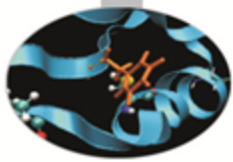
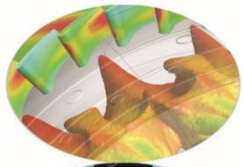
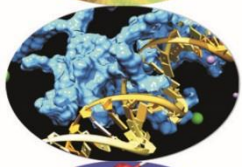
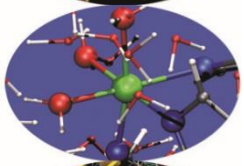
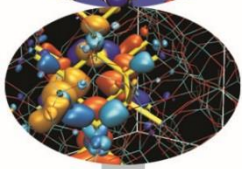
CINECA SCAI
SuperComputing Applications and Innovation



Introduzione al Calcolo Parallelo con MPI

(1° parte)

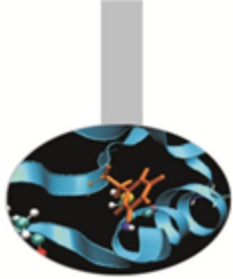





Claudia Truini
c.truini@cineca.it

Mariella Ippolito
m.ippolito@cineca.it

CINECA

Presentazione del corso



Cosa è il calcolo parallelo

- Serie di Fibonacci
- Serie geometrica

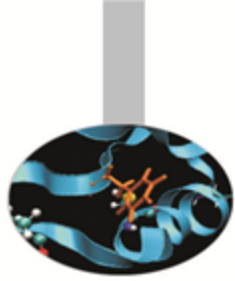
Calcolo parallelo: MPI

- Introduzione alla comunicazione point-to-point
- Le sei funzioni di base

Laboratorio 1

- Introduzione all'ambiente di calcolo
- Esercizi sulle comunicazioni point-to-point

Problema 1: Serie di Fibonacci



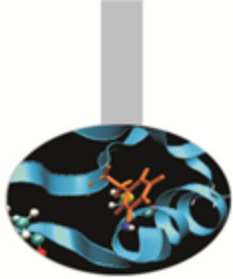
Calcolare e stampare i primi N elementi della serie di Fibonacci

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

$$f_1 = 1; \quad f_2 = 1$$

$$f_i = f_{i-1} + f_{i-2} \qquad \forall i > 2$$

Problema 2: Serie geometrica



Calcolare la somma parziale N-sima della serie geometrica

- La serie geometrica è così definita:

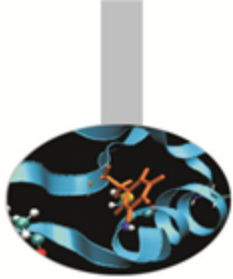
$$g_1 = x, \quad g_2 = x^2, \quad g_3 = x^3, \ldots.$$

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

- Dobbiamo calcolare:

