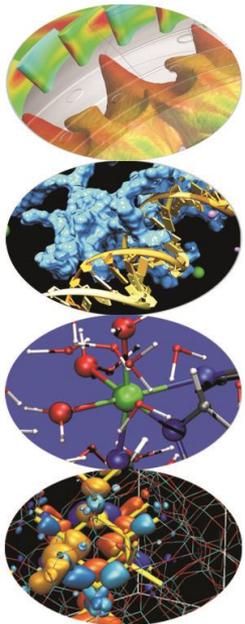
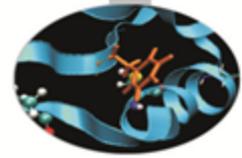


Introduzione al Calcolo Parallelo con MPI e OpenMP

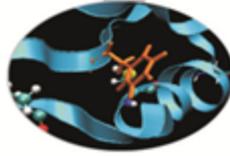


Claudia Truini
c.truini@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

Cristiano Padrin
c.padrin@cineca.it

Presentazione del corso

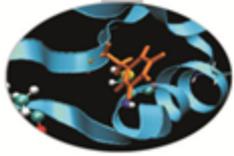


Cosa è il calcolo parallelo

Calcolo parallelo: MPI (base e avanzato)

Calcolo parallelo: OpenMP

Problema 1: Serie di Fibonacci

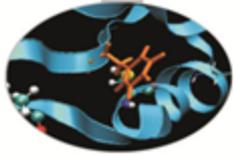


Calcolare e stampare i primi N elementi della serie di Fibonacci

‡ La serie di Fibonacci {1, 1, 2, 3, 5, 8, ...} è così definita:

$$f_1 = 1; \quad f_2 = 1$$
$$f_i = f_{i-1} + f_{i-2} \quad \forall i > 2$$

Problema 2: Serie geometrica



Calcolare la somma parziale N-sima della serie geometrica

‡ La serie geometrica è così definita:

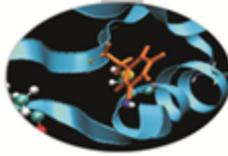
$$g_1 = x, \quad g_2 = x^2, \quad g_3 = x^3, \dots$$

ovvero $g_i = x^i \quad \forall i > 0$

‡ Dobbiamo calcolare:

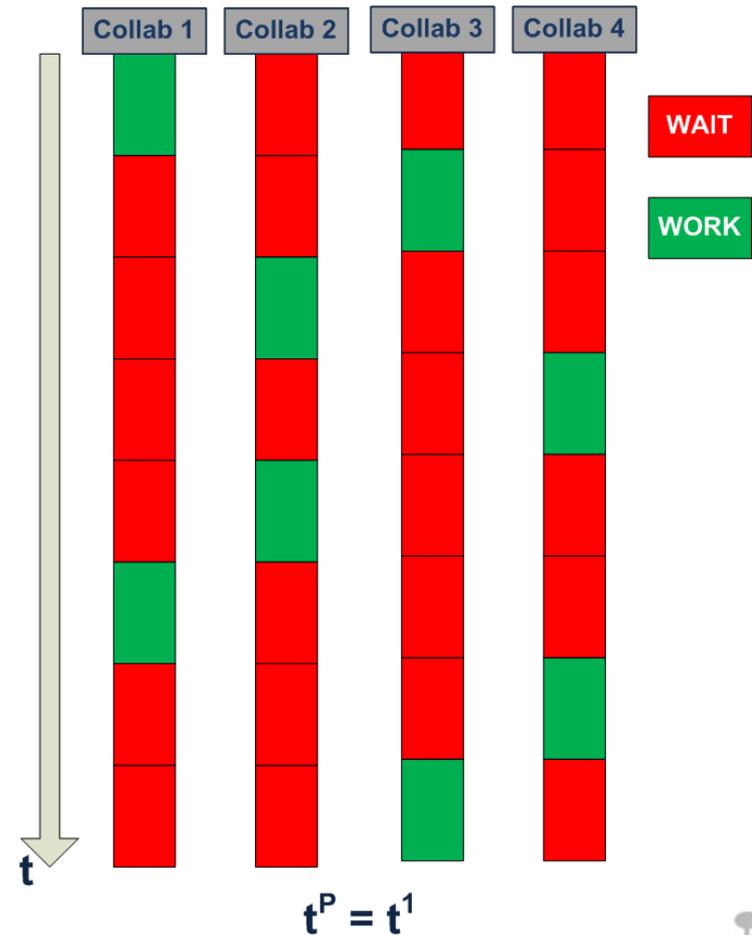
$$G_N = \sum_{i=1}^N x^i$$

Risoluzione del problema 1 con P collaboratori

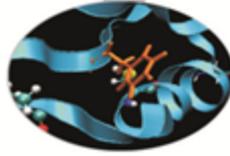


Come calcolare nel minor tempo i primi N numeri di Fibonacci?

- ⌚ Il generico f_i dipende dai 2 numeri precedenti della serie, dunque può essere calcolato solo dopo che siano stati determinati f_{i-1} e f_{i-2}
- ⌚ Utilizzando P collaboratori:
 1. Un qualsiasi collaboratore calcola e stampa il 3° termine
 2. Un qualsiasi collaboratore calcola e stampa il 4° termine
 3. ...
- ⌚ Utilizzando P collaboratori, il tempo necessario all'operazione è uguale al tempo necessario ad un solo collaboratore!



Risoluzione del problema 2 con P collaboratori

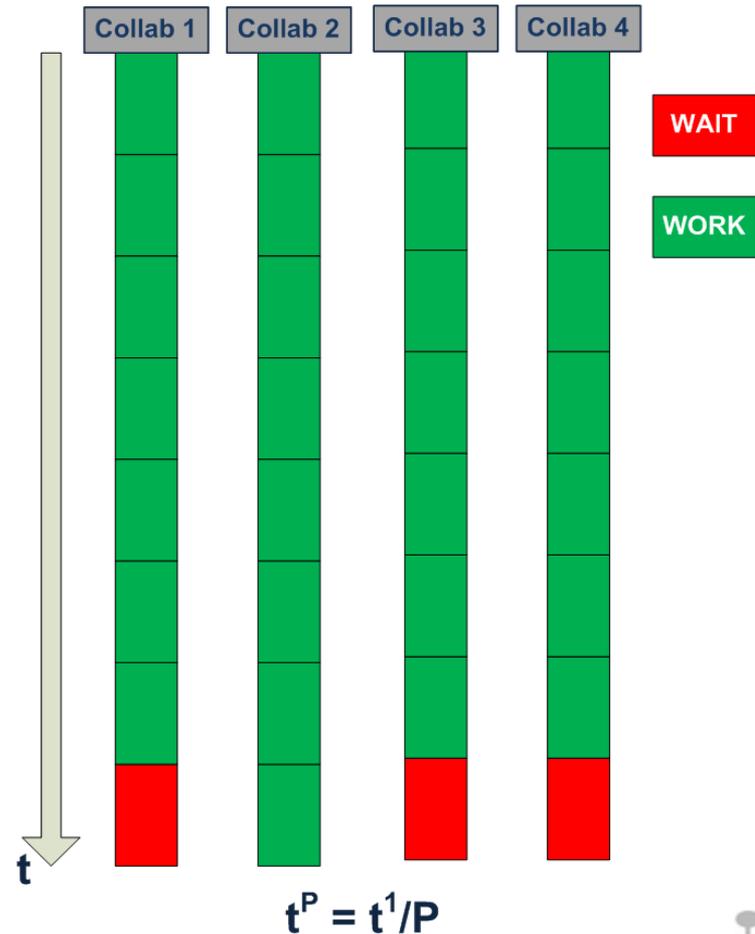


$$G_N = \sum_{i=1}^N x^i = \sum_{i=1}^P \left(\sum_{j=1}^{N/P} x^{P \frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^P S_i$$

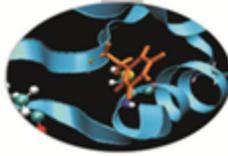
dove

$$S_i = \sum_{j=1}^{N/P} x^{P \frac{N}{P}(i-1)+j}$$

- Utilizzando P collaboratori:
 - Ogni collaboratore calcola una delle P somme parziali S_j
 - Solo uno dei collaboratori somma i P contributi appena calcolati
 - Il tempo impiegato è uguale a $1/P$ del tempo che avrebbe impiegato un solo collaboratore



Benefici nell'uso di P collaboratori



Se il problema e l'algoritmo possono essere decomposti in task indipendenti:

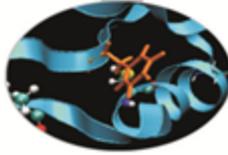
- ✦ Il lavoro complessivo potrà essere completato in un tempo minore

In condizioni ideali, il tempo di calcolo diventa $t^P = t^1/P$, dove t^m è il tempo di calcolo con m collaboratori

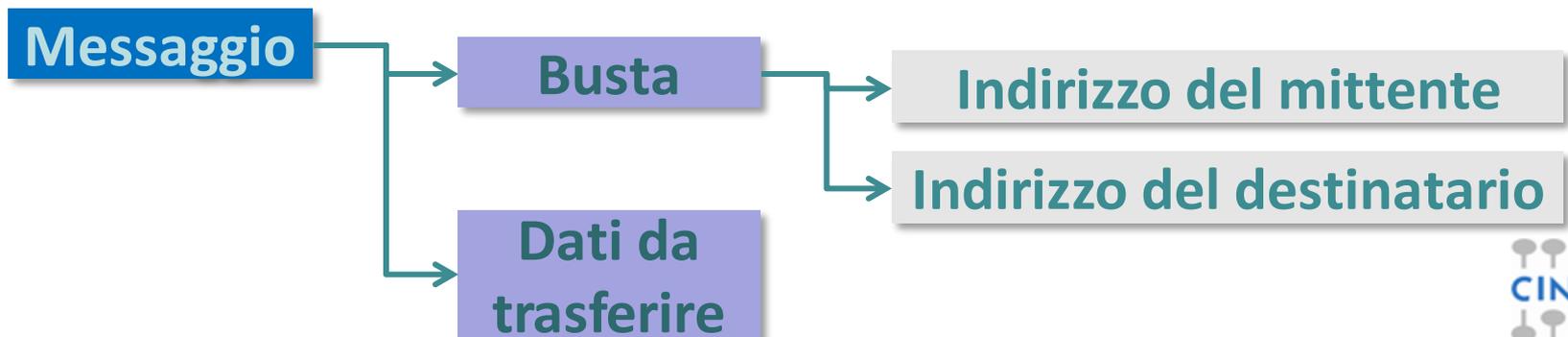
- ✦ Ogni collaboratore dovrà “ricordare” meno dati

In condizioni ideali, la quantità di dati da ricordare diventa $A^P = A^1/P$, in cui A^m è la size dei dati nel caso con m collaboratori

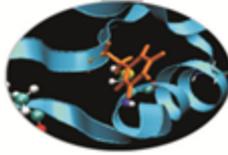
La comunicazione tra collaboratori: il messaggio



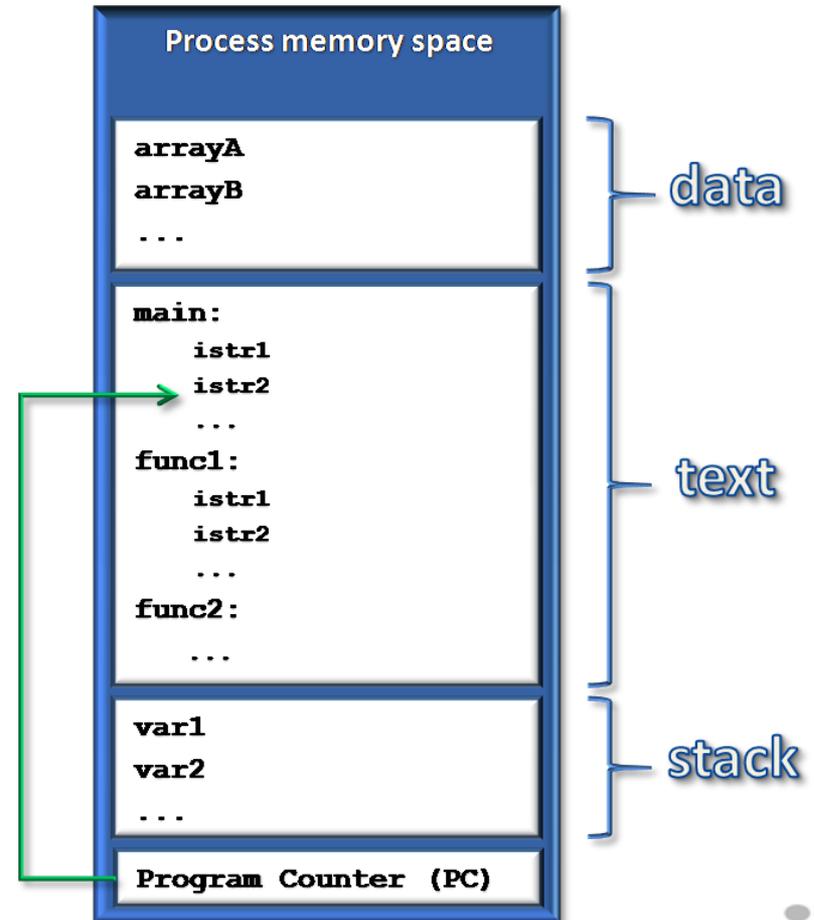
- Per una “fruttuosa cooperazione” tra P collaboratori è necessario che gli stessi possano scambiarsi dati
- Problema 2: Serie geometrica quando i P collaboratori terminano il calcolo della somma parziale di propria competenza, uno di essi
 - richiede a tutti gli altri la somma parziale di competenza
 - somma al proprio i $P-1$ risultati parziali ricevuti
- Il trasferimento di dati tra collaboratori può avvenire attraverso la *spedizione/ricezione di un messaggio*:



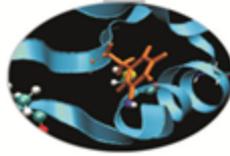
Il collaboratore nel *computing*: il processo



- 1 Un processo è un'istanza in esecuzione di un programma
- 2 Il processo mantiene in memoria i dati e le istruzioni del programma, oltre ad altre informazioni necessarie al controllo del flusso di esecuzione
 - 3 Text
 istruzioni del programma
 - 4 Data
 variabili globali
 - 5 Stack
 variabili locali alle funzioni
 - 6 Program Counter (PC)
 puntatore all'istruzione corrente

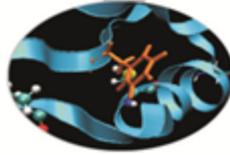


Gruppo di collaboratori → Calcolo Parallelo



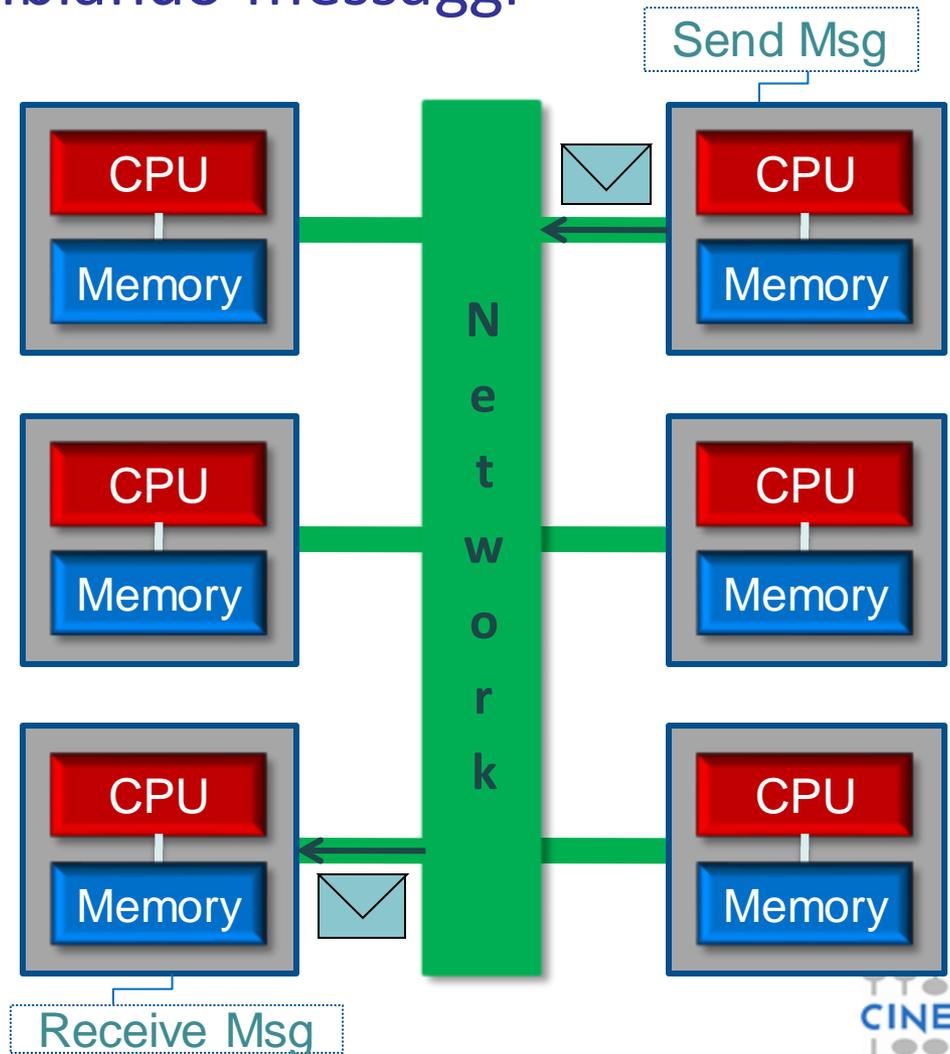
- † Il calcolo parallelo, in generale, è l'uso simultaneo di più processi per risolvere un unico problema computazionale
- † Per girare con più processi, un problema è diviso in parti discrete che possono essere risolte concorrentemente
- † Le istruzioni che compongono ogni parte sono eseguite contemporaneamente su processi diversi
- † Benefici:
 - ‡ si possono risolvere problemi più “grandi”, superando i vincoli di memoria
 - ‡ si riduce il tempo di calcolo

Collaboratori scambiano messaggi → Message-passing

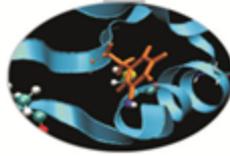


N processi cooperano scambiando messaggi

- Ogni processo svolge autonomamente la parte indipendente del task
- Ogni processo accede in lettura e scrittura ai soli dati disponibili nella sua memoria
- E' necessario **scambiare messaggi** tra i processi coinvolti quando
 - un processo deve accedere ad un dato presente nella memoria di un altro processo
 - più processi devono sincronizzarsi per l'esecuzione di un flusso d'istruzioni

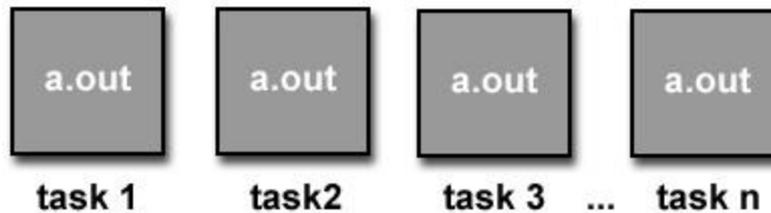


Modello di esecuzione: SPMD



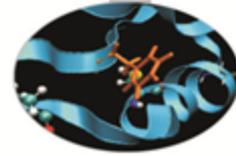
SPMD = Single Program Multiple Data

Ogni processo esegue lo stesso programma, ma opera in generale su dati diversi (nella propria memoria locale)

