_Recv(B, MSIZE, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
printf("Proc %d receives %d integers from proc %d\n",
    rank, MSIZE, from);

/* print first content of arrays A and B */
printf("Proc %d has A[0] = %d, B[0] = %d\n\n", rank, A[0], B[0]);

/* Quit */
MPI_Finalize();
```

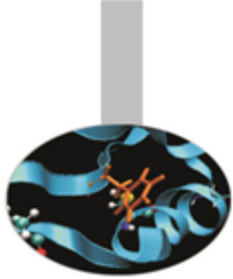
- Cosa succede girando l'esempio al crescere del valore di **MSIZE**?
- Utilizzando l'ambiente parallelo **OpenMPI** sul nostro cluster, il programma funziona correttamente a **MSIZE = 4000**
- Se **MSIZE = 5000**, il programma va in *hang*

# Il *deadlock*



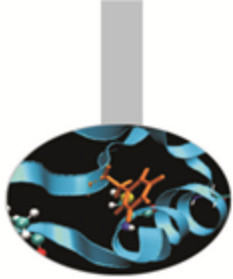
- L'implementazione *naive* dello *shift* circolare non è corretta: per `MSIZE>4000` si genera un *deadlock*
- Il *deadlock* è la condizione in cui ogni processo è in attesa di un altro per terminare la comunicazione e procedere poi nell'esecuzione del programma
- Per comprendere perché il *deadlock* si verifica per `MSIZE>4000` è necessario entrare nel dettaglio del meccanismo di scambio di messaggi tra due processi

# Cenni sul meccanismo di comunicazione



- Le funzioni send standard di MPI non 'ritornano' sino a che l'invio del messaggio non sia stato completato secondo una delle due modalità seguenti:
  - ***Buffered***: l'invio del messaggio avviene attraverso una copia dal buffer di invio in un buffer di sistema
  - ***Synchronous***: l'invio del messaggio avviene attraverso la copia diretta nel buffer di ricezione

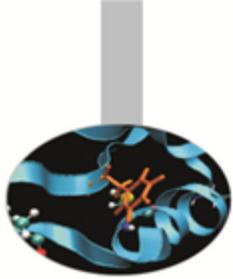
# Cosa fa `MPI_Send` nel nostro esempio



- La modalità di completamento di `MPI_Send` varia a seconda della *size* dei dati da inviare:
  - *Buffered* per *size* piccole
  - *Synchronous* per *size* grandi

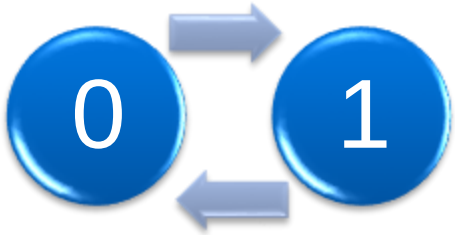
- Nel nostro caso, sino a 4000 elementi di tipo `MPI_INT` la `MPI_Send` si comporta come *buffered,* mentre a 5000 si comporta come *synchronous*
  - per MSIZE=4000 il processo può uscire fuori dalla *send* dopo che A sia stato copiato nel *buffer* locale al sistema in cui il processo è in esecuzione