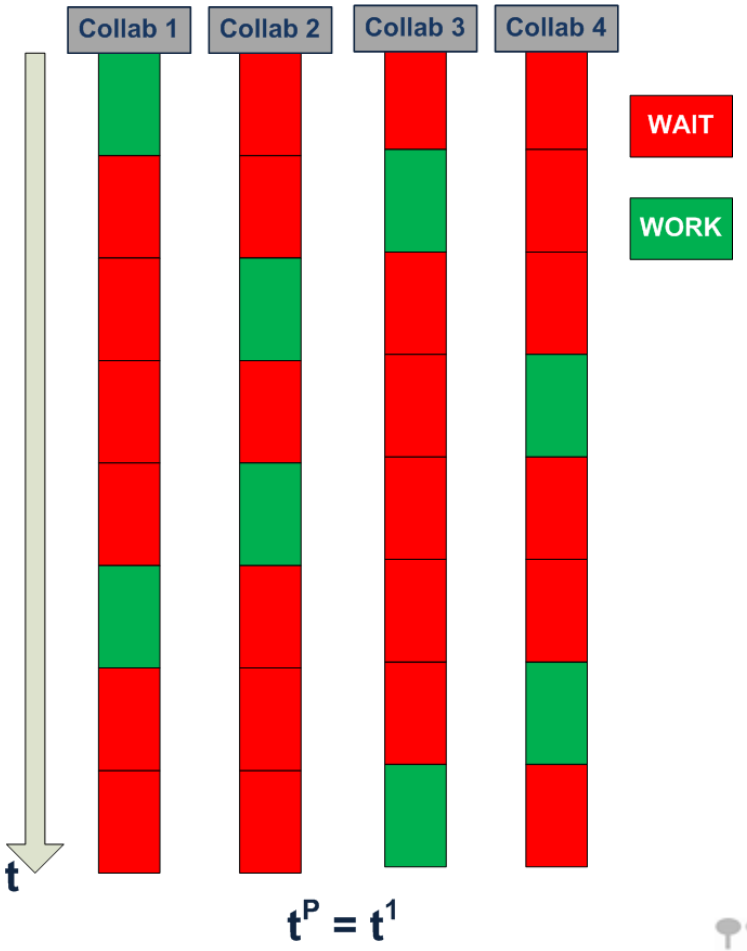
$$G_N = \sum_{i=1}^{N} x^i$$

Risoluzione del problema 1 con P collaboratori

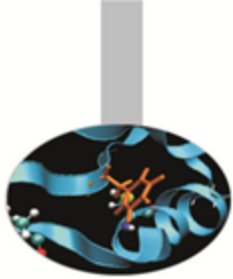


Come calcolare nel minor tempo i primi N numeri di Fibonacci?

- 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



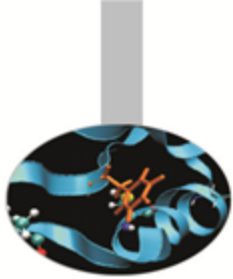
$$G_N = \sum_{i=1}^{N} x^i = \sum_{i=1}^{P} \left(\sum_{j=1}^{N/P} x^{\frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^{P} S_i$$

dove

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

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

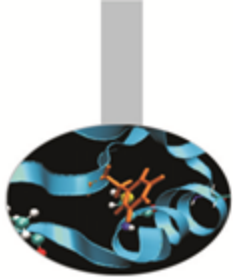
Benefici nell'uso di P collaboratori



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

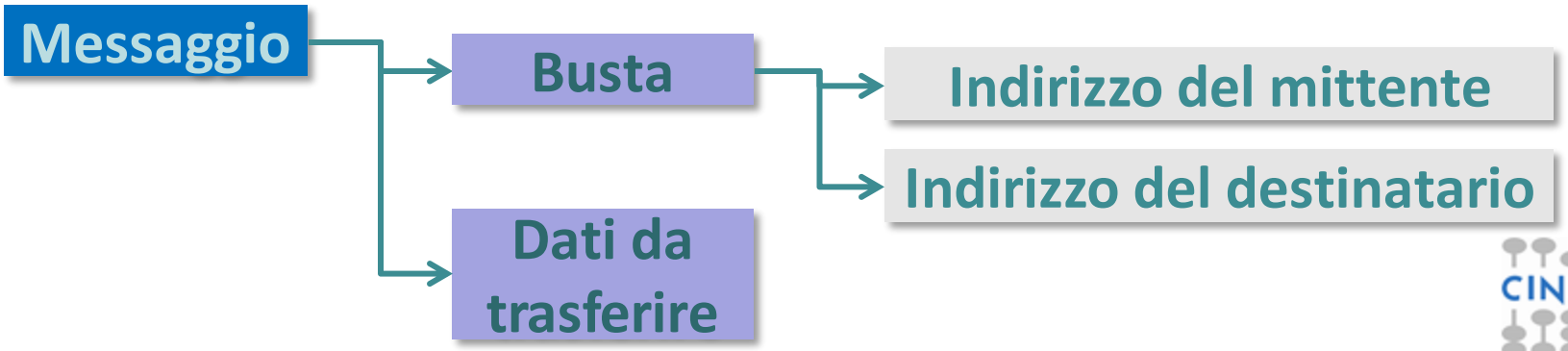
La comunicazione tra collaboratori: il messaggio



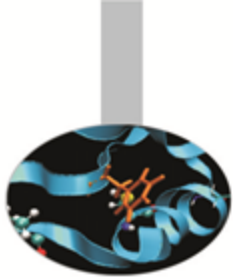
- Per una “fruttuosa cooperazione” tra P collaboratori è necessario che gli stessi possano scambiarsi dati