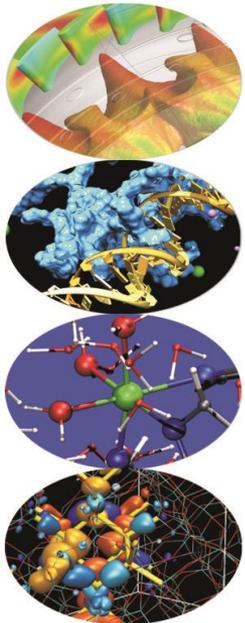


Processi differenti possono eseguire parti di codice differenti:

```
if (I am process 1)
    ... do something ...
if (I am process 2)
    ... do something else ...
```



Introduzione al Calcolo Parallelo con MPI

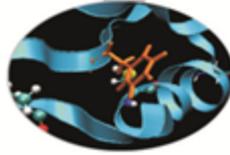


Claudia Truini
c.truini@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

Cristiano Padrin
c.padrin@cineca.it

Calcolo parallelo con MPI (1^a parte)



Le sei funzioni di base: introduzione alla comunicazione *point-to-point*

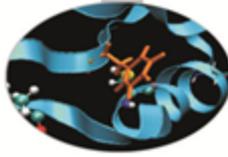
Laboratorio n° 1

Pattern di comunicazione *point-to-point*: *sendrecv*

Introduzione alle comunicazioni collettive

Laboratorio n° 2

Cos'è MPI



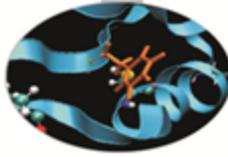
🔑 MPI:

- 🔑 MPI è acronimo di **M**essage **P**assing **I**nterface
- 🔑 MPI è una Application Programming Interface (API)
- 🔑 **Standard** per sviluppatori ed utenti
<http://www.mpi-forum.org>

🔑 Calcolo Parallelo con MPI

- 🔑 MPI è uno strumento di programmazione che permette di implementare il modello di calcolo parallelo appena descritto
- 🔑 MPI consente di:
 - 🔑 generare e gestire il gruppo di collaboratori (processi)
 - 🔑 scambiare dati tra loro

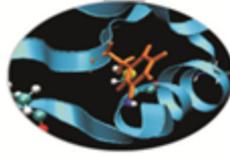
Funzionalità di MPI



La libreria MPI fornisce:

- ✦ Funzioni di management della comunicazione
 - ✦ Definizione/identificazione di gruppi di processi (comunicatori) coinvolti nella comunicazione
 - ✦ Definizione/gestione dell'identità del singolo processo, all'interno di un gruppo
- ✦ Funzioni di scambio messaggi
 - ✦ Inviare/ricevere dati da un processo
 - ✦ Inviare/ricevere dati da un gruppo di processi
- ✦ Nuovi tipi di dati e costanti (macro) che semplificano la vita al programmatore

Formato delle chiamate MPI



† In C:

```
err = MPI_Xxxxx(parameter, ...)
```

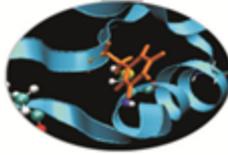
- ‡ MPI_ è prefisso di tutte le funzioni MPI
- ‡ Dopo il prefisso, la prima lettera è maiuscola e tutte le altre minuscole
- ‡ Praticamente tutte le funzioni MPI tornano un codice d'errore intero
- ‡ Le macro sono scritte tutte in maiuscolo

† In Fortran:

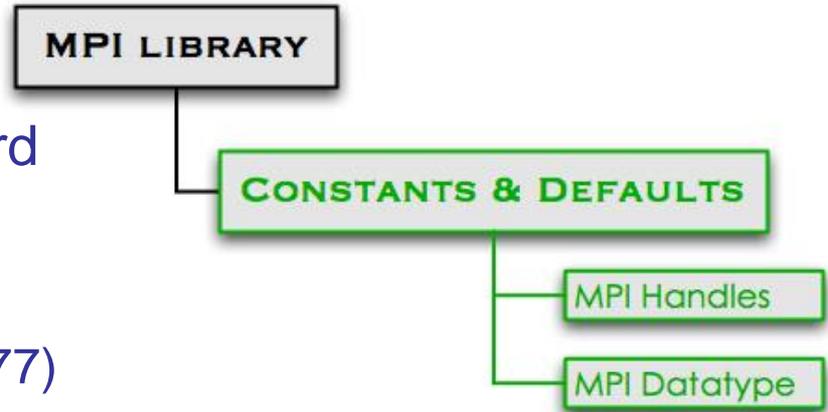
```
call MPI_XXXX(parameter,..., err)
```

- ‡ MPI_ è prefisso di tutte le subroutine MPI
- ‡ Anche se il Fortran è *case insensitive*, le subroutine e le costanti MPI sono convenzionalmente scritte in maiuscolo
- ‡ L'ultimo parametro è il codice d'errore (**INTEGER**)

Constants & Defaults

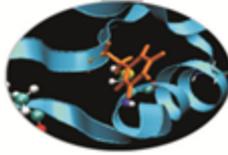


- Tutti i programmi che usano MPI devono includere *l'header file (o modulo) standard*
 - C: **mpi.h**
 - Fortran: **mpif.h**
 - Fortran: **use mpi** (no Fortran 77)
 - Fortran: **use mpi_f08** ← da MPI 3.0



- Nell'*header file* standard sono contenute **definizioni, macro e prototipi di funzioni** necessari per la compilazione di un programma MPI
 - MPI mantiene strutture di dati interne legate alla comunicazione, referenziabili tramite *MPI Handles*
 - MPI riferenzia i tipi di dati standard dei linguaggi C/Fortran attraverso *MPI Datatype*

Communication Environment

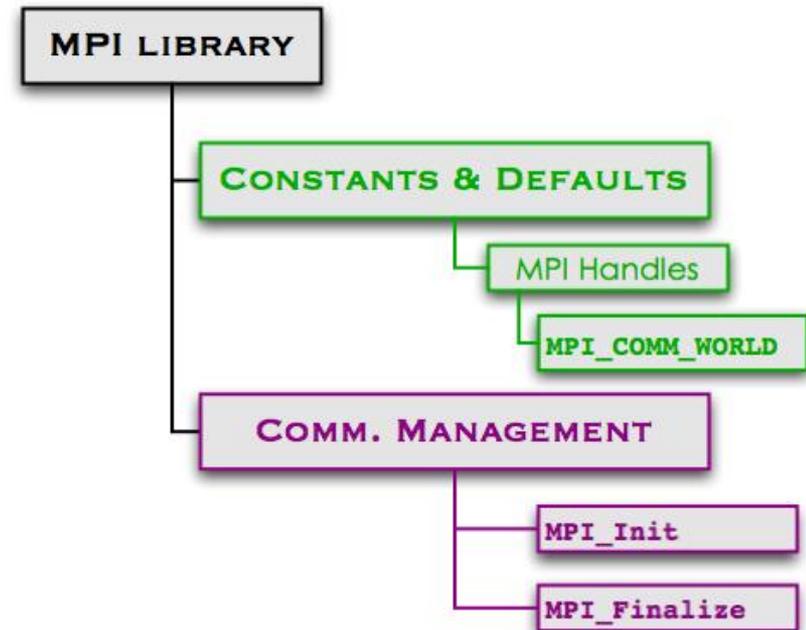


MPI_Init

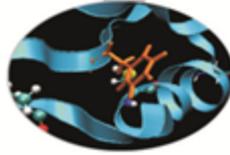
- inizializza l'ambiente di comunicazione
- tutti i programmi MPI devono contenere una sua chiamata
- deve essere chiamata prima di qualsiasi altra routine MPI
- deve essere chiamata una sola volta

MPI_Finalize

- termina l'ambiente MPI
- conclude la fase di comunicazione
- provvede al rilascio pulito dell'ambiente di comunicazione
- non è possibile eseguire ulteriori funzione MPI dopo la MPI_Finalize



MPI_Init e MPI_Finalize



In C

```
int MPI_Init(int *argc, char **argv)
```

```
int MPI_Finalize(void)
```

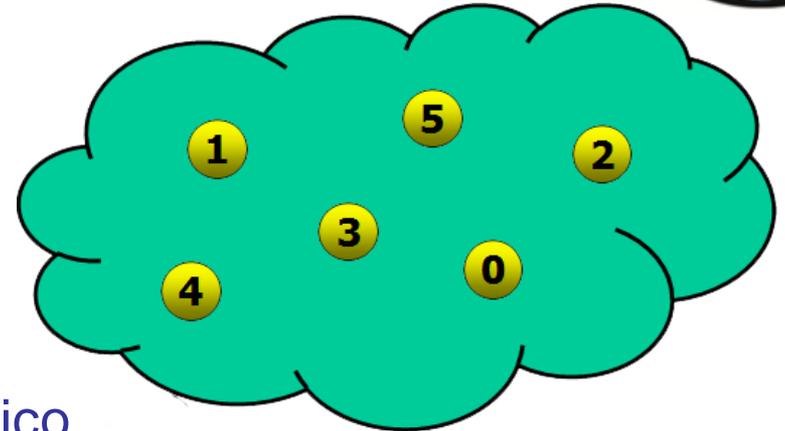
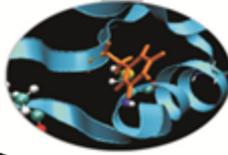
In Fortran

```
INTEGER err  
MPI_INIT(err)
```

```
INTEGER err  
MPI_FINALIZE(err)
```

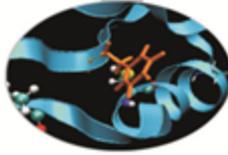
In C, la funzione `MPI_Init` esegue il parsing degli argomenti forniti al programma da linea di comando

Comunicatori



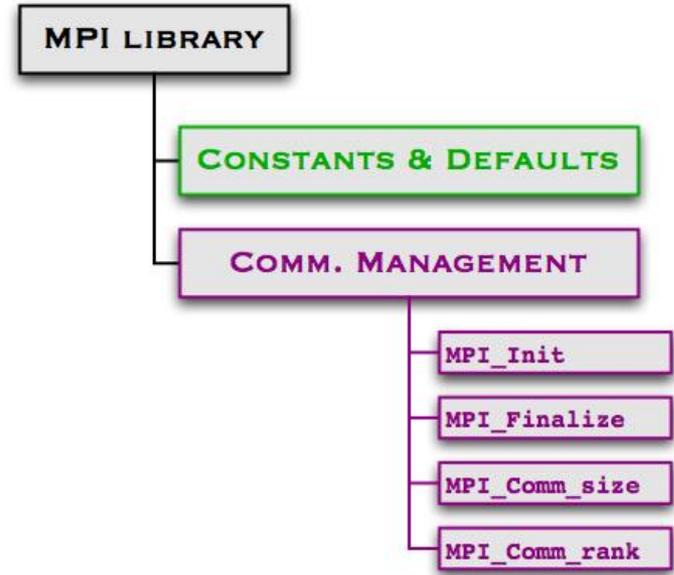
- † Un comunicatore è un “oggetto” contenente un *gruppo* di processi ed un set di attributi associati
- † All’interno di un comunicatore ogni processo ha un identificativo unico
- † Due o più processi possono comunicare solo se fanno parte dello stesso comunicatore
- † La funzione `MPI_Init` inizializza il comunicatore di *default* `MPI_COMM_WORLD`, che comprende tutti i processi che partecipano al job parallelo
- † In un programma MPI può essere definito più di un comunicatore

Informazioni dal comunicatore



📌 *Communicator size*

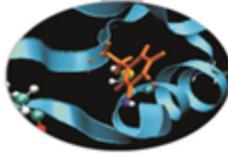
- 📌 La *size* di un comunicatore è la dimensione del gruppo di processi in esso contenuti
- 📌 Un processo può determinare la *size* di un comunicatore di cui fa parte con una chiamata alla funzione **MPI_Comm_size**
- 📌 La *size* di un comunicatore è un intero



📌 *Process rank*

- 📌 Un processo può determinare il proprio identificativo (*rank*) in un comunicatore con una chiamata a **MPI_Comm_rank**
- 📌 I *rank* dei processi che fanno parte di un comunicatore sono numeri interi, consecutivi a partire da 0

MPI_Comm_size e MPI_Comm_rank



In C

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

In Fortran

```
MPI_COMM_SIZE(comm, size, err)
```

```
MPI_COMM_RANK(comm, rank, err)
```

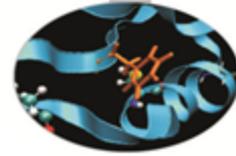
Input:

- **comm** è di tipo **MPI_Comm** (**INTEGER**) ed è il comunicatore di cui si vuole conoscere la dimensione o all'interno del quale si vuole conoscere il rank del processo chiamante

Output:

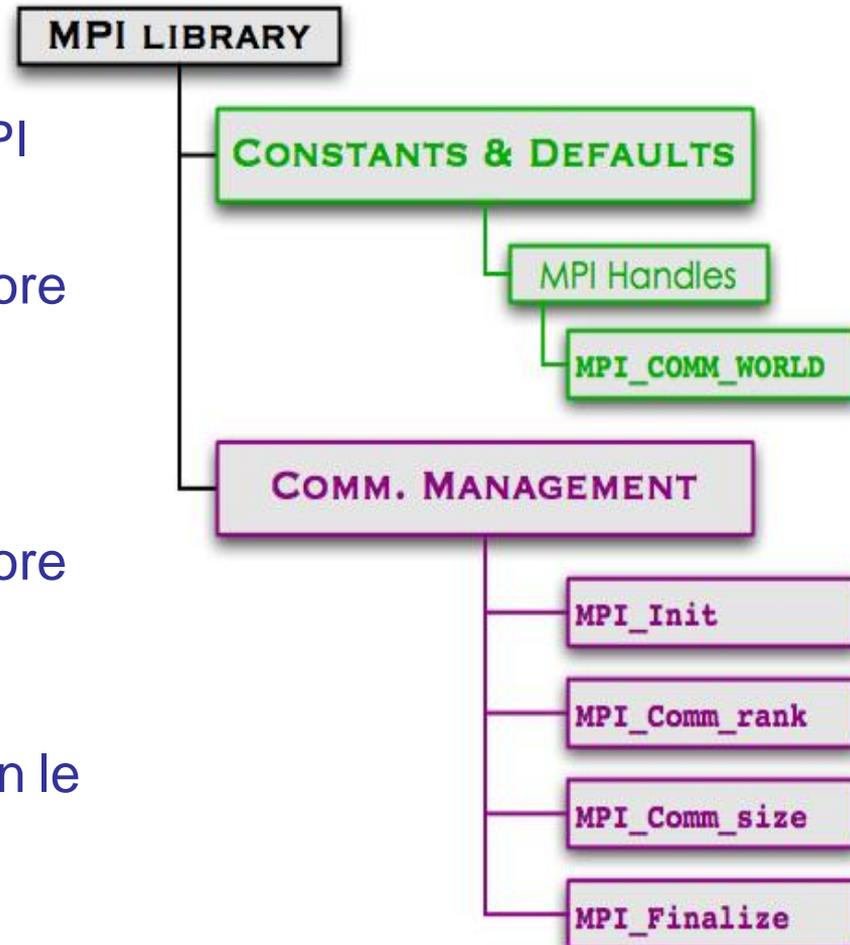
- **size** è di tipo **int** (**INTEGER**) e conterrà la dimensione di **comm**
- **rank** è di tipo **int** (**INTEGER**) e conterrà il *rank* nel comunicatore **comm** del processo chiamante

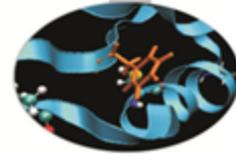
Il primo programma MPI



Operazioni da eseguire:

1. Inizializzare l'ambiente MPI
2. Richiedere al comunicatore di default il *rank* del processo
3. Richiedere al comunicatore di default la sua *size*
4. Stampare una stringa con le informazioni ottenute
5. Chiudere l'ambiente MPI





La versione in C ...

```

#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {

    int myrank, size;

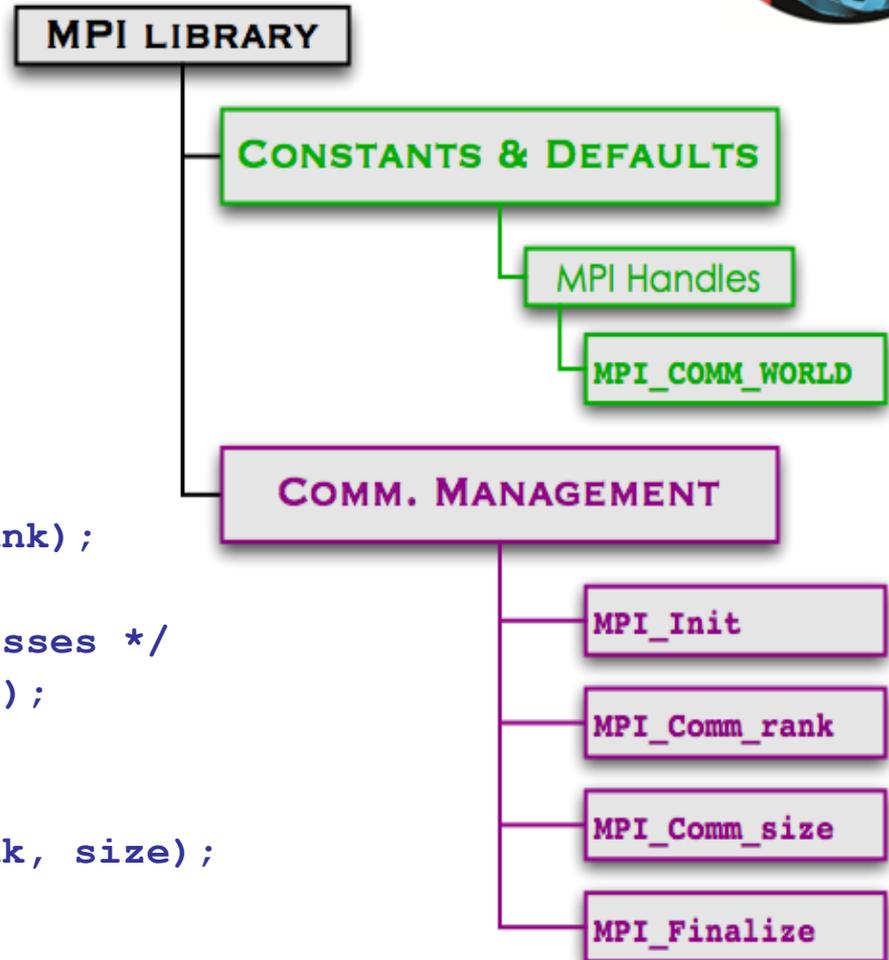
    /* 1. Initialize MPI */
    MPI_Init(&argc, &argv);

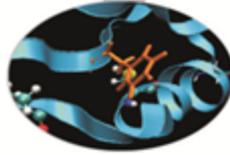
    /* 2. Get my rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    /* 3. Get the total number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* 4. Print myrank and size */
    printf("Process %d of %d \n", myrank, size);

    /* 5. Terminate MPI */
    MPI_Finalize();
}
  