  - Per MSIZE=5000 il processo può uscire fuori dalla *send* solo quando ci sia in esecuzione un'operazione di *receive* pronta ad accogliere A

# Perché il deadlock?



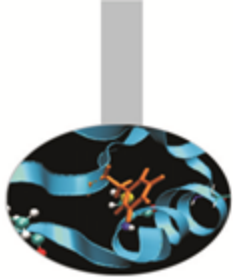
- Nel nostro caso l'algoritmo è del tipo

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



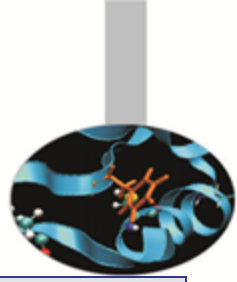

- Per MSIZE=5000, ci sono due *send* in attesa di due *receive*, ma le *receive* potranno essere eseguite solo dopo che le reciproche *send* siano state completate.
- DEADLOCK!

# Soluzione del *deadlock* nello *shift* circolare: *Send-Receive*



- Abbiamo la necessità di una funzione che tratti internamente l'ordine delle operazioni di *send* e *receive*
- La funzione che svolge questo compito è `MPI_Sendrecv`
  - La funzionalità MPI *send-receive* è utile quando un processo deve contemporaneamente inviare e ricevere dati
  - Può essere usata per implementare pattern di comunicazione di tipo *shift*

# Binding MPI_Sendrecv



In C

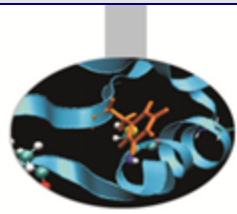
```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype s_dtype,
  int dest,int stag,void *rbuf,int rcount,MPI_Datatype r_type,
  int src, int rtag,MPI_Comm comm, MPI_Status *status)
```

In Fortran

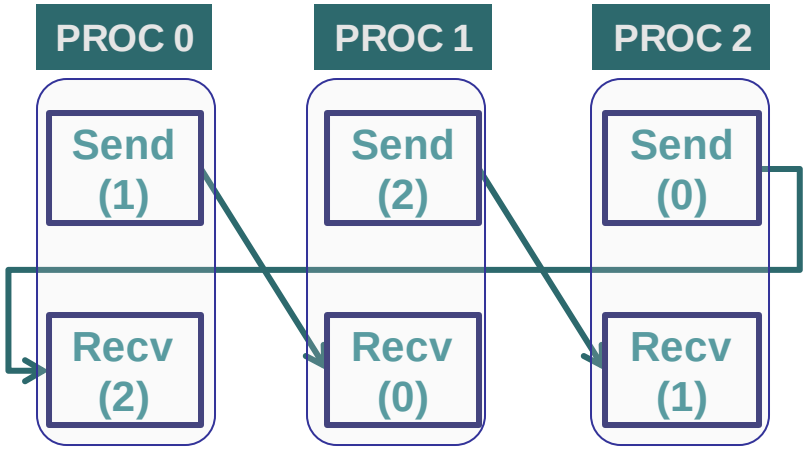
```
MPI_SENDRECV(SBUF, SCOUNT, S_DTYPE, DEST, STAG,
             RBUF, RCOUNT, R_DTYPE, SRC, RTAG,
             COMM, STATUS, ERR)
```

- I primi argomenti sono relativi alla *send*, gli altri alla *receive*
- Argomenti significativi:
  - [IN] **dest** è il *rank* del *receiver* all'interno del comunicatore **comm**
  - [IN] **stag** è l'identificativo del *send message*
  - [IN] **src** è il *rank* del *sender* all'interno del comunicatore **comm**
  - [IN] **rtag** è l'identificativo del *receive message*
  - [OUT] rbuf, status

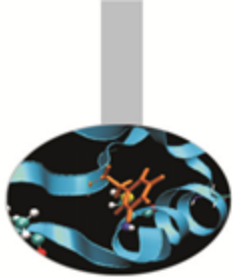
# *Shift* circolare: versione con *Send-Receive*