- *Problema 2*: Serie geometrica
  quando i P collaboratori terminano il calcolo della somma parziale di propria competenza, uno di essi
  1. richiede a tutti gli altri la somma parziale di competenza
  2. somma al proprio i P-1 risultati parziali ricevuti

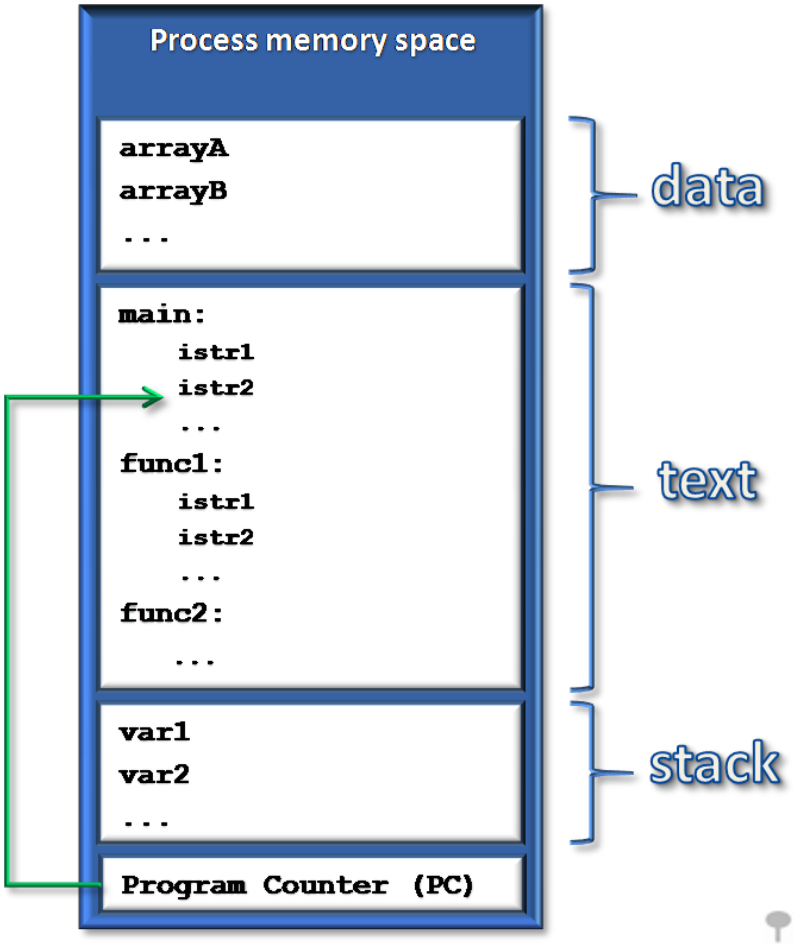
- Il trasferimento di dati tra collaboratori può avvenire attraverso la *spedizione/ricezione di un messaggio*:



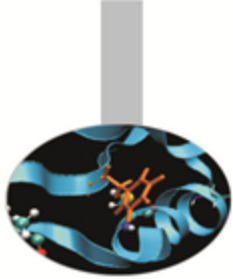
Il collaboratore nel *computing*: il processo