```





.. e quella in Fortran 77

```
PROGRAM hello
```

```
  INCLUDE 'mpif.h'
```

```
  INTEGER myrank, size, ierr
```

```
C 1. Initialize MPI:
```

```
  call MPI_INIT(ierr)
```

```
C 2. Get my rank:
```

```
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
```

```
C 3. Get the total number of processes:
```

```
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
```

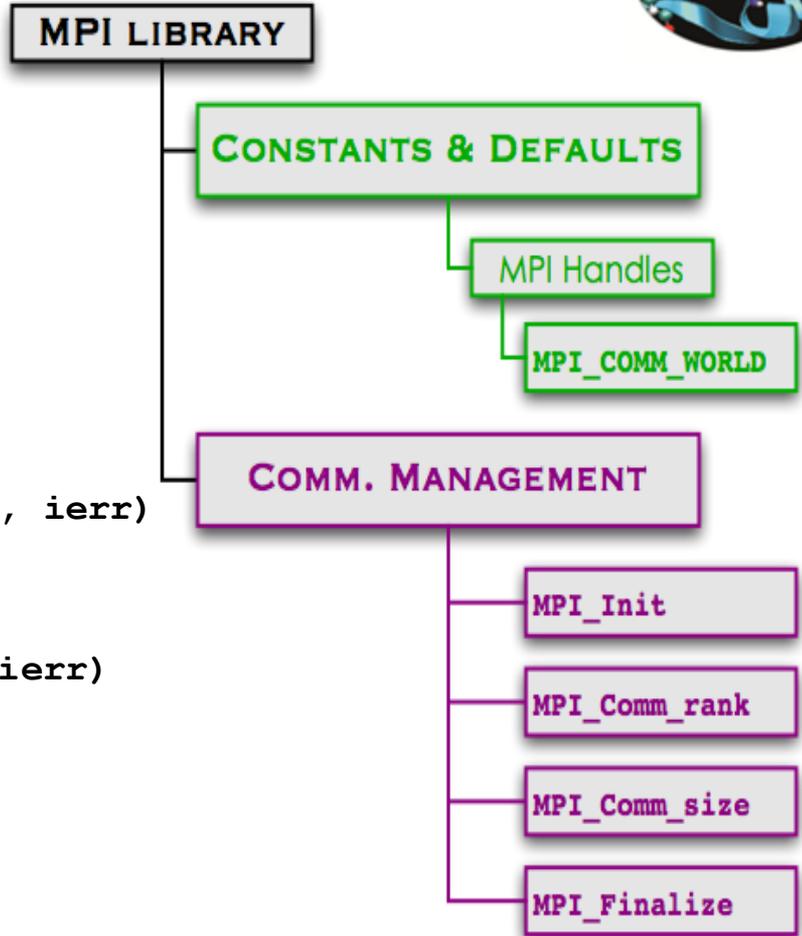
```
C 4. Print myrank and size
```

```
  PRINT *, "Process", myrank, "of", size, "
```

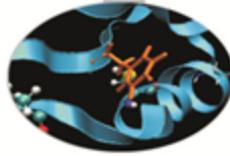
```
C 5. Terminate MPI:
```

```
  call MPI_FINALIZE(ierr)
```

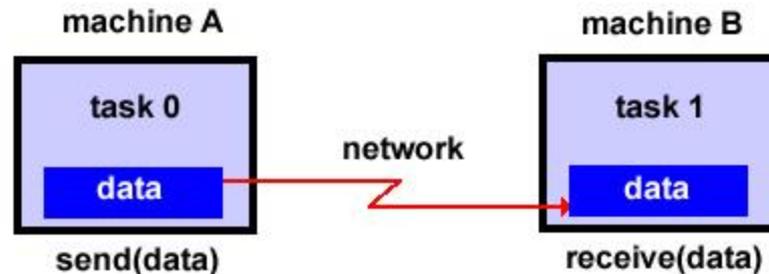
```
END
```



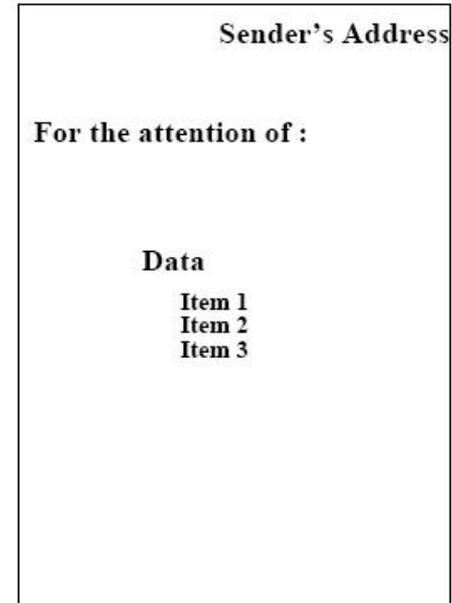
La comunicazione interprocesso



- Nella programmazione parallela **message passing** la cooperazione tra processi avviene attraverso operazioni esplicite di comunicazione interprocesso
- L'operazione elementare di comunicazione è: **point-to-point**
 - vede coinvolti due processi:
 - Il processo *sender* **invia un messaggio**
 - Il processo *receiver* **riceve il messaggio** inviato

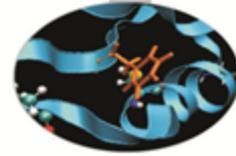


Cos'è un messaggio



- Un messaggio è un blocco di dati da trasferire tra i processi
- È costituito da:
 - Envelope**, che contiene
 - source**: l'identificativo del processo che lo invia
 - destination**: l'identificativo del processo che lo deve ricevere
 - communicator**: l'identificativo del gruppo di processi cui appartengono sorgente e destinazione del messaggio
 - tag**: un identificativo che classifica il messaggio
 - Body**, che contiene
 - buffer**: i dati del messaggio
 - datatype**: il tipo di dati contenuti nel messaggio
 - count**: il numero di occorrenze di tipo *datatype* contenute nel messaggio

I principali *MPI Datatype*



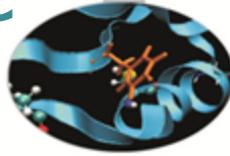
In C

MPI Datatype	C Type
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>

In Fortran

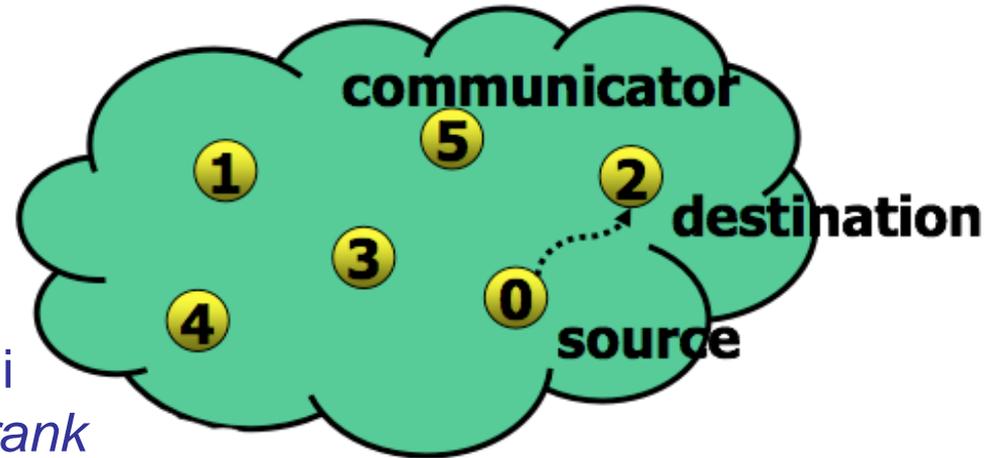
MPI Datatype	Fortran Type
<code>MPI_INTEGER</code>	<code>INTEGER</code>
<code>MPI_REAL</code>	<code>REAL</code>
<code>MPI_DOUBLE_PRECISION</code>	<code>DOUBLE PRECISION</code>
<code>MPI_CHARACTER</code>	<code>CHARACTER(1)</code>
<code>MPI_LOGICAL</code>	<code>LOGICAL</code>

I passi di una comunicazione *point-to-point*

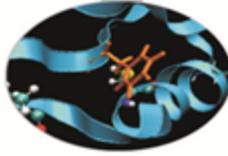


Per inviare/ricevere un messaggio:

- il processo **source** effettua una chiamata ad una funzione MPI, in cui deve essere specificato il *rank* del processo **destination** nel comunicatore
- il processo **destination** deve effettuare una chiamata ad una funzione MPI per ricevere lo specifico messaggio inviato dal **source**

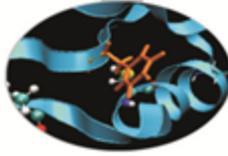


Inviare un messaggio



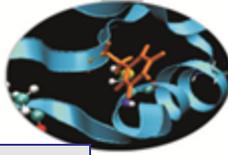
- Il processo **source** effettua una chiamata ad una primitiva con la quale specifica in modo univoco l'*envelope* e il *body* del messaggio da inviare:
 - l'identità della sorgente è implicita (il processo che effettua l'operazione)
 - gli altri elementi che completano la struttura del messaggio
 - identificativo del messaggio
 - identità della destinazione
 - comunicatore da utilizzaresono determinati esplicitamente dagli argomenti che il processo sorgente passa alla funzione di *send*

Ricevere un messaggio



- Il processo destinazione chiama una primitiva, dai cui argomenti è determinato “in maniera univoca” l’*envelope* del messaggio da ricevere
- MPI confronta l’*envelope* del messaggio in ricezione con quelli dell’insieme dei messaggi ancora da ricevere (*pending messages*) e
 - se il messaggio è presente viene ricevuto
 - altrimenti l’operazione non può essere completata fino a che tra i *pending messages* ce ne sia uno con l’*envelope* richiesto
- Il processo di destinazione deve disporre di un’area di memoria sufficiente per salvare il *body* del messaggio

Binding di MPI_Send



In C

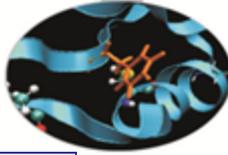
```
int MPI_Send(void *buf,int count,MPI_Datatype dtype,  
            int dest,int tag, MPI_Comm comm)
```

In Fortran

```
MPI_SEND(buf, count, dtype, dest, tag, comm, err)
```

- † Tutti gli argomenti sono di input
 - ‡ **buf** è l'indirizzo iniziale del *send* buffer
 - ‡ **count** è di tipo **int** e contiene il numero di elementi del *send* buffer
 - ‡ **dtype** è di tipo **MPI_Datatype** e descrive il tipo di ogni elemento del *send* buffer
 - ‡ **dest** è di tipo **int** e contiene il *rank* del *receiver* all'interno del comunicatore **comm**
 - ‡ **tag** è di tipo **int** e contiene l'identificativo del messaggio
 - ‡ **comm** è di tipo **MPI_Comm** ed è il comunicatore in cui avviene la *send*

Binding di MPI_Recv



In C

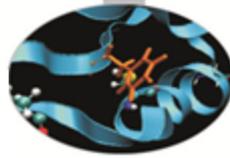
```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,  
            int src, int tag, MPI_Comm comm,  
            MPI_Status *status)
```

In Fortran

```
MPI_RECV(buf, count, dtype, src, tag, comm, status, err)
```

- ⌚ [OUT] **buf** è l'indirizzo iniziale del *receive* buffer
- ⌚ [IN] **count** è di tipo `int` e contiene il numero di elementi del *receive* buffer
- ⌚ [IN] **dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *receive* buffer
- ⌚ [IN] **src** è di tipo `int` e contiene il *rank* del *sender* all'interno del comunicatore `comm`
- ⌚ [IN] **tag** è di tipo `int` e contiene l'identificativo del messaggio
- ⌚ [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore in cui avviene la *send*
- ⌚ [OUT] **status** è di tipo `MPI_Status` (`INTEGER(MPI_STATUS_SIZE)`) e conterrà informazioni sul messaggio che è stato ricevuto

send/receive: intero (C)



```

#include <stdio.h>
#include <mpi.h>

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

    MPI_Status status;
    int rank, size;

    /* data to communicate */
    int data_int;

    /* Start up MPI environment*/
    MPI_Init(&argc, &argv);

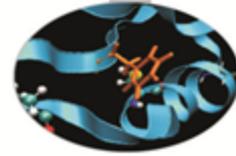
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        data_int = 10;
        MPI_Send(&data_int, 1, MPI_INT, 1, 666, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&data_int, 1, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
        printf("Process 1 receives %d from process 0.\n", data_int);
    }

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

```

send/receive: *array di double (FORTRAN)*



```
program main
```

```

implicit none
include 'mpif.h'
integer ierr, rank, nprocs
integer i, j, status(MPI_STATUS_SIZE)

```

```

c--- data to communicate-----
integer MSIZE
parameter (MSIZE=10)
double precision matrix(MSIZE, MSIZE)

```

```

c--- Start up MPI -----
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

```

```

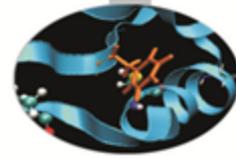
if (rank.eq.0) then
  do i=1,MSIZE
    do j=1,MSIZE
      matrix(i,j)= dble(i+j)
    enddo
  enddo
  CALL MPI_SEND(matrix, MSIZE*MSIZE, MPI_DOUBLE_PRECISION, 1, 666
  $      ,MPI_COMM_WORLD, ierr)
else if (rank.eq.1) then
  CALL MPI_RECV(matrix, MSIZE*MSIZE, MPI_DOUBLE_PRECISION, 0, 666
  $      ,MPI_COMM_WORLD, status, ierr)
  print *,'Proc 1 receives the following matrix from proc 0'
  write (*,'(10(f6.2,2x))') matrix
endif

```

```

call MPI_FINALIZE(ierr)
end

```



send/receive: array di float (C)

```

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

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

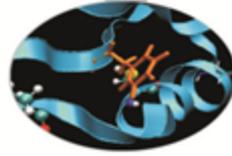
    MPI_Status status;
    int rank, size;
    int i, j;

    /* 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;
}
  
```



send/receive: porzione di array (C)

```

#include <stdio.h>
#include <mpi.h>
#define USIZE 50
#define BORDER 12

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

```

```

  MPI_Status status;
  int indx, rank, nprocs;
  int start_send_buf = BORDER;
  int start_recv_buf = USIZE - BORDER;
  int length = 10;
  int vector[USIZE];

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

```

```

  /* all process initialize vector */
  for (indx = 0; indx < USIZE; indx++) vector[indx] = rank;

```

```

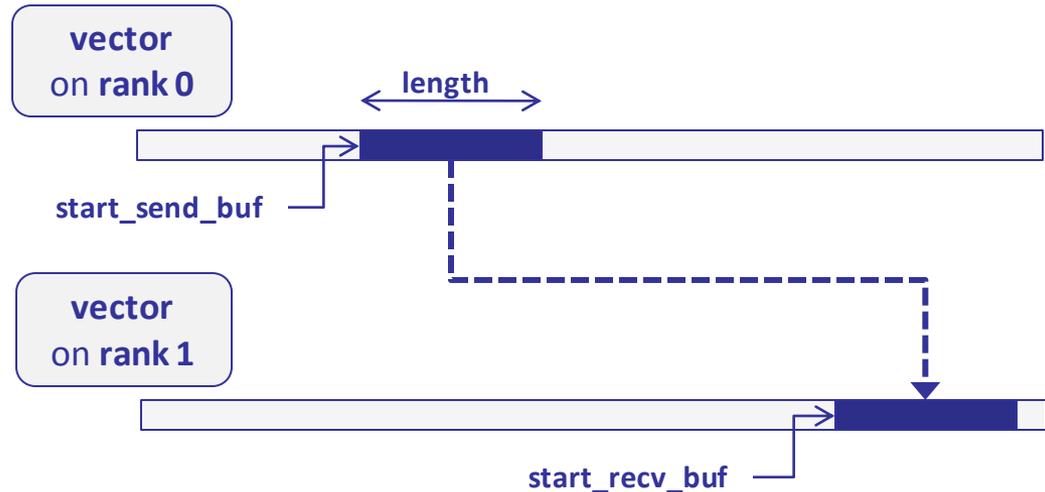
  if (rank == 0) {
    /* send length integers starting from the "start_send_buf"-th position of vector */
    MPI_Send(&vector[start_send_buf], length, MPI_INT, 1, 666, MPI_COMM_WORLD);
  }
  if (rank == 1) {
    /* receive length integers in the "start_recv_buf"-th position of vector */
    MPI_Recv(&vector[start_recv_buf], length, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
  }

```

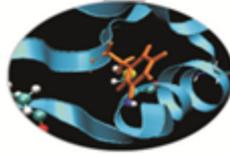
```

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

```



Calcolo parallelo con MPI (1^a parte)



Le sei funzioni di base: introduzione alla comunicazione *point-to-point*

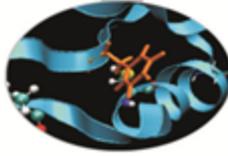
Laboratorio n° 1

Pattern di comunicazione *point-to-point*:
sendrecv

Introduzione alle comunicazioni collettive

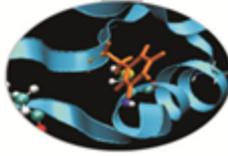
Laboratorio n° 2

Compilare un sorgente MPI



- MPI è una libreria che consiste di due componenti:
 - un archivio di funzioni
 - un *include file* con i prototipi delle funzioni, alcune costanti e *default*
- Per compilare un'applicazione MPI basterà quindi seguire le stesse procedure che seguiamo solitamente per compilare un programma che usi una libreria esterna:
 - Istruire il compilatore sul *path* degli include file (*switch -I*)
 - Istruire il compilatore sul *path* della libreria (*switch -L*)
 - Istruire il compilatore sul nome della libreria (*switch -l*)

Compilare un sorgente MPI (2)



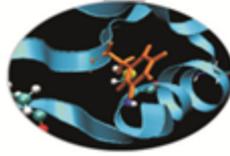
Per compilare il sorgente `sample.f` usando:

- il compilatore `gnu gfortran` (linux)
- le librerie `libmpi_f77.so` e `libmpi.so` che si trovano in `/usr/local/openmpi/lib/`
- gli include file che si trovano in `/usr/local/openmpi/include/`

utilizzeremo il comando:

```
gfortran -I/usr/local/include/ sample.f  
-L/usr/local/openmpi/lib/ -lmpi_f77 -lmpi -o sample.x
```

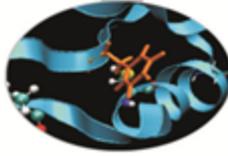
Compilare un sorgente MPI (3)



- † ... ma esiste un modo più comodo
- † Ogni ambiente MPI fornisce un 'compilatore' (basato su uno dei compilatori seriali disponibili) che ha già definiti il giusto set di switch per la compilazione
- † Ad esempio, usando `OpenMPI` (uno degli ambienti MPI più diffusi) per compilare `sample.F`

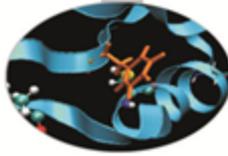
```
mpif90 sample.F -o sample.x
```

Eseguire un programma MPI



- † Per eseguire un programma MPI è necessario lanciare tutti i processi (*process spawn*) con cui si vuole eseguire il calcolo in parallelo
- † Ogni ambiente parallelo mette a disposizione dell'utente un ***MPI launcher***
- † Il *launcher* MPI chiederà tipicamente:
 - ‡ Numero di processi
 - ‡ 'Nome' dei nodi che ospiteranno i processi
 - ‡ Stringa di esecuzione dell'applicazione parallela