```
#include <stdio.h>
#include <mpi.h>
#define MSIZE 50000

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

    MPI_Status status;
    int rank, size, tag, to, from;

    int A[MSIZE], B[MSIZE], i;

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

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

    for (i = 0; i < MSIZE; i++)
        A[i] = rank;

    MPI_Sendrecv(A, MSIZE, MPI_INT, to, 201, /* sending info */
                 B, MSIZE, MPI_INT, from, 201, /* recving info */
                 MPI_COMM_WORLD, &status);

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

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



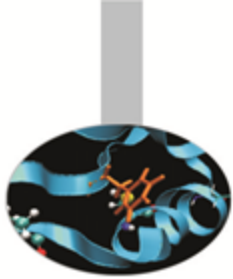

# Modalità di comunicazione



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

# Criteri di completamento della comunicazione

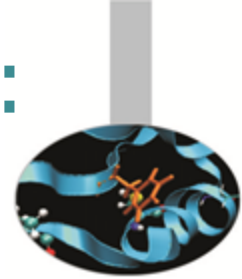


- Dal punto di vista non locale, ovvero di entrambi i processi coinvolti nella comunicazione, *è rilevante il criterio in base al quale si considera completata la comunicazione*

| Funzionalità | Criterio di completamento |
|---|---|
| *Synchronous 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 *synchronous* 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

# Synchronous SEND blocking: MPI_Ssend

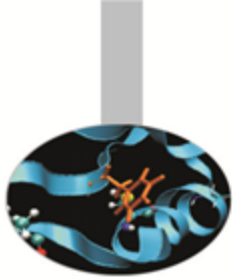


- Per il completamento dell'operazione il *sender* deve essere informato dal *receiver* che il messaggio è stato ricevuto
- **Gli argomenti della funzione sono gli stessi della `MPI_Send`**
- Pro: è la modalità di comunicazione *point-to-point* più semplice ed affidabile
- Contro: può comportare rilevanti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

# Synchronous Send: esempio (C)

```c
#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

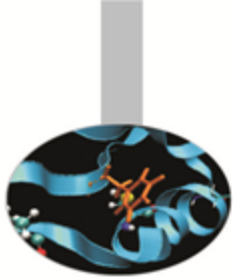


- Una ***buffered send*** è completata immediatamente, non appena il processo ha copiato il messaggio su un opportuno *buffer* di trasmissione

- 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

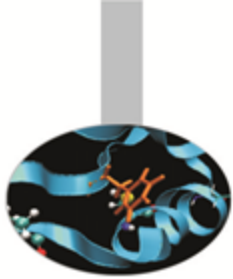
```
MPI_Buffer_attach(void *buf, int size)
```

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:
  - [IN] **`buf`** è l'indirizzo iniziale del buffer da allocare
  - [IN] **`size`** è un **`int`** (**`INTEGER`**) e contiene la dimensione in byte del buffer da allocare

# Gestione dei buffer: MPI_Buffer_detach



In C

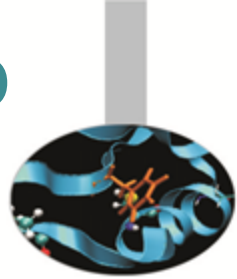
```
MPI_Buffer_detach(void *buf, int *size)
```

In Fortran

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

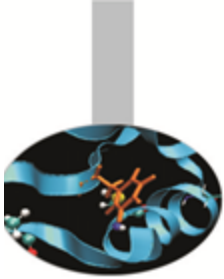
- Consente di rilasciare il buffer creato con **MPI_Buffer_attach**
- Argomenti:
  - [OUT] **buf** è l'indirizzo iniziale del buffer da deallocare
  - [OUT] **size** è un **int*** (INTEGER) e contiene la dimensione in byte del buffer deallocato

# *Buffered Send*: un esempio (C)



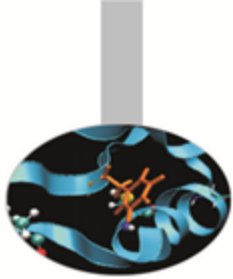
```c
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

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

   MPI_Status status;
   int rank, size, i, mpibuffer_length;
   double *mpibuffer;
   double  vector[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++)   vector[i] = (double) i;