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

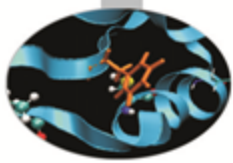


Gruppo di collaboratori → Calcolo Parallelo



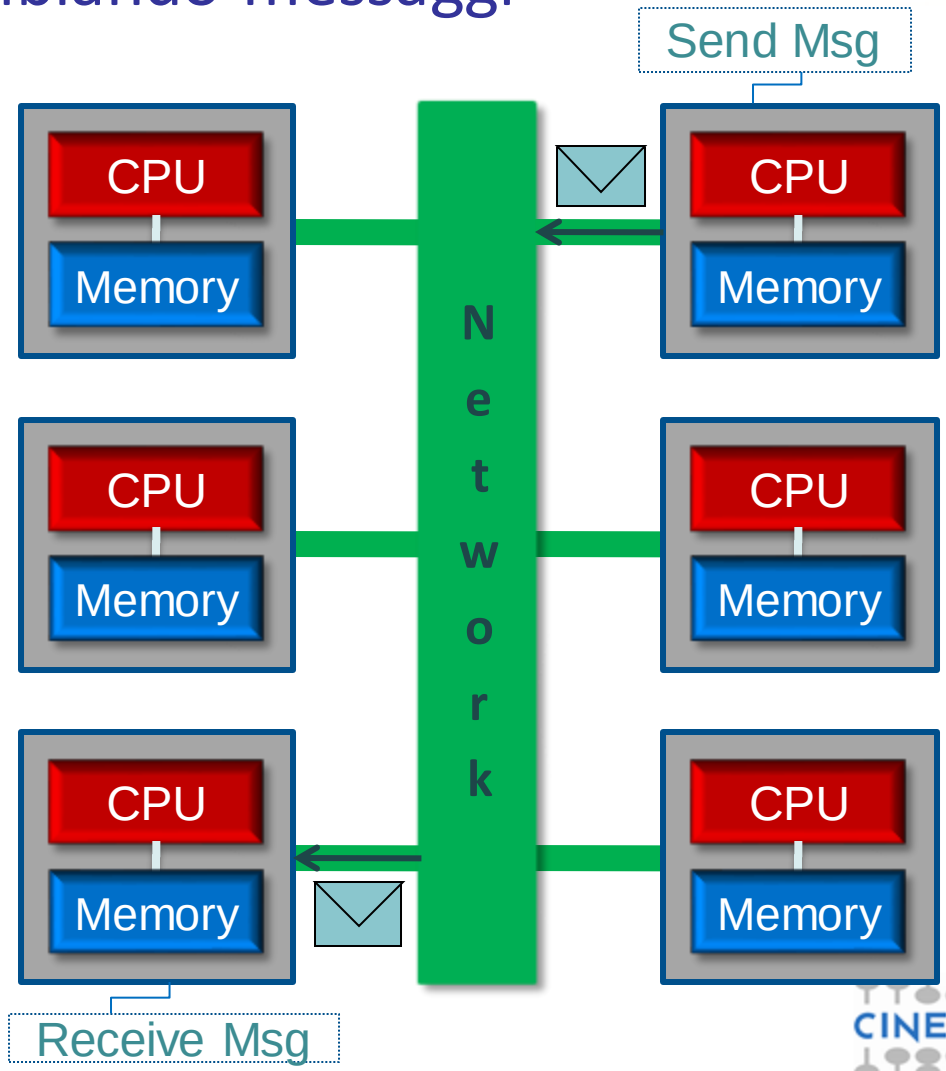
- Il calcolo parallelo, in generale, è l'uso simultaneo di più processi per risolvere un unico problema computazionale
- Per girare con più processi, un problema è diviso in parti discrete che possono essere risolte concorrentemente
- Le istruzioni che compongono ogni parte sono eseguite contemporaneamente 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

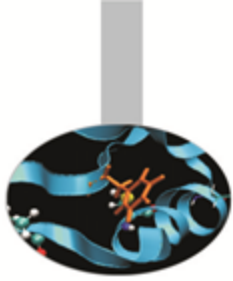


N processi cooperano scambiando messaggi

- 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

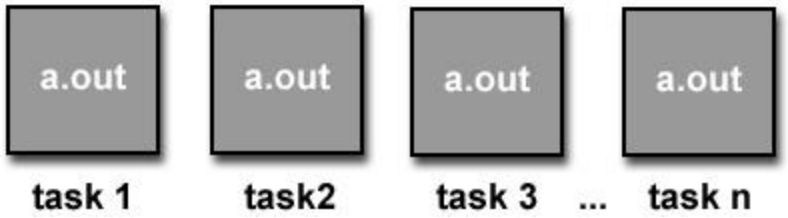


Modello di esecuzione: SPMD



SPMD = Single Program Multiple Data

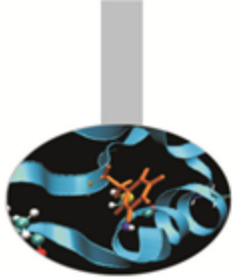
Ogni processo esegue lo stesso programma, ma opera in generale su dati diversi (nella propria memoria locale)



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 (1° parte)