Definizione dei processori su cui girare MPI

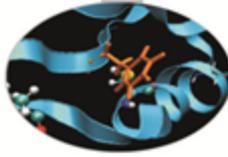


Il nome dei processori che ospiteranno i processi può essere scritto in un file con una specifica sintassi accettata dal *launcher* MPI

- ✦ Nel nostro caso si scrivono di seguito, su righe successive, i nomi delle macchine sulle quali gireranno i processi seguiti dalla *keyword* `slots=XX`, dove `XX` è il numero di processori della macchina
- ✦ Esempio:
volendo girare 6 processi, 2 sulla macchina `node1` e 4 su `node2`, il file `my_hostfile` sarà:

```
node1 slots=2  
node2 slots=4
```

Compilare ed eseguire (OpenMPI)



📌 Compilare:

📌 Fortran F77 o F90 source:

```
(mpif77) mpif90 sample.F90 -o sample.x  
mpifort sample.F90 -o sample.x (da OpenMPI 1.7)
```

📌 C source:

```
mpicc sample.c -o sample.x
```

📌 C++ source:

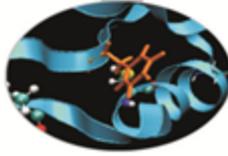
```
mpic++ sample.cpp -o sample.x
```

📌 Eseguire:

📌 il launcher OpenMPI è `mpirun` (oppure `mpiexec`):

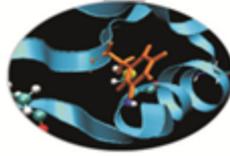
```
mpiexec -hostfile my_hostfile -n 4 ./sample.x
```

Programma della 1° sessione di laboratorio



- Familiarizzare con l'ambiente MPI
 - *Hello World* in MPI (Esercizio 1)
- Esercizi da svolgere
 - *Send/Receive* di un intero e di un *array* di *float* (Esercizio 2)
 - Calcolo di π con il metodo integrale (Esercizio 3)
 - Calcolo di π con il metodo Monte Carlo (Esercizio 4)
 - *Communication Ring* (Esercizio 5)





MPI Hello World

- Come si compila il codice:

- In C:

- ```
mpicc helloworld.c -o hello.x
```

- In Fortran:

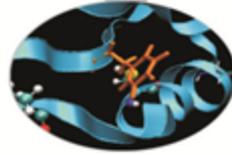
- ```
mpif90 helloworld.f -o hello.x
```

- ```
mpifort helloworld.f -o hello.x
```

- Come si manda in esecuzione utilizzando 4 processi:

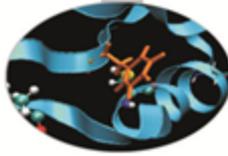
- ```
mpirun -n 4 ./hello.x
```

Send/Receive di un intero e di un array



- Utilizzare tutte le sei funzioni di base della libreria MPI (MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Comm_size, MPI_Send e MPI_Recv)
 - Provare a spedire e a ricevere dati da e in posizioni diverse dall'inizio dell'array
- Il processo con rank 0 inizializza la variabile (intero o array di float) e la spedisce al processo con rank 1
- Il processo con rank 1 riceve i dati spediti dal processo 0 e li stampa
- Provare a vedere cosa succede inviando e ricevendo quantità diverse di dati

Send/Receive di quantità diverse di dati

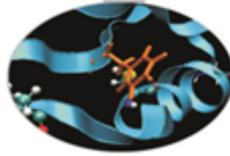


- ☛ Cosa succede se la lunghezza del messaggio ricevuto (`r_count`) è diversa dalla lunghezza del messaggio spedito (`s_count`) ?
 - ☛ Se `s_count < r_count` → solo le prime `s_count` locazioni di `r_buf` sono modificate
 - ☛ Se `s_count > r_count` → errore overflow

Inoltre

- ☛ La lunghezza del messaggio ricevuto (`r_count`) deve essere minore o uguale alla lunghezza del receive buffer (`length_buf`)
 - ☛ Se `r_count < length_buf` → solo le prime `r_count` locazioni di `buf` sono modificate
 - ☛ Se `r_count > length_buf` → errore overflow
- ☛ Per conoscere, al termine di una receive, la lunghezza del messaggio effettivamente ricevuto si può analizzare l'argomento **status**

Argomento status del recv



- struct in C e array of integer di lunghezza MPI_STATUS_SIZE in Fortran
- status contiene direttamente 3 field, più altre informazioni:
 - MPI_TAG
 - MPI_SOURCE
 - MPI_ERROR
- Per conoscere la lunghezza del messaggio ricevuto si utilizza la funzione MPI_GET_COUNT

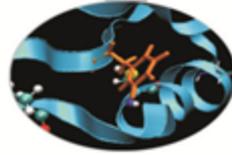
In C

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype  
dtype, int *count)
```

In Fortran

```
MPI_GET_COUNT(status, dtype, count, err)
```

• [IN]: status, dtype [OUT]: count



Calcolo di π con il metodo integrale

- Il valore di π può essere calcolato tramite l'integrale

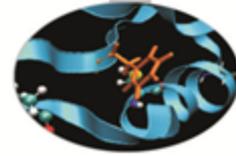
$$\int_0^1 \frac{4}{1+x^2} dx = 4 \cdot \arctan(x) \Big|_0^1 = \pi$$

- In generale, se f è integrabile in $[a,b]$

$$\int_a^b f(x) dx = \lim_{N \rightarrow \infty} \sum_{i=1}^N f_i \cdot h \quad \text{con} \quad f_i = f(a + ih) \quad \text{e} \quad h = \frac{b-a}{N}$$

- Dunque, per N sufficientemente grande

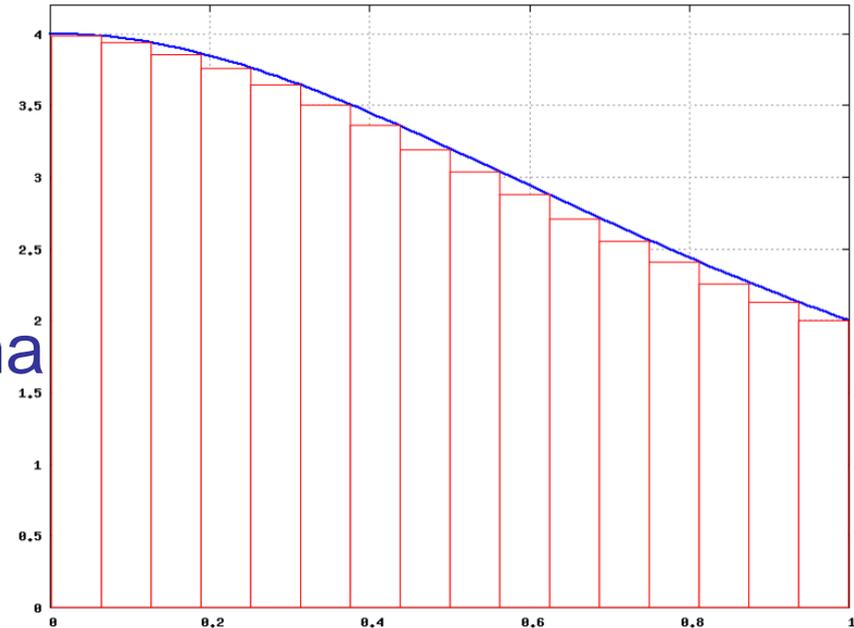
$$\pi \cong \sum_{i=1}^N \frac{4 \cdot h}{1 + (ih)^2} \quad \text{con} \quad h = \frac{1}{N}$$



Calcolo di π in seriale con il metodo integrale

- L'intervallo $[0, 1]$ è diviso in N sotto intervalli, di dimensione $h=1/N$
- L'integrale può essere approssimato con la somma della serie

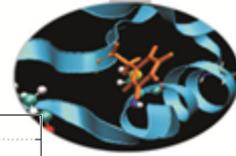
$$\sum_{i=1}^N \frac{4 \cdot h}{1 + (ih)^2} \quad \text{con} \quad h = \frac{1}{N}$$



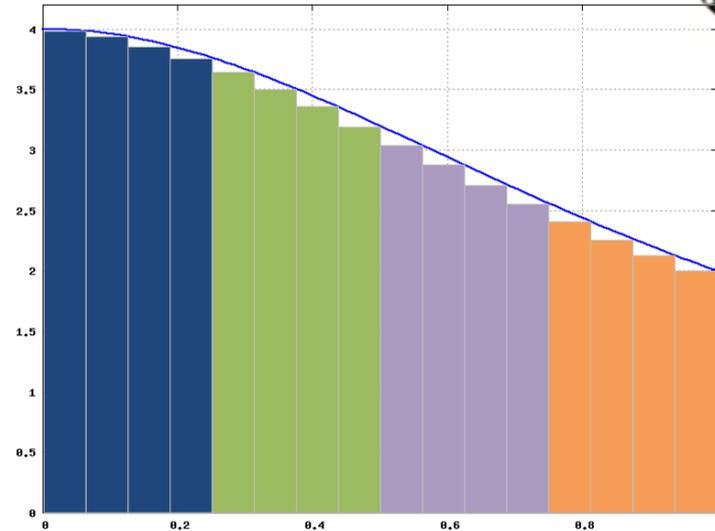
che è uguale alla somma delle aree dei rettangoli in rosso

- Al crescere di N si ottiene una stima sempre più precisa di π

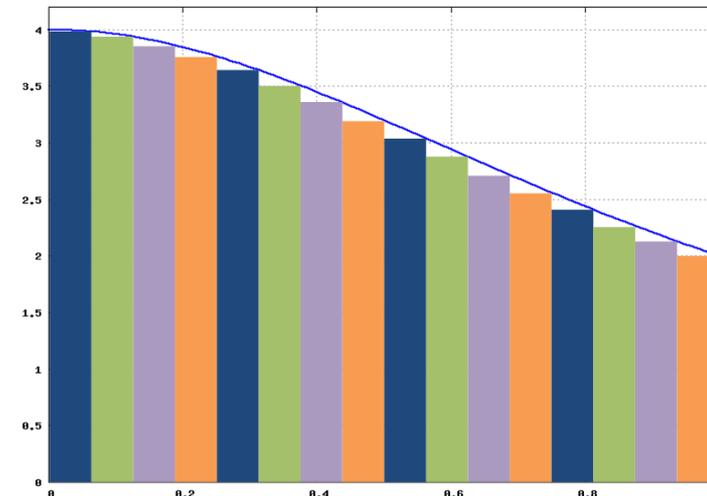
Il Calcolo di π : l'algoritmo parallelo



1. Ogni processo calcola la somma parziale di propria competenza rispetto alla decomposizione scelta
2. Ogni processo con $rank \neq 0$ invia al processo di $rank 0$ la somma parziale calcolata
3. Il processo di $rank 0$
 - ☛ Riceve le $P-1$ somme parziali inviate dagli altri processi
 - ☛ Ricostruisce il valore dell'integrale sommando i contributi ricevuti dagli altri processi con quello calcolato localmente

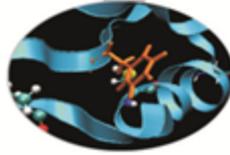


Process 0 Process 1 Process 2 Process 3

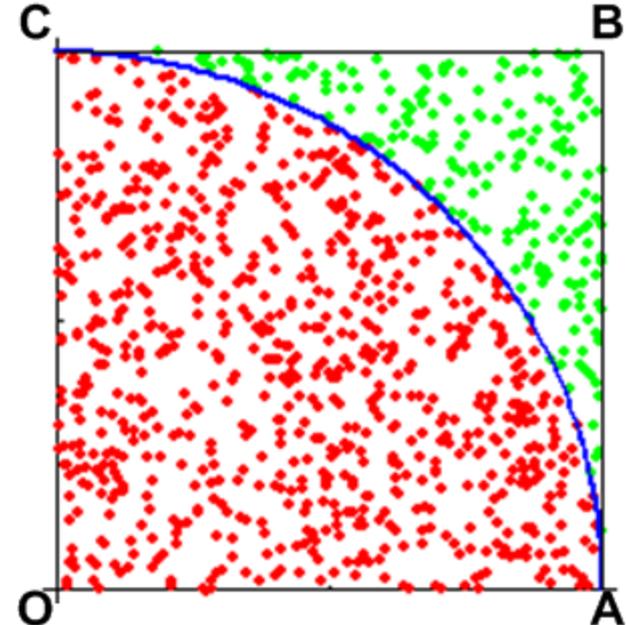


Process 0 Process 1 Process 2 Process 3

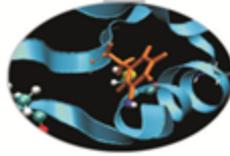
Il Calcolo di π con il metodo Monte Carlo



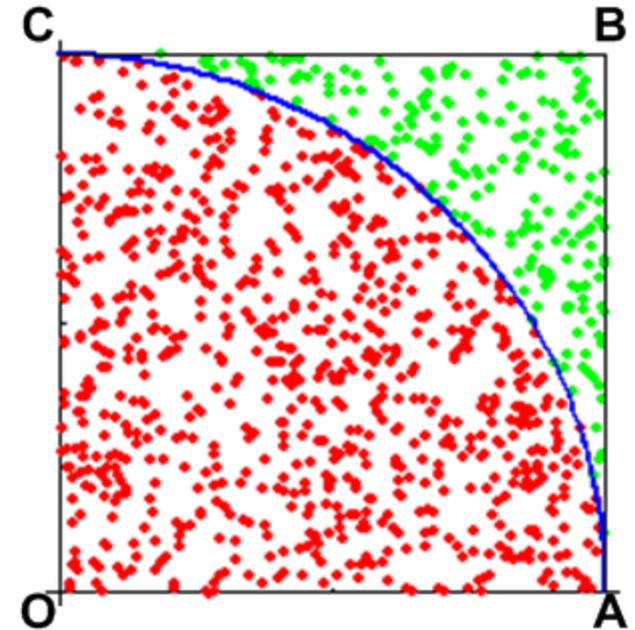
- AOC è il quadrante del cerchio unitario, la cui area è $\pi/4$
- Sia $Q = (x,y)$ una coppia di numeri casuali estratti da una distribuzione uniforme in $[0, 1]$
- La probabilità p che il punto Q sia interno al quadrante AOC è pari al rapporto tra l'area di AOC e quella del quadrato ABCO, ovvero $4p = \pi$
- Con il metodo Monte Carlo possiamo campionare p e dunque stimare il valore di π



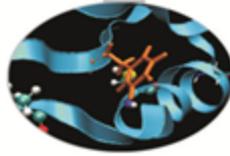
Il Calcolo di π in seriale (Monte Carlo)



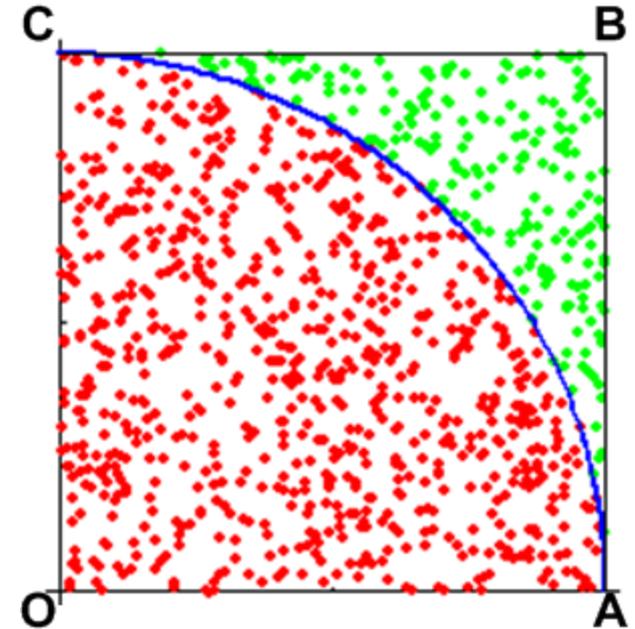
- † Estrarre N coppie $Q_i=(x_i, y_i)$ di numeri pseudo casuali uniformemente distribuiti nell'intervallo $[0, 1]$
- † Per ogni punto Q_i
 - ‡ calcolare $d_i = x_i^2 + y_i^2$
 - ‡ se $d_i \leq 1$ incrementare il valore di N_c , il numero di punti interni al quadrante AOC
- † Il rapporto N_c/N è una stima della probabilità p
- † $4*N_c/N$ è una stima di π , con errore dell'ordine $1/\sqrt{N}$

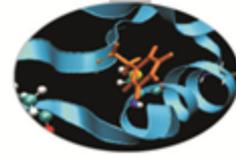


Il Calcolo di π con P processi (Monte Carlo)



1. Ogni processo estrae N/P coppie $Q_i=(x_i,y_i)$ e calcola il relativo numero N_c di punti interni al quadrante AOC
2. Ogni processo con $rank \neq 0$ invia al processo di $rank 0$ il valore calcolato di N_c
3. Il processo di $rank 0$
 - ✦ Riceve i $P-1$ valori di N_c inviati dagli altri processi
 - ✦ Ricostruisce il valore globale di N_c sommando i contributi ricevuti dagli altri processi con quello calcolato localmente
 - ✦ Calcola la stima di π ($= 4*N_c/N$)





Esercizio facoltativo

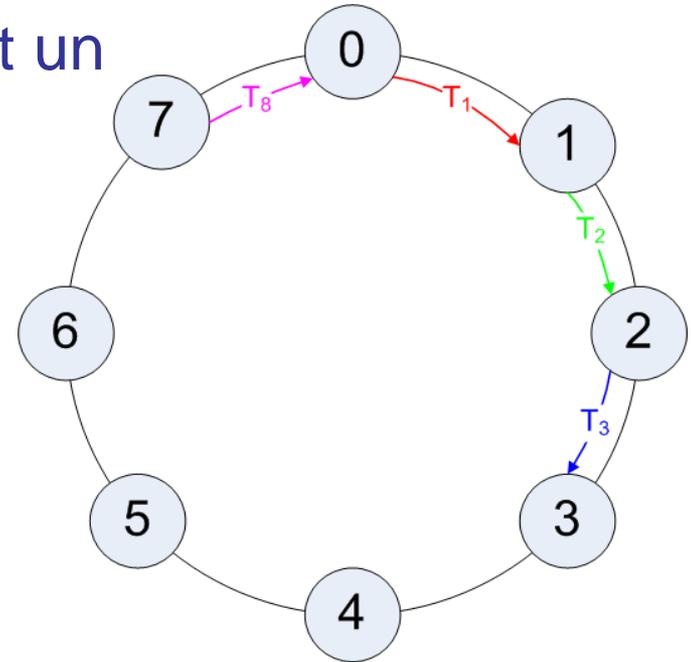
Scrivere un programma MPI in cui

🕒 Il processo 0 legge da standard input un numero intero positivo A

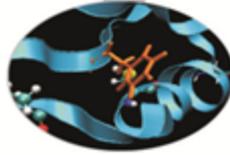
1. All'istante T_1 il processo 0 invia A al processo 1 e il processo 1 lo riceve
2. All'istante T_2 il processo 1 invia A al processo 2 e il processo 2 lo riceve
3.
4. All'istante T_N il processo $N-1$ invia A al processo 0 e il processo 0 lo riceve

🕒 Il processo 0

- 🕒 decrementa e stampa il valore di A
- 🕒 se A è ancora positivo torna al punto 1, altrimenti termina l'esecuzione



Calcolo parallelo con MPI (1^a parte)



Le sei funzioni di base: introduzione alla comunicazione *point-to-point*

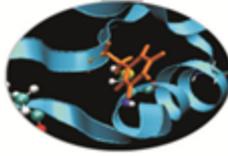
Laboratorio n° 1

Pattern di comunicazione *point-to-point*:
sendrecv

Introduzione alle comunicazioni collettive

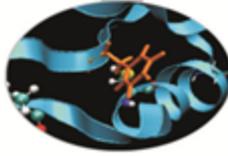
Laboratorio n° 2

Cosa abbiamo imparato di MPI



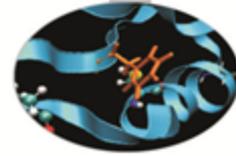
- Concetto di “aritmetica parallela”:
 - gruppo di collaboratori che lavorano indipendentemente su parti del problema
 - sulla base dei risultati parziali, si ottiene il risultato complessivo: questa fase richiede la comunicazione tra collaboratori
- MPI come strumento per implementare la comunicazione 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

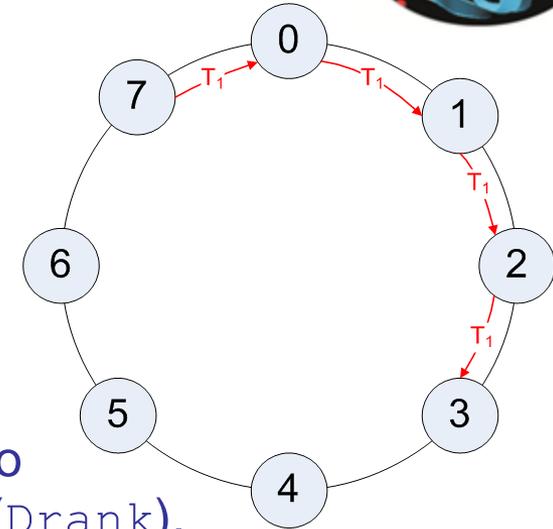


- † Nella parallelizzazione di programmi reali, alcuni schemi di invio/ricezione del messaggio sono largamente diffusi:
pattern di comunicazione
- † I pattern di comunicazione possono essere di tipo
 - ‡ *point-to-point*, coinvolgono solo due processi
 - ‡ collettivi, coinvolgono più processi
- † MPI 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**



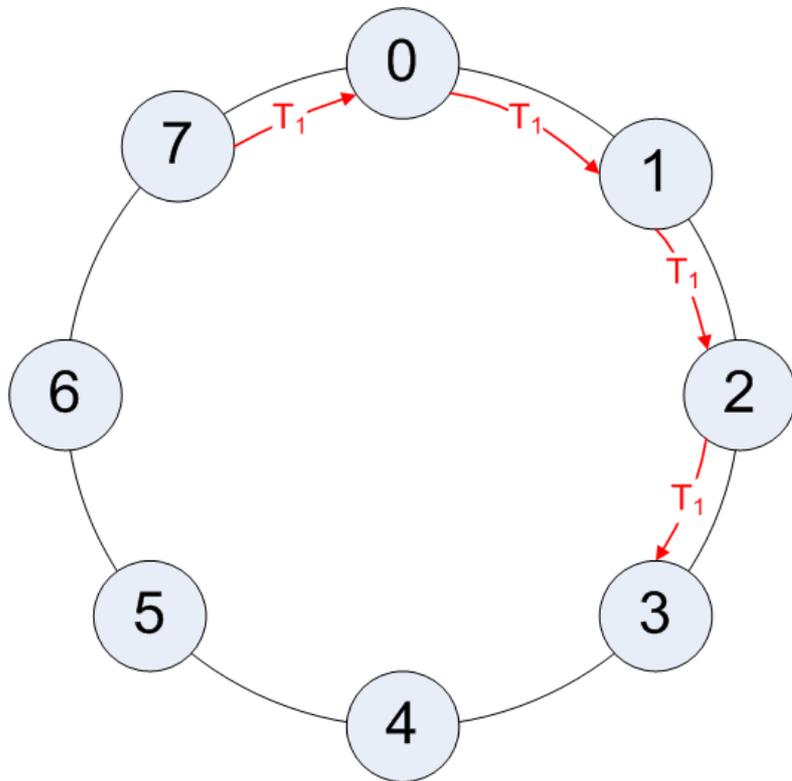
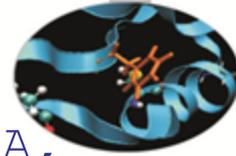
- Lo **shift** è un pattern *point-to-point*.

- Ogni processo invia/riceve un set di dati in un verso (positivo/negativo) con una certa distanza di *rank* (D_{rank}).

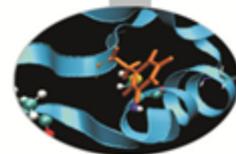
Per esempio:

- se $D_{rank}=1$ con verso positivo: il processo i comunica al processo $i+1$
- se $D_{rank}=3$ con verso positivo: il processo i comunica al processo $i+3$
- se $D_{rank}=1$ con verso negativo: il processo i comunica al processo $i-1$
- Se lo **shift** è **periodico**:
 - Il processo con $rank=size-D_{rank}$ 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 500, 1000 e 2000 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;

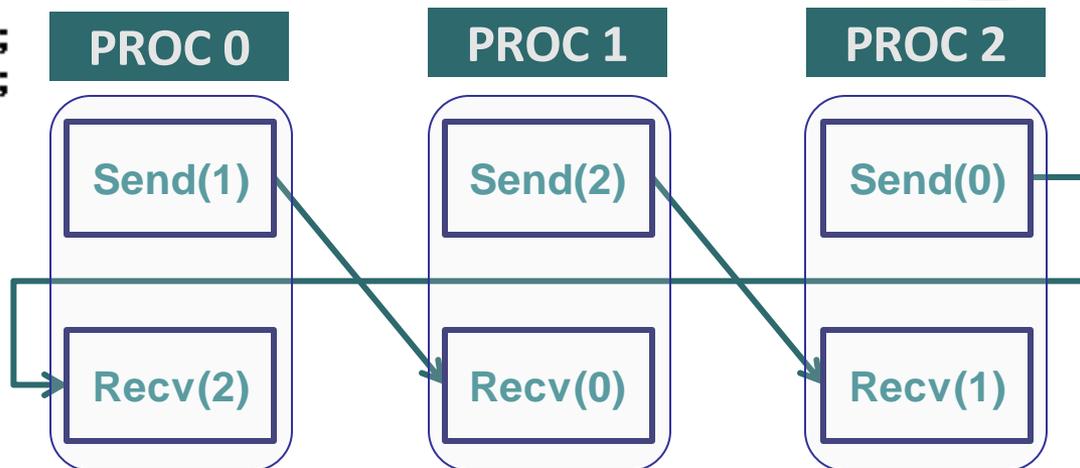
/* starting send of array A */
MPI_Send(A, MSIZE, MPI_INT, to, tag, MPI_COMM_WORLD);
printf("Proc %d sends %d integers to proc %d\n",
       rank, MSIZE, to);

/* 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);

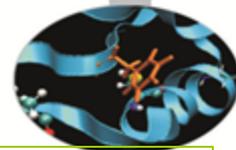
/* print first content of arrays A and B */
printf("Proc %d has A[0] = %d, B[0] = %d\n\n", rank, A[0], B[0]);

/* 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);

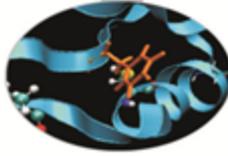
/* print first content of arrays A and B */
printf("Proc %d has A[0] = %d, B[0] = %d\n\n", rank, A[0], B[0]);

/* Quit */
MPI_Finalize();