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

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

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

# *Buffered Send*: un esempio (FORTRAN)



```
program main
use mpi
implicit none
integer ierr, rank, size, i, status(MPI_STATUS_SIZE)
integer, parameter :: MSIZE=10000
double precision matrix(MSIZE)

INTEGER mpibuffer_length, typesize
double precision, DIMENSION(:), ALLOCATABLE :: mpibuffer

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

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

   CALL MPI_BSEND(matrix, MSIZE, MPI_DOUBLE_PRECISION, 1, 666, MPI_COMM_WORLD, ierr)
   CALL MPI_BUFFER_DETACH(mpibuffer, mpibuffer_length, 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
```

# Calcolo Parallelo con MPI
(2° parte)



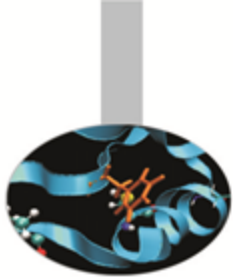
**Approfondimento sulle comunicazioni *point-to-point***

- Pattern di comunicazione *point-to-point: sendrecv*
- Synchronous Send
- Buffered Send

**La comunicazione non *blocking***

**Laboratorio 2**

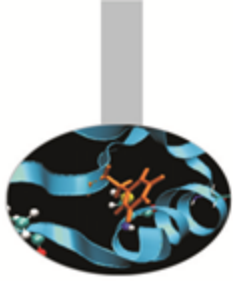
# Modalità di comunicazione



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

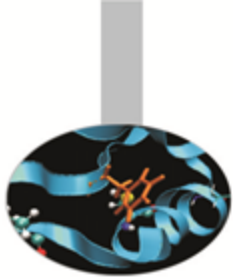
- **Blocking**:
  - il controllo è restituito al processo che ha invocato la primitiva di comunicazione solo quando la stessa è stata completata
- **Non blocking**:
  - il controllo è restituito al processo che ha invocato la primitiva di comunicazione quando la stessa è stata eseguita
  - il controllo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

# Comunicazioni non-blocking

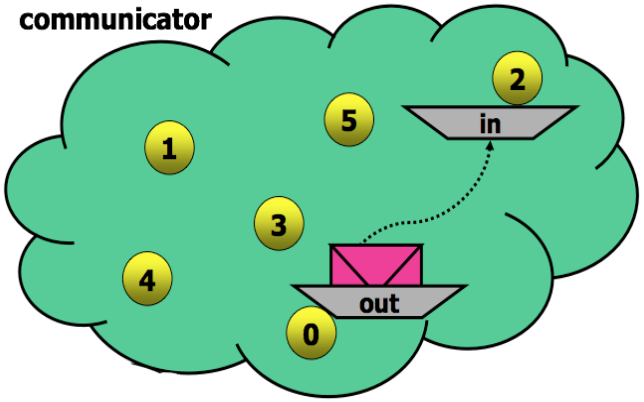


- Una comunicazione *non-blocking* è tipicamente costituita da tre fasi successive:
  1. L'inizio della operazione di *send* o *receive* del messaggio
  2. Lo svolgimento di un'attività che non implichi l'accesso ai dati coinvolti nella operazione di comunicazione avviata
  3. Controllo/attesa del completamento 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



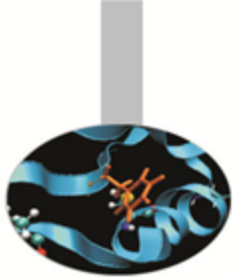
- Dopo che la spedizione è stata avviata il controllo torna al processo *sender*