Cosa è il calcolo parallelo

- Serie di Fibonacci
- Serie geometrica

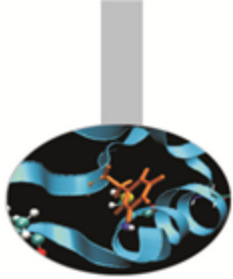
Calcolo parallelo: MPI

- Introduzione alla comunicazione point-to-point
- Le sei funzioni di base

Laboratorio 1

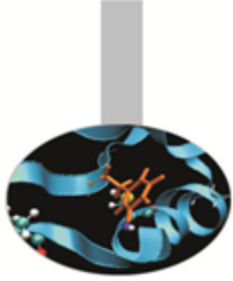
- Introduzione all'ambiente di calcolo
- Esercizi sulle comunicazioni point-to-point

Cos'è MPI



- **MPI** acronimo di **M**essage **P**assing **I**nterface
  - **http://www.mpi-forum.org**
  - *'MPI is a message passing library interface specification'*
  - MPI è una specifica, non un'implementazione
  - ***Standard*** per sviluppatori ed utenti
  - MPI è uno strumento di programmazione che permette di implementare il modello di calcolo parallelo message-passing
  - MPI consente di:
    - generare e gestire il gruppo di collaboratori (processi)
    - scambiare dati tra loro

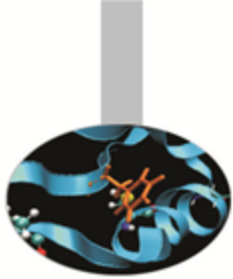
Funzionalità di MPI



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, …)`

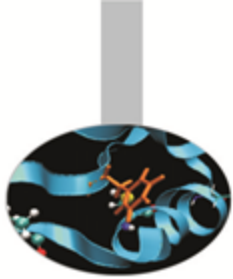
  - **`MPI_`** è prefisso di tutte le funzioni MPI
  - Dopo il prefisso, la prima lettera è maiuscola e tutte le altre minuscole
  - Praticamente tutte le funzioni MPI tornano un codice d'errore intero
  - Le macro sono scritte tutte in maiuscolo

- In Fortran:

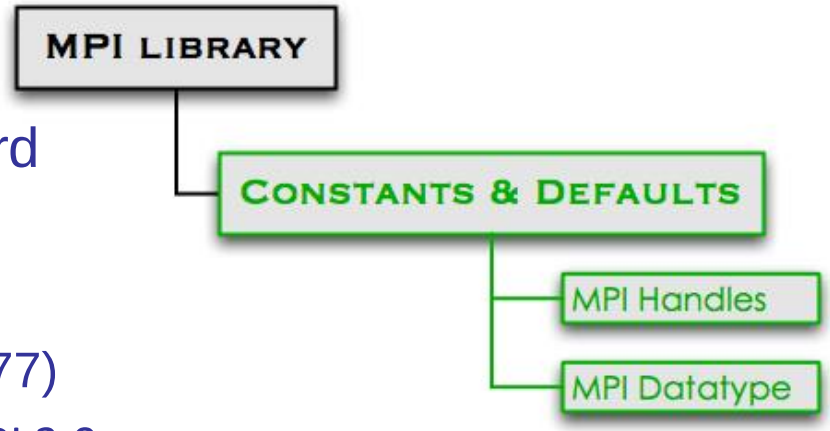
  `call MPI_XXXX(parameter,…, err)`

  - **`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

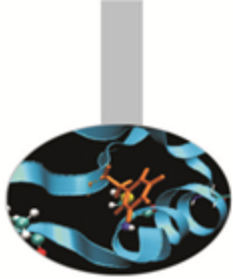


- Tutti i programmi che usano MPI devono includere *l'header file* (o *modulo*) standard
  - C: **mpi.h**
  - Fortran: **mpif.h**
  - Fortran: **use mpi** (no Fortran 77)
  - Fortran: **use mpi_f08** ← da MPI 3.0