```

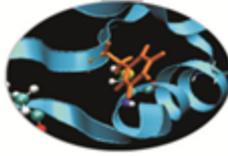
- Cosa succede girando l'esempio al crescere del valore di **MSIZE**?
- Utilizzando l'ambiente parallelo OpenMPI sul nostro cluster, il programma funziona correttamente a **MSIZE = 1000**
- Se **MSIZE = 2000**, il programma va in *hang*

Il *deadlock*



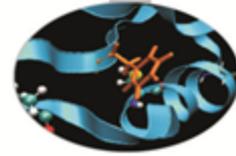
- † L'implementazione *naive* dello *shift* circolare non è corretta: per $MSIZE > 1000$ si genera un *deadlock*
- † Il *deadlock* è la condizione in cui ogni processo è in attesa di un altro per terminare la comunicazione e procedere poi nell'esecuzione del programma
- † Per comprendere perché il *deadlock* si verifica per $MSIZE > 1000$ è necessario entrare nel dettaglio del meccanismo di scambio di messaggi tra due processi

Cenni sul meccanismo di comunicazione



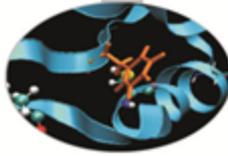
- † Le funzioni send standard di MPI non ‘ritornano’ sino a che l’invio del messaggio non sia stato completato secondo una delle due modalità seguenti:
 - † *Buffered*: l’invio del messaggio avviene attraverso una copia dal buffer di invio in un buffer di sistema
 - † *Synchronous*: l’invio del messaggio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

Cosa fa MPI_Send nel nostro esempio



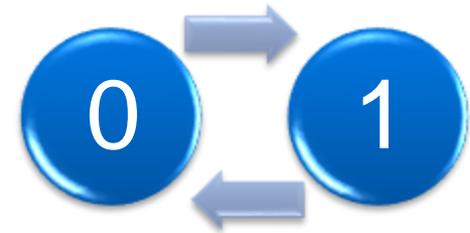
- † La modalità di completamento di MPI_Send varia a seconda della *size* dei dati da inviare:
 - ‡ *Buffered* per *size* piccole
 - ‡ *Synchronous* per *size* grandi
- † Nel nostro caso, sino a 1000 elementi di tipo MPI_INT la MPI_Send si comporta come *buffered*, mentre a 2000 si comporta come *synchronous*
 - ‡ per MSIZE=1000 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=2000 il processo può uscire fuori dalla *send* solo quando ci sia in esecuzione un'operazione di *receive* pronta ad accogliere A

Perché il deadlock?



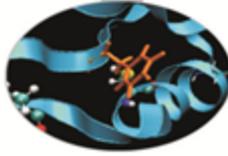
- Nel nostro caso l'algoritmo è del tipo

```
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
```



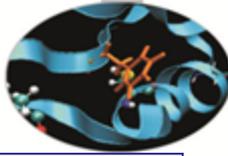
- Per MSIZE=2000, ci sono due *send* in attesa di due *receive*, ma le *receive* potranno essere eseguite solo dopo che le reciproche *send* siano state completate.
- DEADLOCK!

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

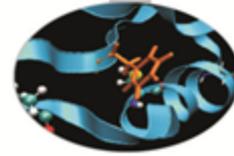
```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype s_dtype,  
int dest,int stag,void *dbuf,int dcount,MPI_Datatype d_type,  
int src,int dtag,MPI_Comm comm,MPI_Status *status)
```

In Fortran

```
MPI_SENDRECV(SBUF, SCOUNT, S_DTYPE, DEST, STAG,  
DBUF, DCOUNT, D_DTYPE, SRC, DTAG,  
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] **dtag** è l'identificativo del *receive message*

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);

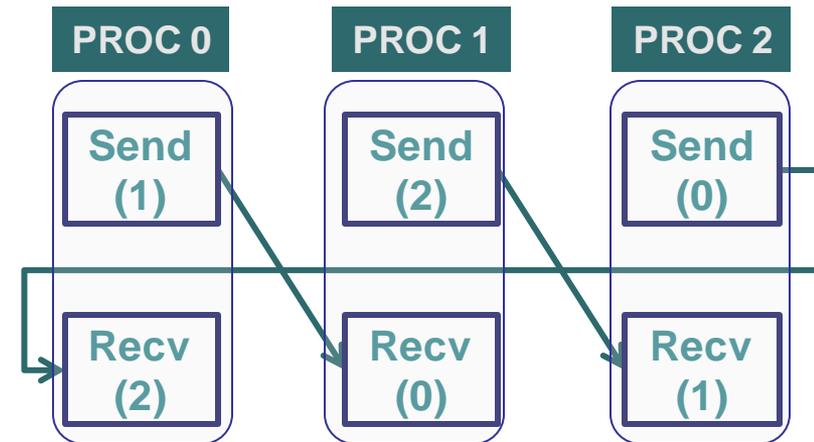
    to = (rank + 1) % size;
    from = (rank + size - 1) % size;

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

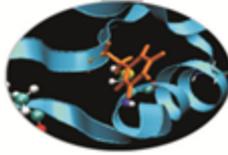
    MPI_Sendrecv(A, MSIZE, MPI_INT, to, 201, /* sending info */
                B, MSIZE, MPI_INT, from, 201, /* recving info */
                MPI_COMM_WORLD, &status);

    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;
}
  
```



Calcolo parallelo con MPI (1^a parte)



Le sei funzioni di base: introduzione alla comunicazione *point-to-point*

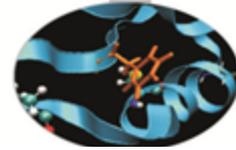
Laboratorio n° 1

Pattern di comunicazione *point-to-point*:
sendrecv

Introduzione alle comunicazioni collettive

Laboratorio n° 2

Introduzione alle comunicazioni collettive



Alcuni pattern di comunicazione prevedono il coinvolgimento di tutti i processi di un comunicatore

Esempio: il calcolo della somma dei contributi parziali dell'integrale per il calcolo del π

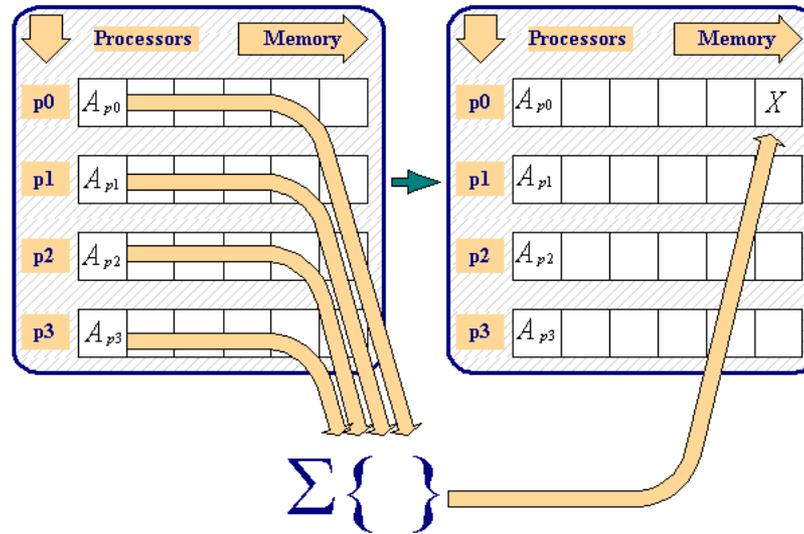
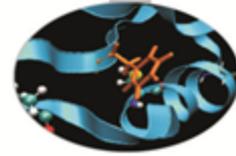
```
if (rank != 0) {  
    /* slave processes send partial pi sum to master process 0 */  
    MPI_Send(&pi, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);  
} else {  
    for (from = 1; from < size; from++) {  
        printf("I have pi = %f\n", pi);  
        /* master process receives partial pi sum from other processes */  
        MPI_Recv(&sum, 1, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &status);  
        printf(".. received %f from proc %d\n", sum, from);  
        pi = pi + sum;  
        fflush(stdout);  
    }  
}
```

MPI mette a disposizione alcune funzioni che implementano questi pattern

- Si evita così al programmatore l'onere e la complicazione di dover programmare questi pattern a partire da comunicazioni *point-to-point*
- Sono implementati con gli algoritmi più efficaci

È possibile catalogare queste funzioni, sulla base del/dei *sender* e del/dei *receiver*, in tre classi: ***all-to-one***, ***one-to-all***, ***all-to-all***. La divisione in classi ci permette di trovare facilmente la funzione cercata

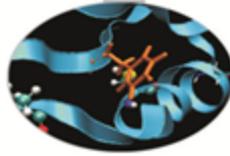
REDUCE



- 📌 L'operazione di *REDUCE* consente di:
 - 📌 Raccogliere da ogni processo i dati provenienti dal *send buffer*
 - 📌 Ridurre i dati ad un solo valore attraverso un operatore (la somma in figura)
 - 📌 Salvare il risultato nel *receive buffer* del processo di destinazione, chiamato convenzionalmente *root* (p0 in figura)

- 📌 Appartiene alla classe *all-to-one*

Binding di MPI_Reduce



In C

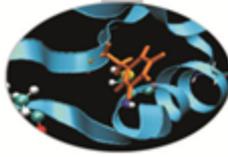
```
int MPI_Reduce(void* sbuf, void* rbuf, int count,  
MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
```

In Fortran

```
MPI_REDUCE(SBUF, RBUF, COUNT, DTYPE, OP, ROOT, COMM, ERR)
```

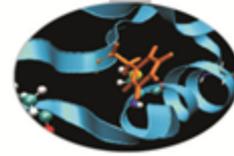
- ⌚ [IN] **sbuf** è l'indirizzo del *send* buffer
- ⌚ [OUT] **rbuf** è l'indirizzo del *receive* buffer
- ⌚ [IN] **count** è di tipo `int` e contiene il numero di elementi del *send/receive* buffer
- ⌚ [IN] **dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *send/receive* buffer
- ⌚ [IN] **op** è di tipo `MPI_Op` e riferenzia l'operatore di reduce da utilizzare
- ⌚ [IN] **root** è di tipo `int` e contiene il *rank* del processo *root* della reduce
- ⌚ [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nella reduce

Operatori di *Reduce*



- † Le principali operazioni di reduce predefinite sono
 - ‡ Massimo (MPI_MAX)
 - ‡ Minimo (MPI_MIN)
 - ‡ Somma (MPI_SUM)
 - ‡ Prodotto (MPI_PROD)
 - ‡ operazioni logiche (MPI_BAND, MPI_LOR, MPI_LXOR)
 - ‡ operazioni *bitwise* (MPI_BAND, MPI_BOR, MPI_BXOR)
- † Gli operatori di reduce sono associativi e commutativi (almeno nella versione a precisione infinita)
- † L'utente può definire operatori *ad-hoc* (MPI_Op_create)