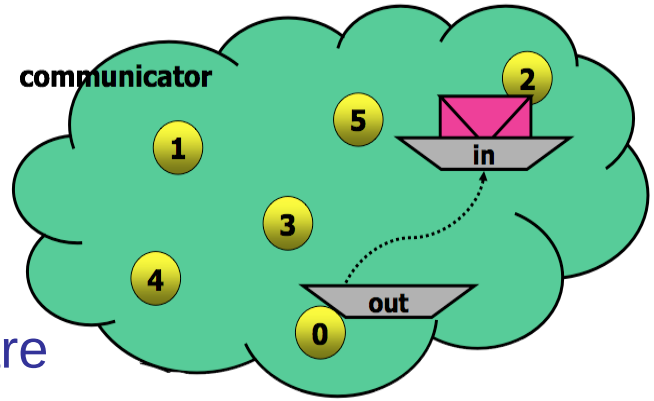

- Prima di riutilizzare le aree di memoria coinvolte nella comunicazione, il processo *sender* deve controllare che l'operazione sia stata completata, attraverso opportune funzioni della libreria MPI

- Anche per la *send non-blocking, sono previste* le diverse modalità di completamento della comunicazione

# RECEIVE non-blocking : MPI_Irecv



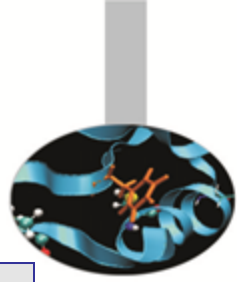
- Dopo che la fase di ricezione è stata avviata il controllo torna al processo *receiver*




- 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

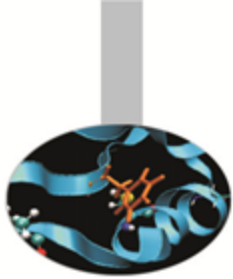
```
MPI_ISEND(buf, count, dtype, dest, tag, comm, req, err)
```

```
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:

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

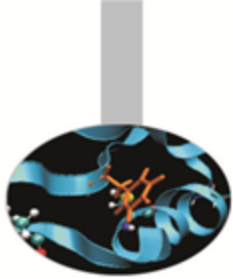
# Comunicazioni non blocking: completamento



- 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)
```