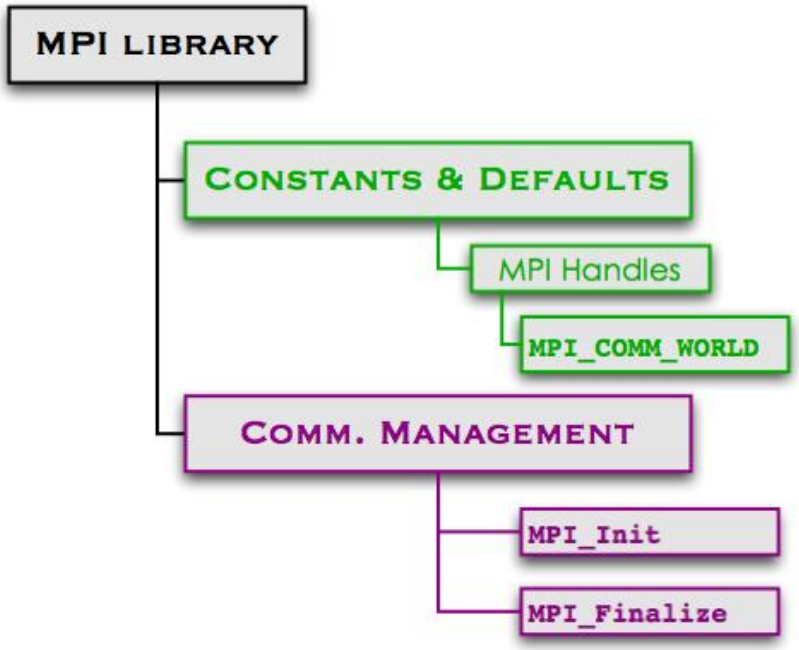


- Nell'*header file* standard sono contenute **definizioni**, **macro** e **prototipi di funzioni** necessari per la compilazione di un programma MPI
  - MPI mantiene strutture di dati interne legate alla comunicazione, referenziabili tramite *MPI Handles*
  - MPI referenzia i tipi di dati standard dei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

Communication Environment

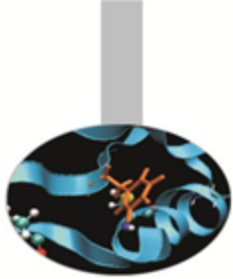


- **`MPI_Init`**
  - inizializza l'ambiente di comunicazione
  - tutti i programmi MPI devono contenere una sua chiamata
  - deve essere chiamata prima di qualsiasi altra routine MPI
  - deve essere chiamata una sola volta



- **`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)
```

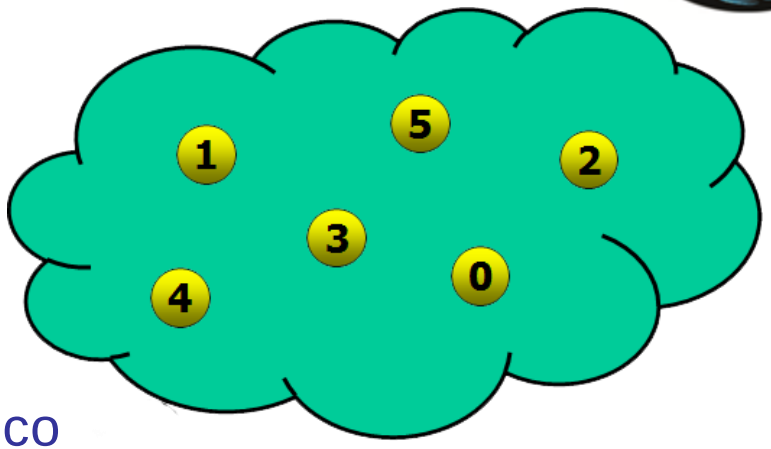
In Fortran

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

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

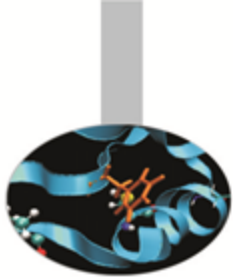
In C, la funzione **`MPI_Init`** esegue il parsing degli argomenti forniti al programma da linea di comando

Comunicatori

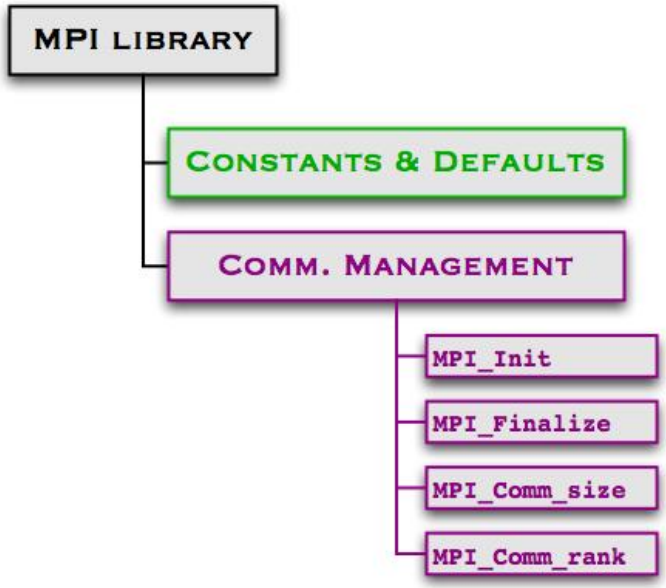


- Un comunicatore è un "oggetto" contenente un *gruppo* di processi ed un set di attributi associati
- All'interno di un comunicatore ogni processo ha un identificativo unico
- Due o più processi possono comunicare solo se fanno parte dello stesso comunicatore
- La funzione MPI_Init inizializza il comunicatore 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

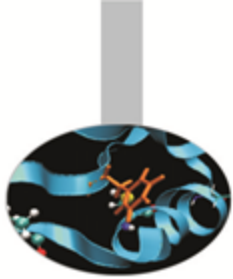


- *Communicator size*
  - La *size* di un comunicatore è la dimensione del gruppo di processi in esso contenuti
  - Un processo può determinare la *size* di un comunicatore di cui fa parte con una chiamata alla funzione **`MPI_Comm_size`**
  - La *size* di un comunicatore è un intero



- *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

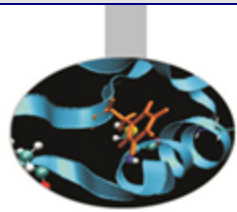
```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