Calcolo di π con *reduce*



```

#include <stdio.h>
#include "mpi.h"
#define INTERVALS 10000

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

    int rank, nprocs, tag;
    int i;
    int interval = INTERVALS;
    double x, dx, f, sum, pi;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

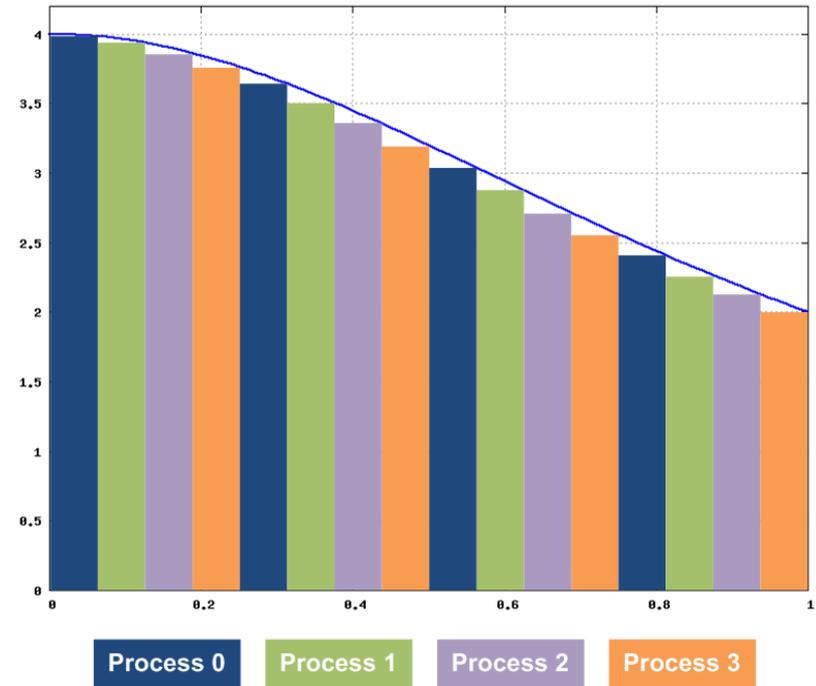
    sum = 0.0; dx = 1.0 / (double) interval;

    /* each process computes integral */
    for (i = rank; i < interval; i = i+nprocs) {
        x = dx * ((double) (i - 0.5));
        f = 4.0 / (1.0 + x*x);
        sum = sum + f;
    }
    pi = dx*sum;
    sum = pi; /* using variable sum as sending buffer */

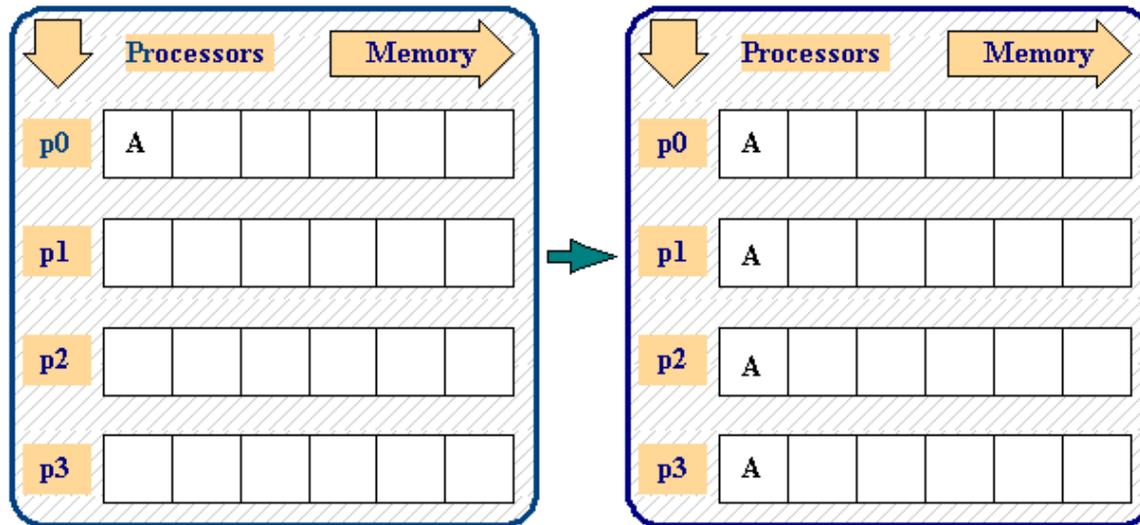
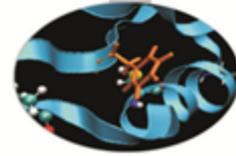
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("Computed PI %.24f\n", pi);

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

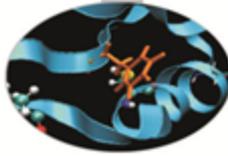


BROADCAST



- La funzionalità di *BROADCAST* consente di copiare dati dal *send* buffer del processo *root* (p0 nella figura) al *receive* buffer di tutti gli altri processi appartenenti al comunicatore utilizzato (processo *root* incluso)
- Appartiene alla classe *one-to-all*

Binding di MPI_Bcast



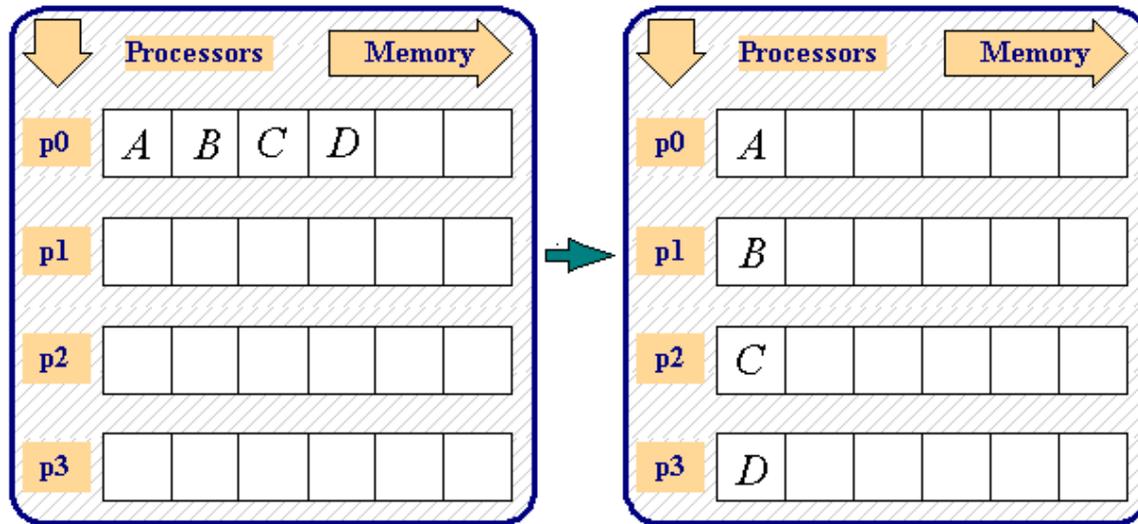
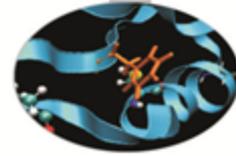
In C

```
int MPI_Bcast(void* buf, int count, MPI_Datatype dtype,  
             int root, MPI_Comm comm)
```

In Fortran

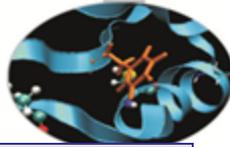
```
MPI_BCAST(BUF, COUNT, DTYPE, ROOT, COMM, ERR)
```

- 📍 [IN/OUT] **buf** è l'indirizzo del *send/receive buffer*
- 📍 [IN] **count** è di tipo `int` e contiene il numero di elementi del *buffer*
- 📍 [IN] **dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *buffer*
- 📍 [IN] **root** è di tipo `int` e contiene il *rank* del processo *root* dell'operazione di *broadcast*
- 📍 [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nell'operazione di *broadcast*



- Il processo *root* (p0 nella figura)
 - divide in N parti uguali un insieme di dati contigui in memoria
 - invia una parte ad ogni processo in ordine di *rank*
- Appartiene alla classe *one-to-all*

Binding di MPI_Scatter



In C

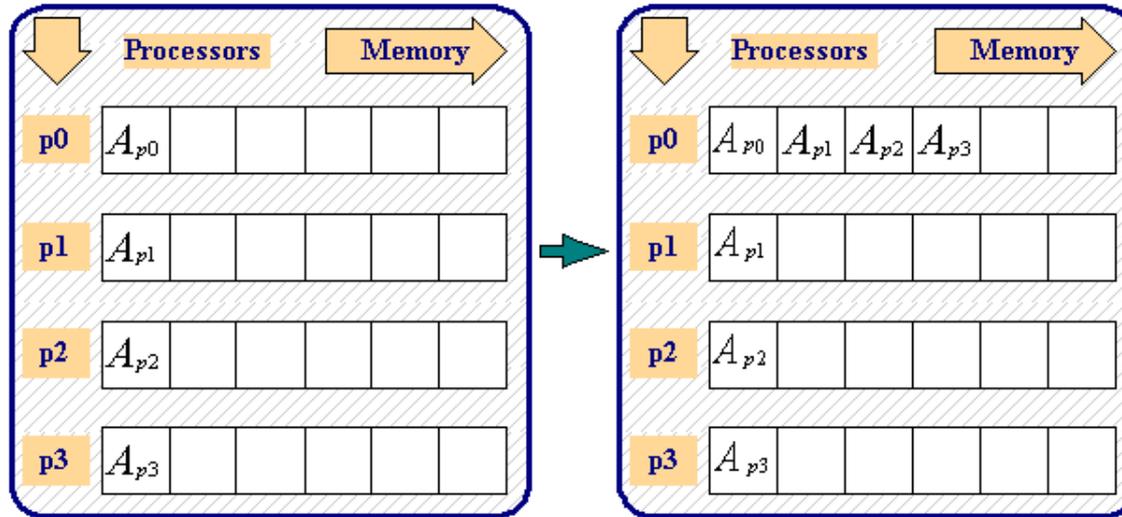
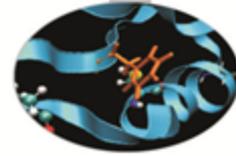
```
int MPI_Scatter(void* sbuf, int scount, MPI_Datatype s_dtype, void*
               rbuf, int rcount, MPI_Datatype r_dtype, int root, MPI_Comm comm)
```

In Fortran

```
MPI_SCATTER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE, ROOT,
            COMM, ERR)
```

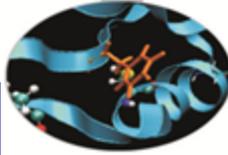
- 📍 [IN] **sbuf** è l'indirizzo del send buffer
- 📍 [IN] **scount**, di tipo `int`, contiene il numero di elementi spediti ad ogni processo
- 📍 [IN] **s_dtype**, di tipo `MPI_Datatype`, descrive il tipo di ogni elemento del *send buffer*
- 📍 [OUT] **rbuf** è l'indirizzo del *receive buffer*
- 📍 [IN] **rcount**, di tipo `int`, contiene il numero di elementi del *receive buffer*
- 📍 [IN] **r_dtype**, di tipo `MPI_Datatype`, descrive il tipo di ogni elemento del *receive buffer*
- 📍 [IN] **root**, di tipo `int`, contiene il *rank* del processo *root* della *scatter*
- 📍 [IN] **comm**, di tipo `MPI_Comm`, è il comunicatore cui appartengono i processi coinvolti nella *scatter*

GATHER



- Con la funzionalità *GATHER* ogni processo (incluso il *root*) invia il contenuto del proprio *send* buffer al processo *root*
- Il processo *root* riceve i dati e li ordina in funzione del *rank* del processo *sender*
- È l'inverso dell'operazione di *scatter*
- Appartiene alla classe *all-to-one*

Binding di MPI_Gather



```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype s_dtype,  
              void* rbuf, int rcount, MPI_Datatype r_dtype,  
              int root, MPI_Comm comm)
```

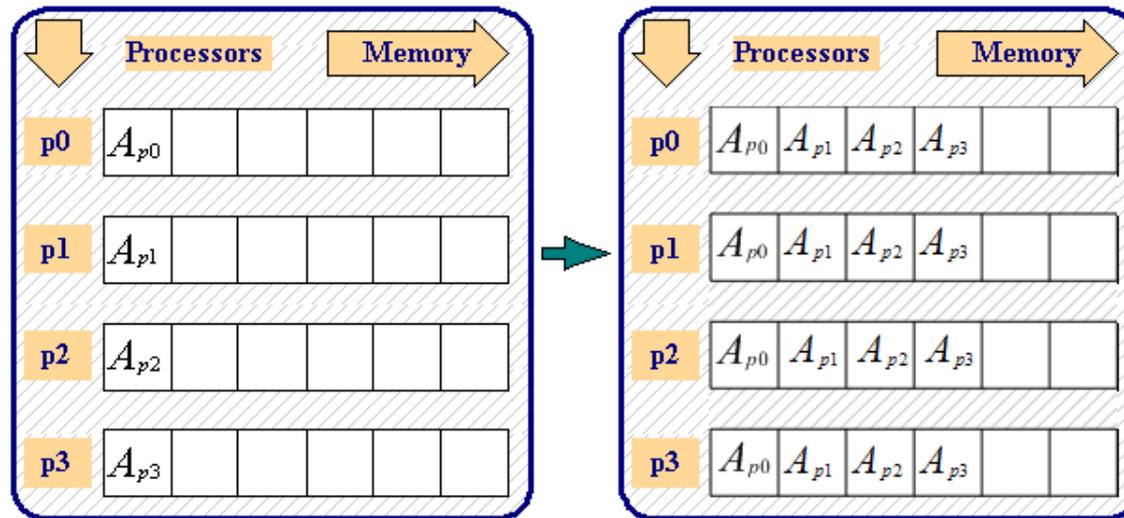
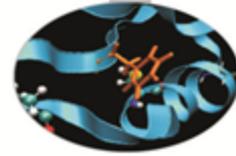
In C

```
MPI_GATHER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE,  
          ROOT, COMM, ERR)
```

In Fortran

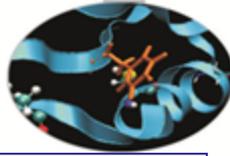
- ⌚ [IN] **sbuf** è l'indirizzo del *send buffer*
- ⌚ [IN] **scount** (*int*) contiene il numero di elementi del *send buffer*
- ⌚ [IN] **s_dtype** (tipo *MPI_Datatype*) descrive il tipo di ogni elemento del *sbuf*
- ⌚ [OUT] **rbuf** è l'indirizzo del *receive buffer*
- ⌚ [IN] **rcount** (*int*) contiene il numero di elementi ricevuti da ogni processo
- ⌚ [IN] **r_dtype** (tipo *MPI_Datatype*) descrive il tipo di ogni elemento del *rbuf*
- ⌚ [IN] **root** (*int*) contiene il *rank* del processo *root* della *gather*
- ⌚ [IN] **comm** (tipo *MPI_Comm*) è il comunicatore cui appartengono i processi coinvolti nella *gather*

ALLGATHER



- Di fatto è l'equivalente di un'operazione di *GATHER*, in cui il processo *root* dopo esegue una *BROADCAST*
- È molto più conveniente ed efficiente eseguire una operazione *ALLGATHER* piuttosto che la sequenza *GATHER+BROADCAST*
- Appartiene alla classe *all-to-all*

Binding di MPI_ALLGather



In C

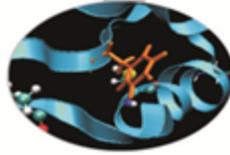
```
int MPI_Allgather(void* sbuf, int scount, MPI_Datatype s_dtype,  
                void* rbuf, int rcount, MPI_Datatype r_dtype, MPI_Comm comm)
```

In Fortran