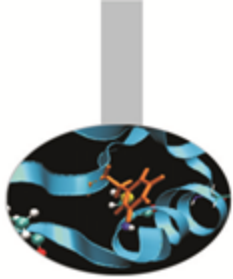
In Fortran

```
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**

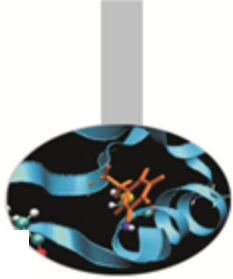
```
int MPI_Test(MPI_Request *request,
             int *flag, MPI_Status *status)
```

**In Fortran**

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

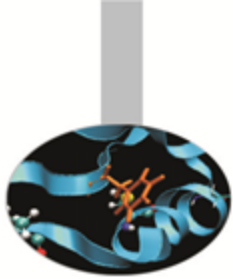
- Argomenti
  - `[IN]` **`request`** è l'*handler* necessario per referenziare la comunicazione di cui controllare il completamento (`INTEGER`)
  - `[OUT]` **`flag`** conterrà **`TRUE`** se la comunicazione è stata completata, **`FALSE`** altrimenti (`LOGICAL`)
  - `[OUT]` **`status`** conterrà lo stato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

# Calcolo Parallelo con MPI
(2° parte)



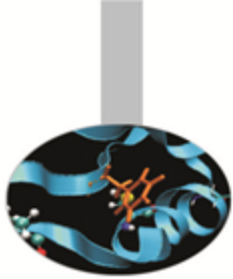
## Approfondimento sulle comunicazioni *point-to-point*

- Pattern di comunicazione *point-to-point: sendrecv*
- Synchronous Send
- Buffered Send

## La comunicazione non *blocking*

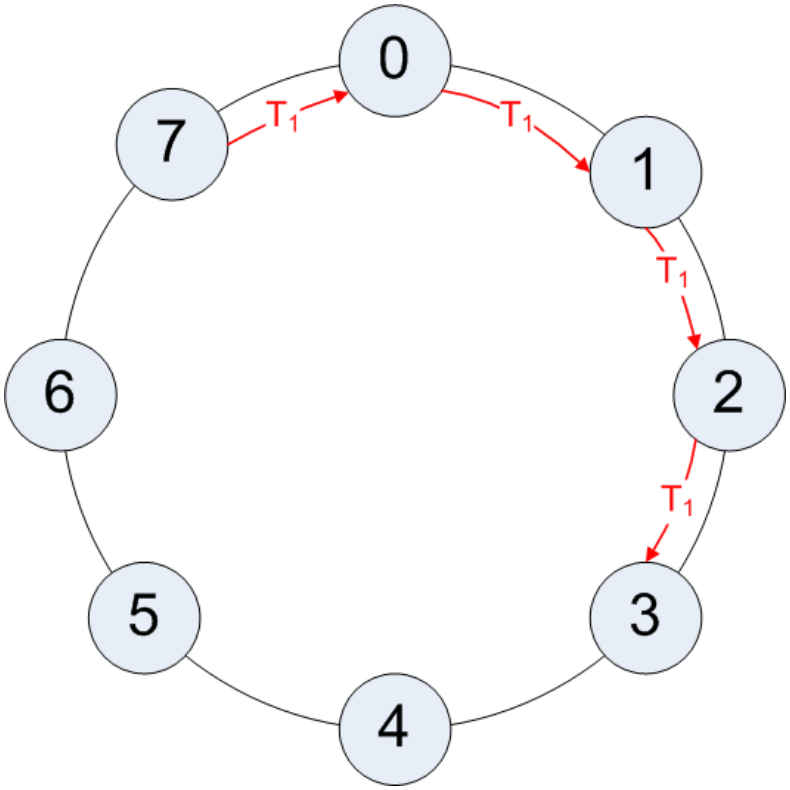
## Laboratorio 2

# Programma della 2° sessione di laboratorio



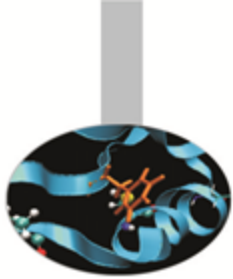
- Pattern di comunicazione
  - *Shift* circolare con `MPI_Sendrecv` (Esercizio 7)
- Modalità di completamento della comunicazione
  - Uso della *synchronous send*, della *buffered send* e delle funzioni di gestione dei buffer (Esercizio 12)
- Uso delle funzioni di comunicazione *non-blocking*
  - *Non blocking circular shift* (Esercizio 13)

# Shift Circolare periodico con `MPI_Sendrecv`




- Ogni processo genera un array `A`, popolandolo con interi pari al proprio rank
- Ogni processo invia il proprio array `A` al processo con rank immediatamente successivo
  - Periodic Boundary: L'ultimo processo invia l'array al primo processo
- Ogni processo riceve l'array `A` dal processo immediatamente precedente e lo immagazzina in un altro array `B`.
  - Periodic Boundary: il primo processo riceve l'array dall'ultimo processo
- Le comunicazioni devono essere di tipo *Sendrecv*

# Funzioni per il timing: MPI_Wtime e MPI_Wtick

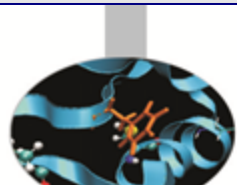


- È importante conoscere il tempo impiegato dal codice nelle singole parti per poter analizzare le performance
- MPI_Wtime: fornisce il numero floating-point di secondi intercorso tra due sue chiamate successive

```
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

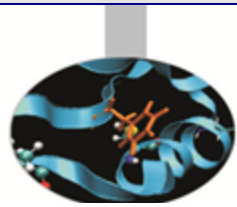
# Synchronous Send blocking



- 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`

```c
#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;
}
```

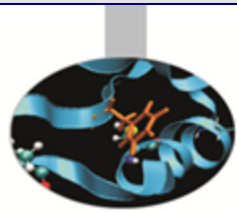
# Buffered Send blocking



- 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;
}
```

# 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=5000