In Fortran

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

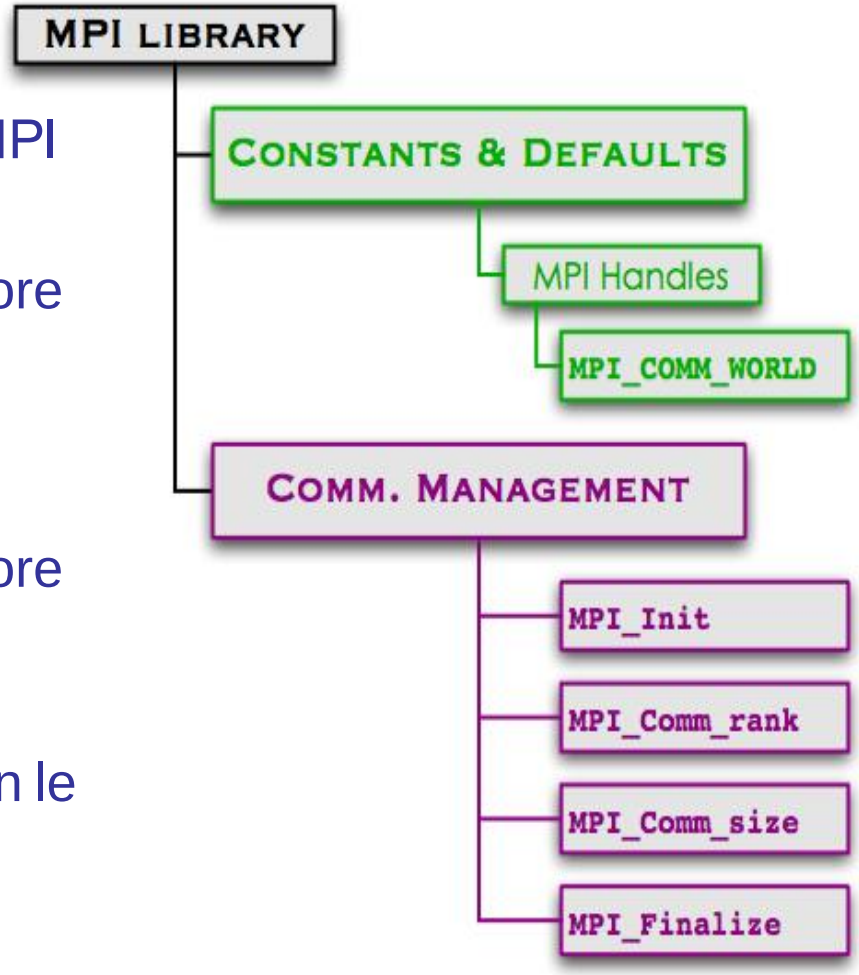
- Input:
  - **comm** è di tipo **MPI_Comm** (INTEGER) ed è il comunicatore di cui si vuole conoscere la dimensione o all'interno del quale si vuole conoscere il rank del processo chiamante
- Output:
  - **size** è di tipo **int** (INTEGER) e conterrà la dimensione di **comm**
  - **rank** è di tipo **int** (INTEGER) e conterrà il *rank* nel comunicatore **comm** del processo chiamante