```
MPI_ALLGATHER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE, COMM,  
             ERR)
```

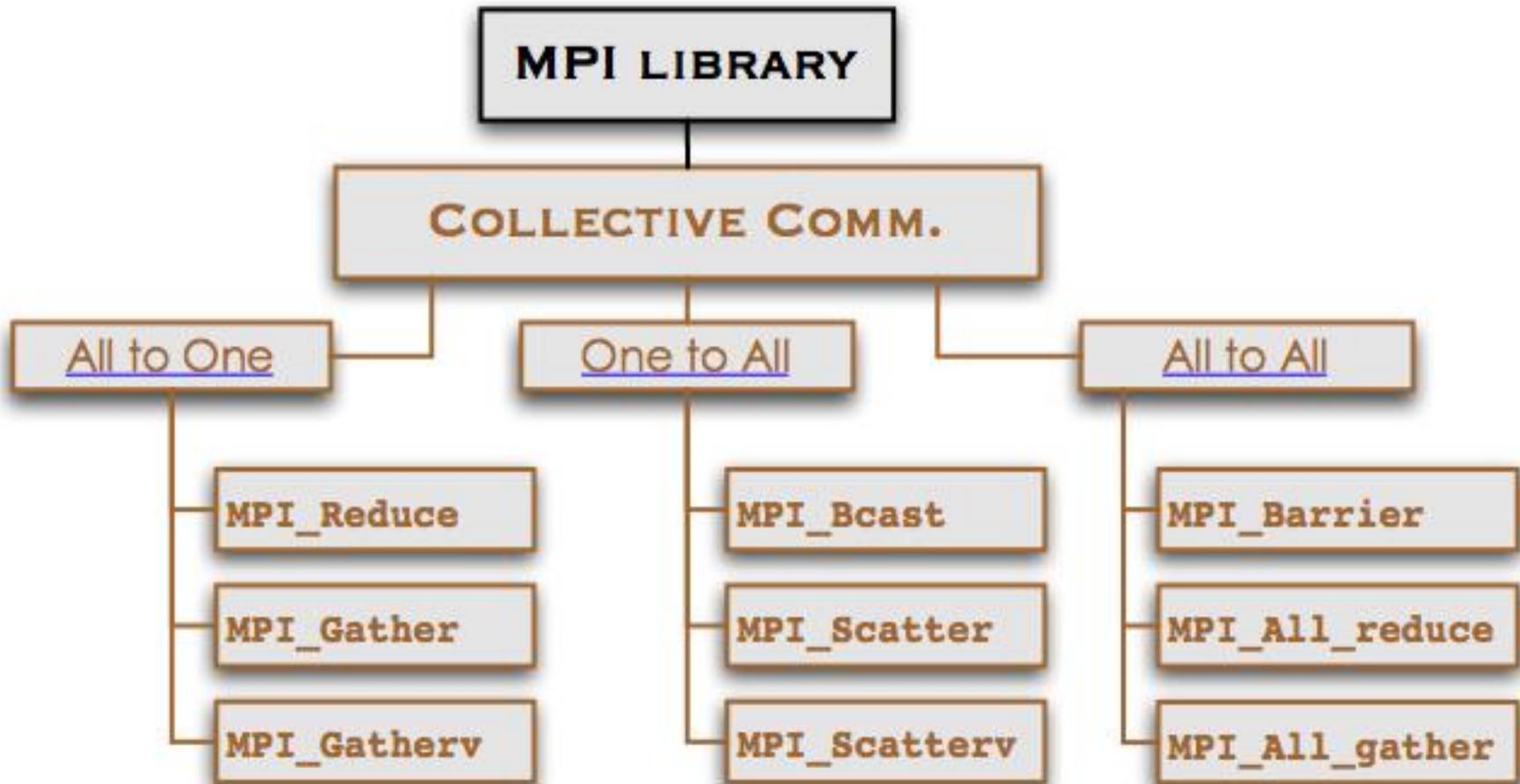
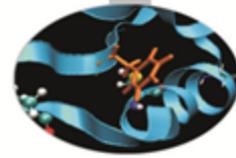
- ⌚ [IN] **sbuf** è l'indirizzo del *send buffer*
- ⌚ [IN] **scount** è di tipo `int` e contiene il numero di elementi del *send buffer*
- ⌚ [IN] **s_dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *send buffer*
- ⌚ [OUT] **rbuf** è l'indirizzo del *receive buffer*
- ⌚ [IN] **rcount** è di tipo `int` e contiene il numero di elementi del *receive buffer*
- ⌚ [IN] **r_dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *receive buffer*
- ⌚ [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nella *ALLGather*

Altre comunicazioni collettive

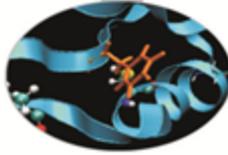


- **MPI_BARRIER:** L'esecuzione di ogni processo appartenente allo stesso comunicatore viene messa in pausa fino a quando tutti i processi non sono giunti a questa istruzione
- **MPI_ALL_REDUCE:** Il risultato della REDUCE viene comunicato a tutti i processi. È equivalente ad una REDUCE seguita da un BROADCAST
- **MPI_SCATTERV** e **MPI_GATHERV:** come SCATTER e GATHER, ma consentono di comunicare blocchi di dati di dimensione diversa

Overview: principali comunicazioni collettive



Calcolo parallelo con MPI (1^a parte)



Le sei funzioni di base: introduzione alla
comunicazione *point-to-point*

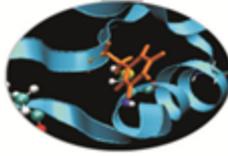
Laboratorio n° 1

Pattern di comunicazione *point-to-point*:
sendrecv

Introduzione alle comunicazioni collettive

Laboratorio n° 2

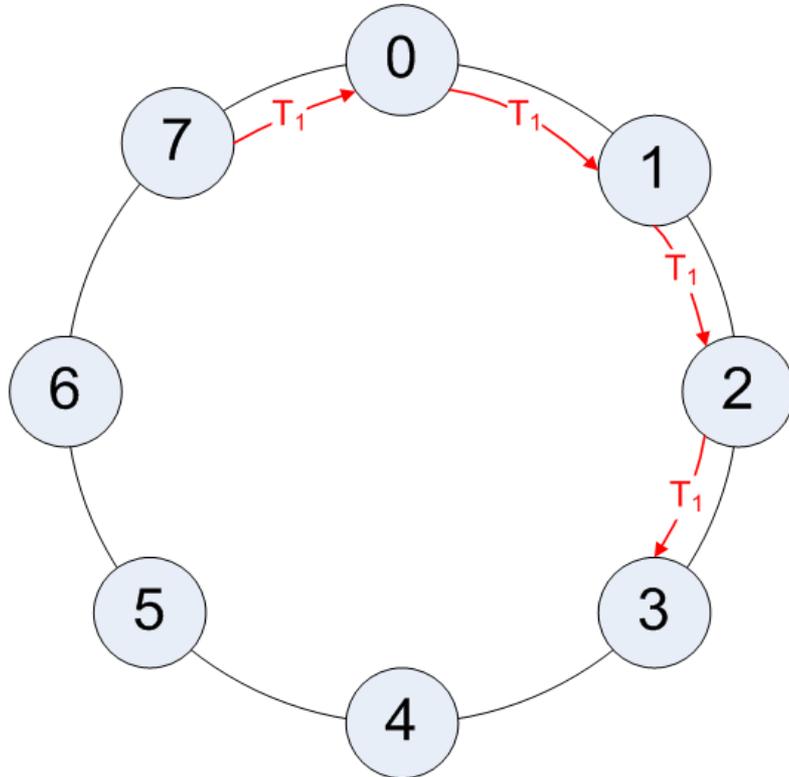
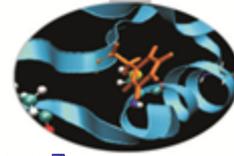
Programma della 2° sessione di laboratorio



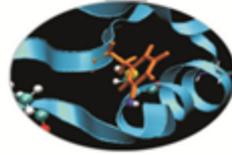
- † Funzioni di comunicazione standard e pattern di comunicazione
 - ‡ *Shift* circolare con `MPI_Sendrecv` (Esercizio 7)
 - ‡ *Array Smoothing* (Esercizio 8)
- † Utilizzare le funzioni collettive per implementare pattern di comunicazione standard
 - ‡ Calcolo di π con comunicazioni collettive (Eserc. 9)
 - ‡ Prodotto matrice-vettore (Esercizio 10)
 - ‡ Prodotto matrice-matrice (Esercizio 11)



Shift Circolare periodico con MPI_Sendrecv

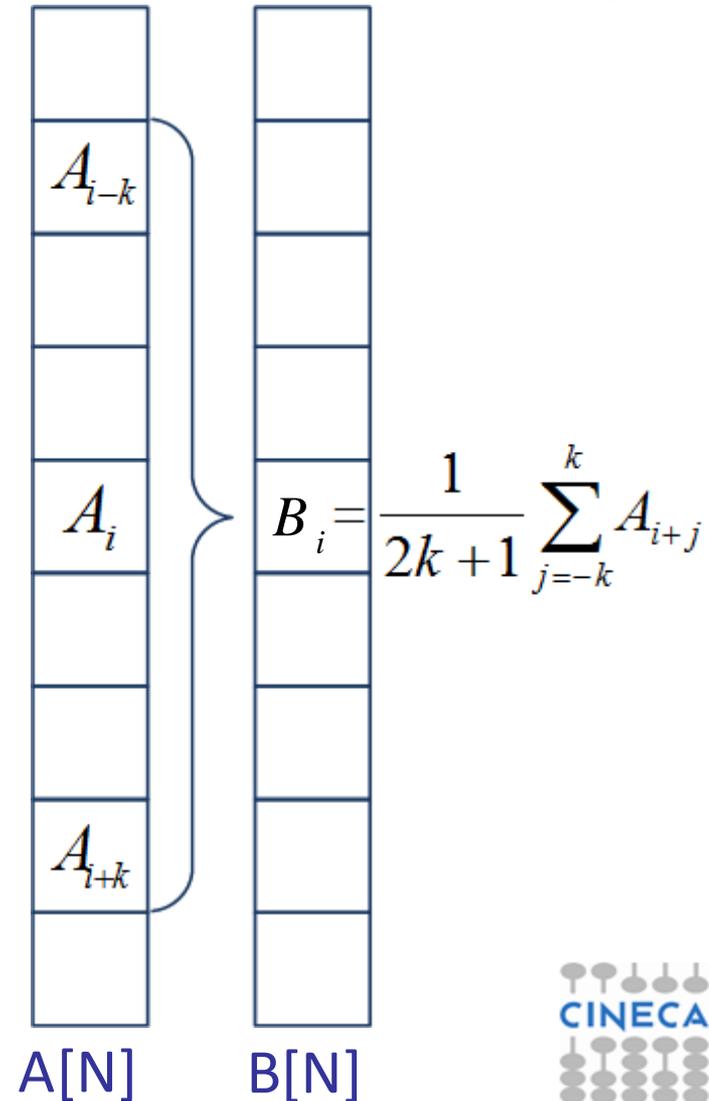


- † Ogni processo genera un array A , popolandolo con interi pari al proprio rank
- † Ogni processo invia il proprio array A al processo con rank immediatamente successivo
 - ‡ Periodic Boundary: L'ultimo processo invia l'array al primo processo
- † Ogni processo riceve l'array A dal processo immediatamente 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*

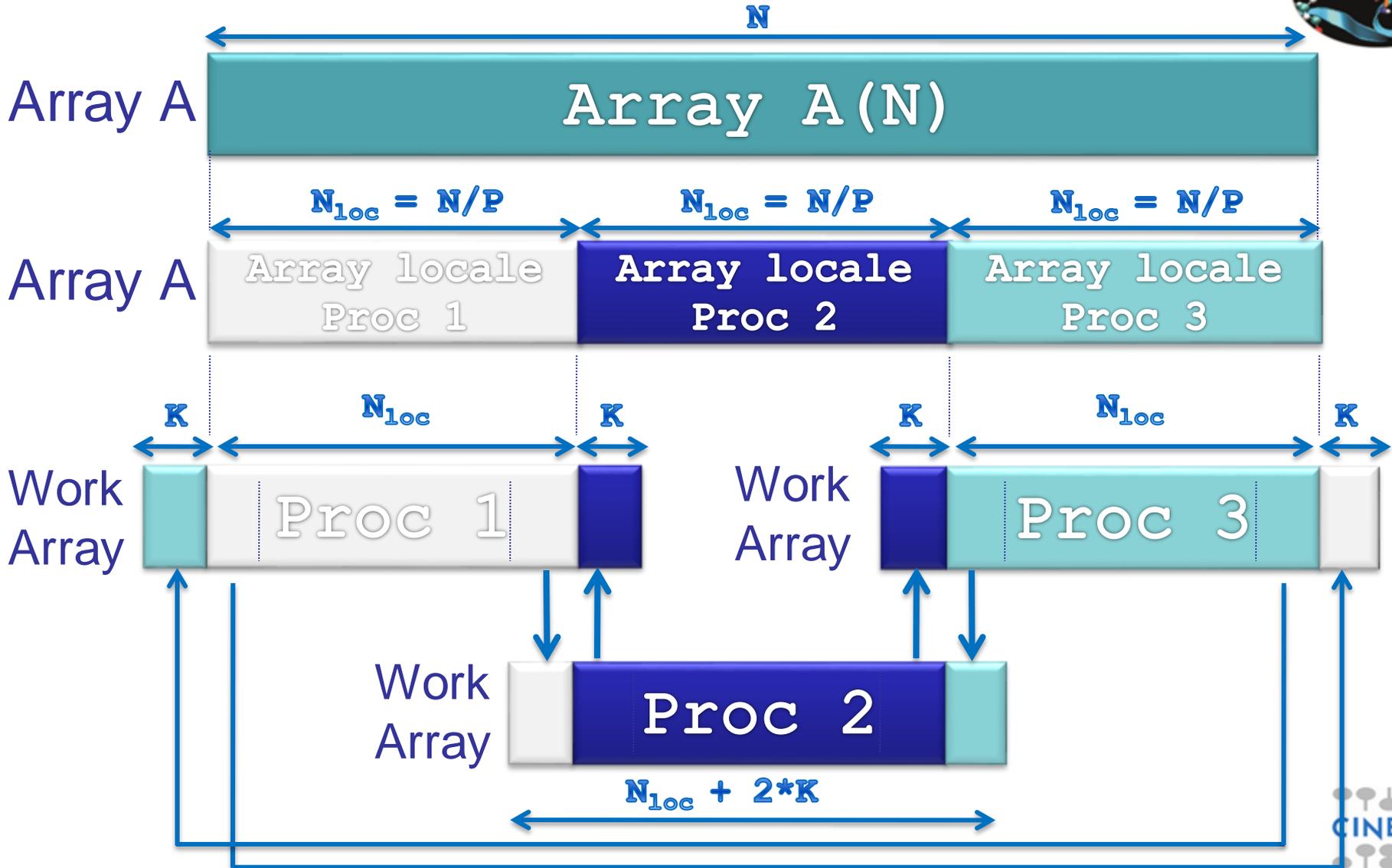
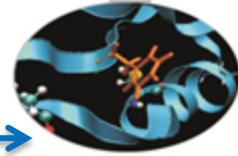


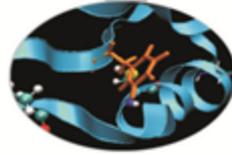
Array smoothing

- ☛ Dato un *array* $A[N]$
 - ☛ inizializzare e 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



Array smoothing: algoritmo parallelo





Array smoothing: algoritmo parallelo

Il processo di *rank 0*

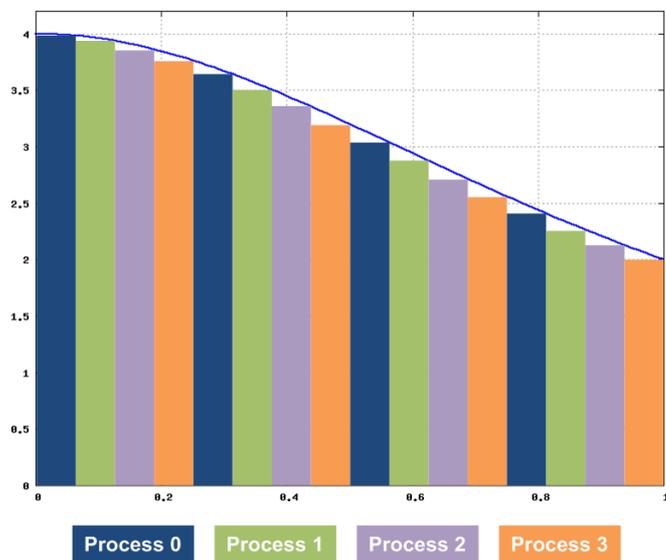
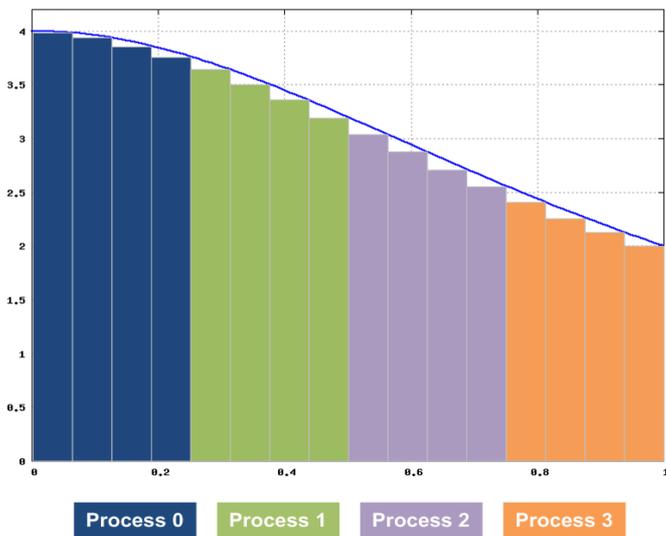
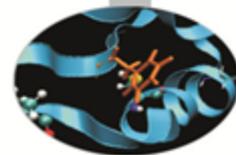
- genera l'*array* globale di dimensione N , multiplo del numero P di processi
- inizializza il vettore A con $A[i] = i$
- distribuisce il vettore A ai P processi i quali riceveranno N_{loc} elementi nell'*array* locale (MPI_Scatter)

Ciascun processo ad ogni passo di *smoothing*:

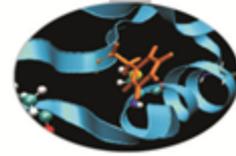
- costruisce l'*array* di lavoro:
 - I primi K elementi dovranno ospitare la copia degli ultimi K elementi dell'*array* locale in carico al processo precedente (MPI_Sendrecv)
 - I successivi N_{loc} elementi dovranno ospitare la copia degli N_{loc} elementi dell'*array* locale in carico al processo stesso
 - Gli ultimi K elementi dovranno ospitare la copia dei primi K elementi dell'*array* locale in carico al processo di *rank* immediatamente superiore (MPI_Sendrecv)
- Effettua lo *smoothing* degli N_{loc} elementi interni e scrive i nuovi elementi sull'*array* A

Il processo di *rank 0* ad ogni passo raccoglie (MPI_Gather) e stampa i risultati parziali

Calcolo di π con *reduction*: algoritmo parallelo

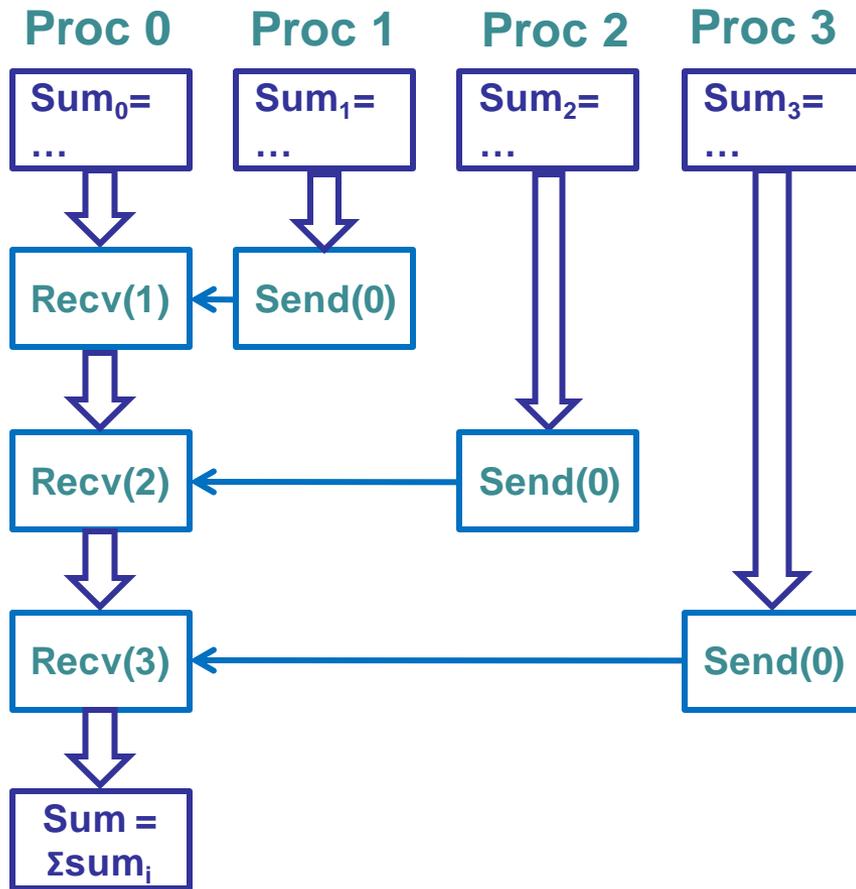


- Ogni processo calcola la somma parziale di propria competenza rispetto alla decomposizione scelta, come nel caso dell'esercizio 3
- Tutti i processi contribuiscono all'operazione di somma globale utilizzando la funzione di comunicazione collettiva `MPI_Reduce`

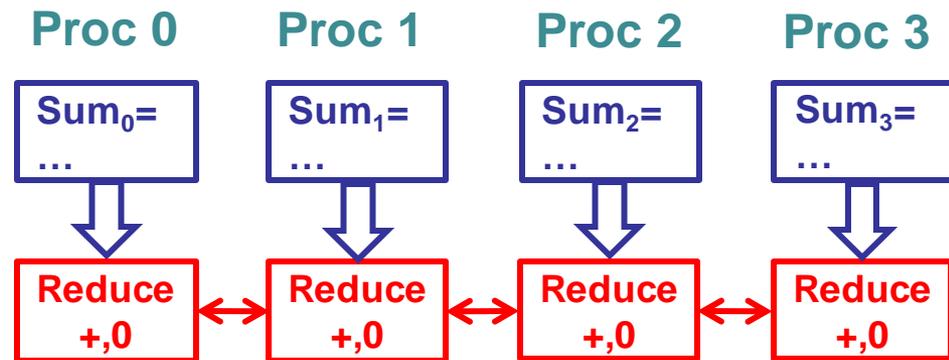


Calcolo di π in parallelo con *reduction*: flowchart

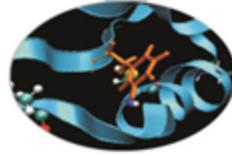
Standard



Reduction



Prodotto Matrice-Vettore

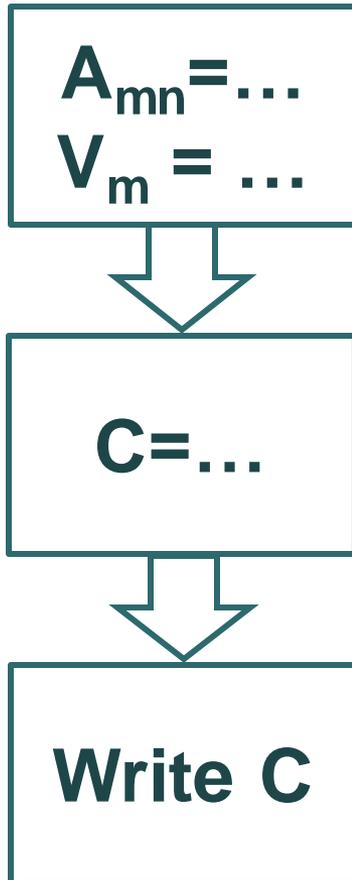
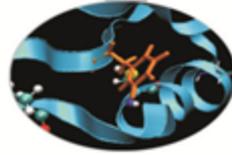


† Data una matrice A , di dimensione $size * size$, ed un vettore V di dimensione $size$, calcolare il prodotto $C=A * V$

† Ricordando che:

$$C_m = \sum_{n=1}^{size} A_{mn} V_n$$

† Nella versione parallela, per semplicità, assumiamo che $size$ sia multiplo del numero di processi



- Inizializzare gli array A e V

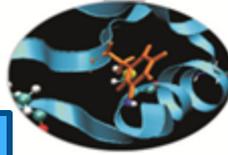
- $A_{mn} = m+n$
- $V_m = m$

- Core del calcolo

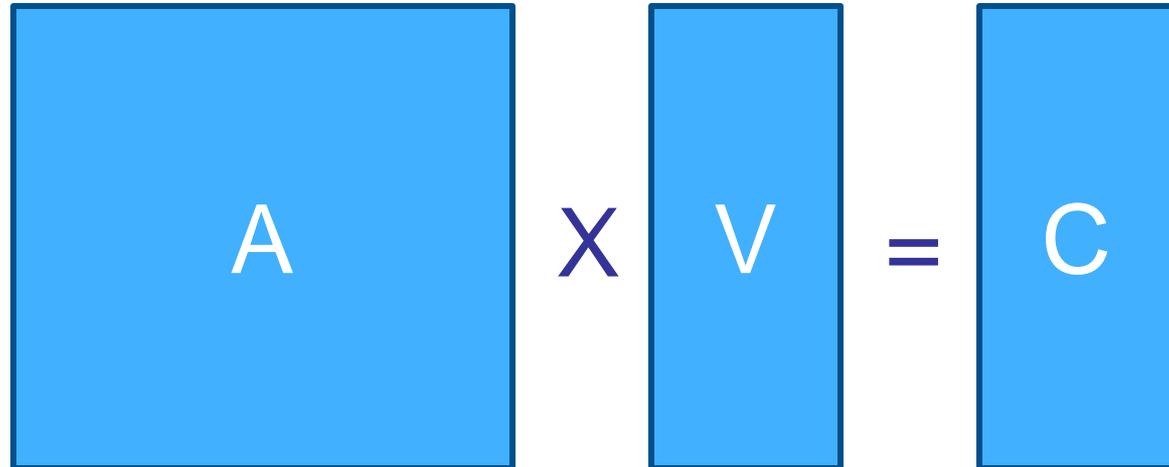
- Loop esterno sull'indice $m=1, \text{size}$ di riga della matrice A (e del vettore C)
 - Loop interno sull'indice $n=1, \text{size}$ del vettore V
- $$C_m = \sum_n (A_{mn} * V_n)$$

- Scrittura del vettore C

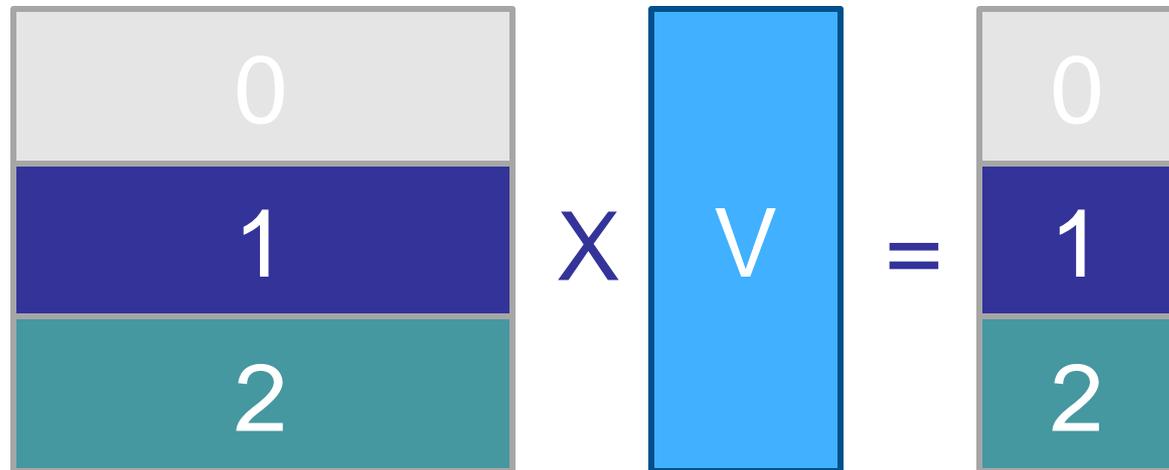
Prodotto matrice-vettore



Versione
seriale

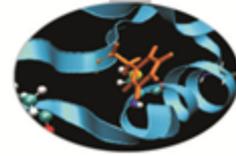


Versione
parallela

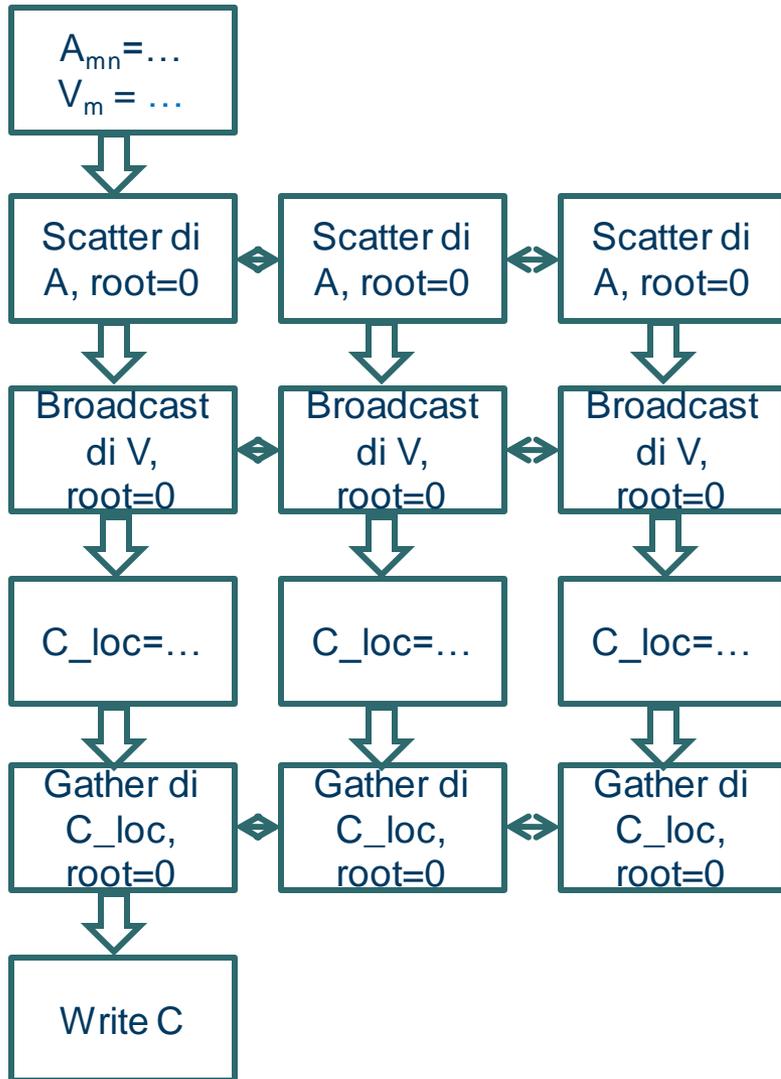


N.B. Poiché in Fortran le matrici sono allocate per colonne, è necessario effettuare la trasposizione della matrice A

Prodotto matrice-vettore in parallelo



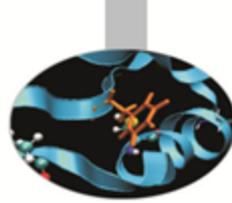
Rank = 0 Rank = 1 Rank = 2



- † Inizializzare gli array A e V sul solo processo master (rank = 0)
- † Scatter della matrice A
 - † Il processo master distribuisce a tutti i processi, se stesso incluso, un sotto-array (un set di righe contigue) di A
 - † I vari processi raccolgono i sotto-array di A in array locali al processo (es. A_loc)
- † Broadcast del vettore V
 - † Il processo master distribuisce a tutti i processi, se stesso incluso, l'intero vettore V
- † Core del calcolo sui soli elementi di matrice locali ad ogni processo (es. A_loc e C_loc)
 - † Loop esterno sull'indice $m=1, size/nprocs$
 - † Loop interno sull'indice $n=1, size$
 - $$C_loc_m = \sum_n (A_loc_{mn} * V_n)$$
- † Gather del vettore C
 - † Il processo master (rank = 0) raccoglie gli elementi di matrice del vettore risultato C calcolate da ogni processo (C_loc)
- † Scrittura del vettore C da parte del solo processo master

Prodotto Matrice-Matrice

esercizio facoltativo

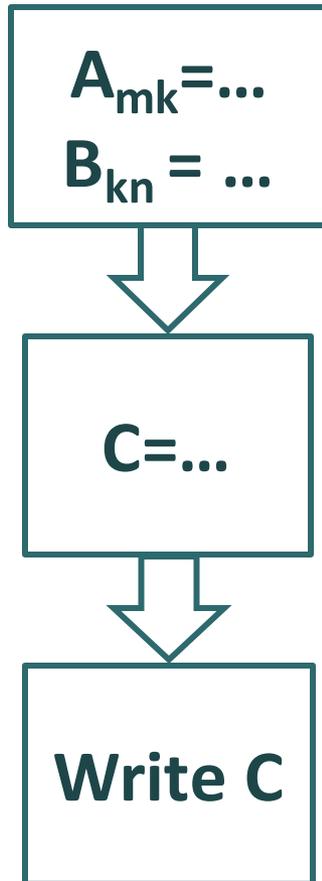
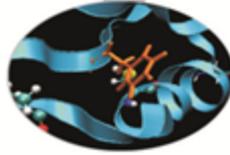


- † Date due matrici A e B di dimensione $size * size$, calcolare il prodotto $C=A*B$
- † Ricordando che:

$$C_{mn} = \sum_{k=1}^{size} A_{mk} B_{kn}$$

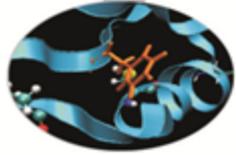
- † La versione parallela andrà implementata assumendo che $size$ sia multiplo del numero di processi

Prodotto matrice-matrice: algoritmo seriale



- † Inizializzazione degli array A e B
 - ‡ $A_{mk} = m+k$
 - ‡ $B_{kn} = n+k$
- † Core del calcolo
 - ‡ Loop esterno sull'indice $m=1$, size di riga della matrice A (e della matrice C)
 - ‡ Loop intermedio sull'indice $n=1$, size di colonna della matrice B (e della matrice C)
 - ‡ Loop interno sull'indice $k=1$, size di colonna della matrice A e di riga della matrice B
 - ‡ Calcolo del prodotto $A_{mk} * B_{kn}$ ed accumulo su C_{mn}
- † Scrittura della matrice C

Prodotto matrice-matrice in C



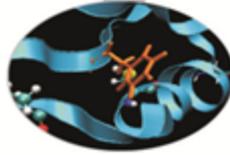
Versione
seriale



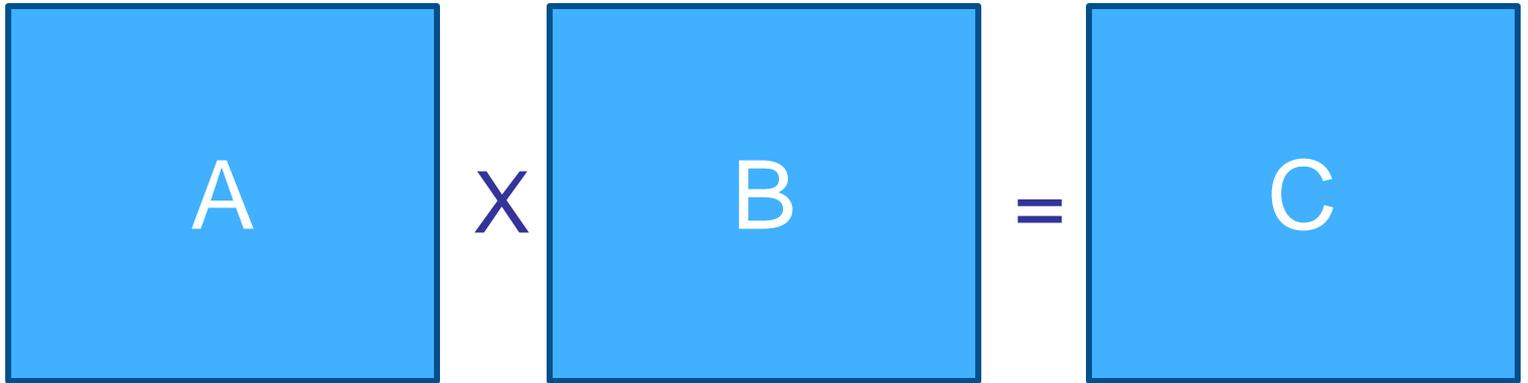
Versione
parallela



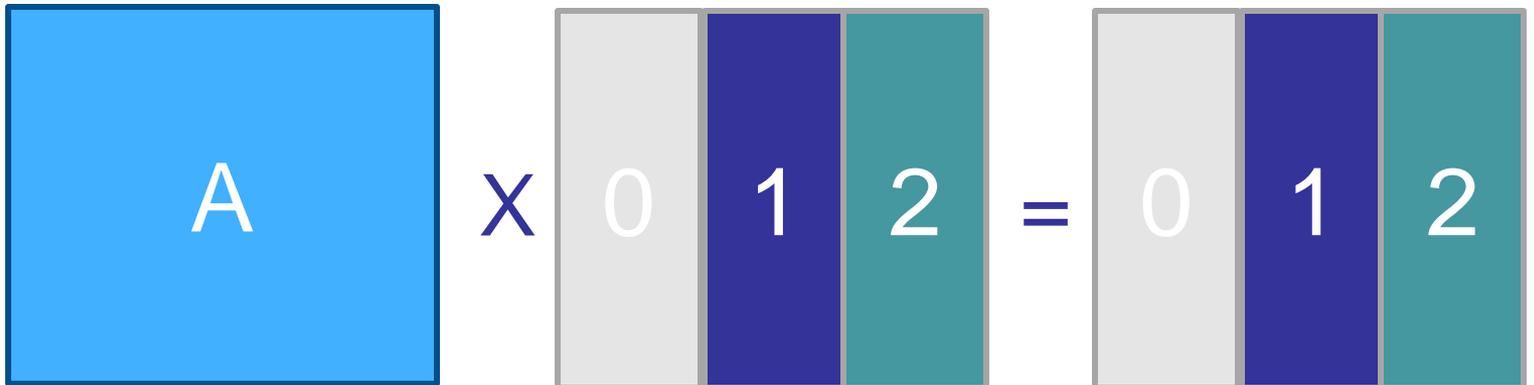
Prodotto matrice-matrice in Fortran

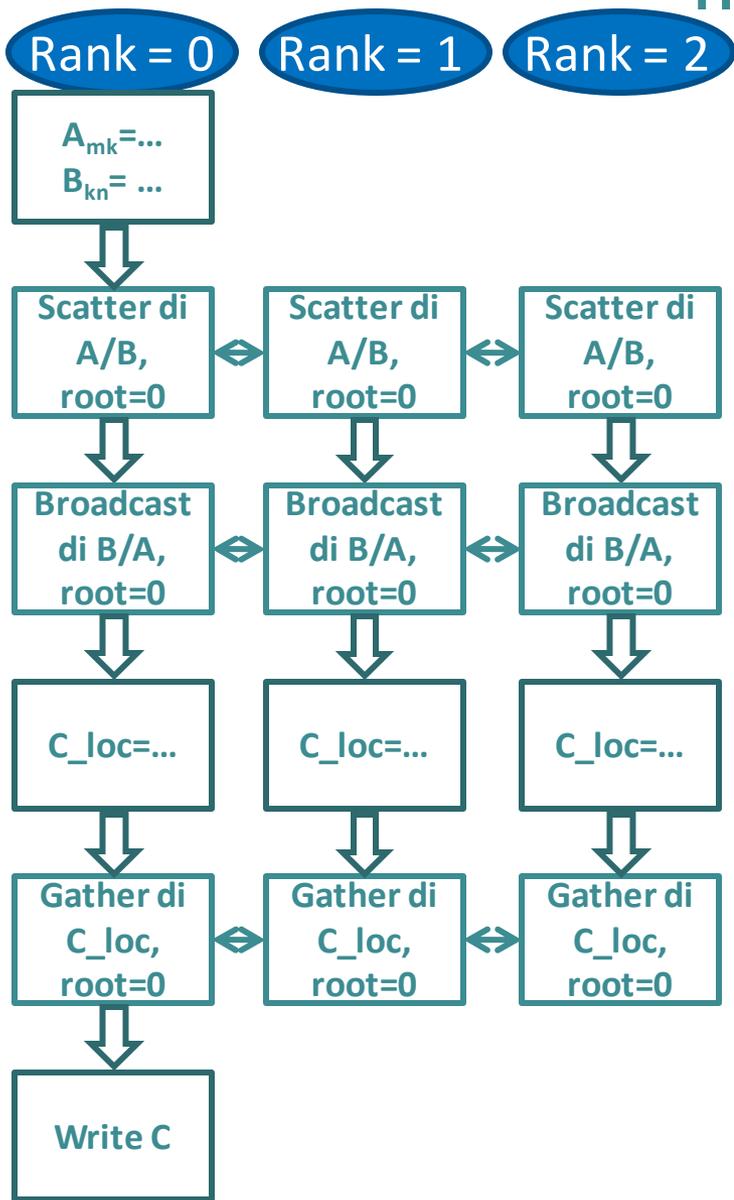
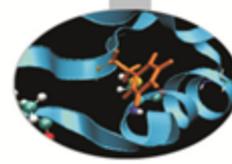


Versione
seriale



Versione
parallela





- † Inizializzare le matrici A e B sul solo processo master (rank = 0)
- † In C : Scatter della matrice A e Broadcast di B (viceversa in Fortran)
 - Il processo master distribuisce a tutti i processi, se stesso incluso, un sotto-array (un set di righe contigue) di A (B in Fortran)
 - Il processo root distribuisce a tutti i processi, se stesso incluso, l'intera matrice B (A)
 - I vari processi raccolgono i sotto-array di A (B) in un array locale al processo (es A_{loc})
- † Core del calcolo sui soli elementi di matrice locali ad ogni processo
 - Loop esterno sull'indice $m=1, size/nprocs$
 - Accumulo su C_{loc_m}
- † Gather della matrice C
 - Il processo master (rank = 0) raccoglie gli elementi della matrice risultato C calcolate da ogni processo (C_{local})
- † Scrittura della matrice C da parte del solo processo master