Il primo programma MPI

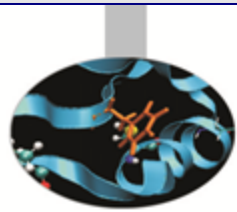


Operazioni da eseguire:

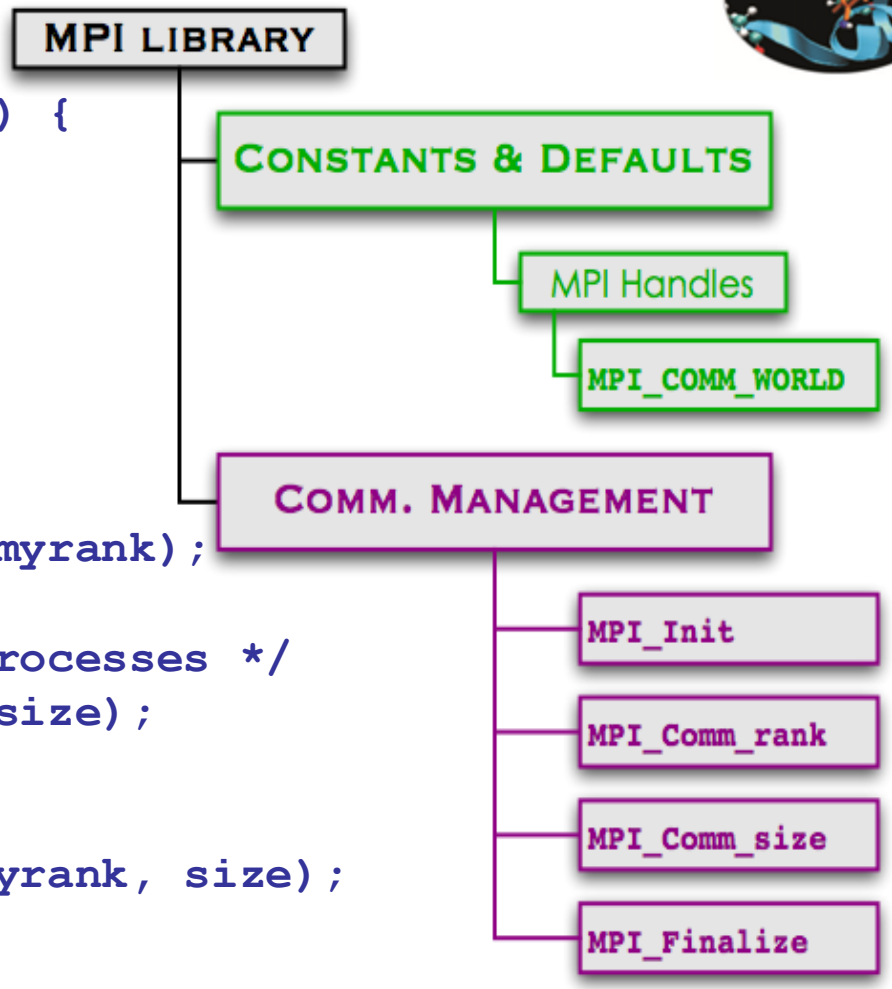
1. Inizializzare l'ambiente MPI
2. Richiedere al comunicatore di default il *rank* del processo
3. Richiedere al comunicatore di default la sua *size*
4. Stampare una stringa con le informazioni ottenute
5. Chiudere l'ambiente MPI



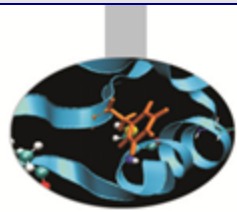
La versione in C …

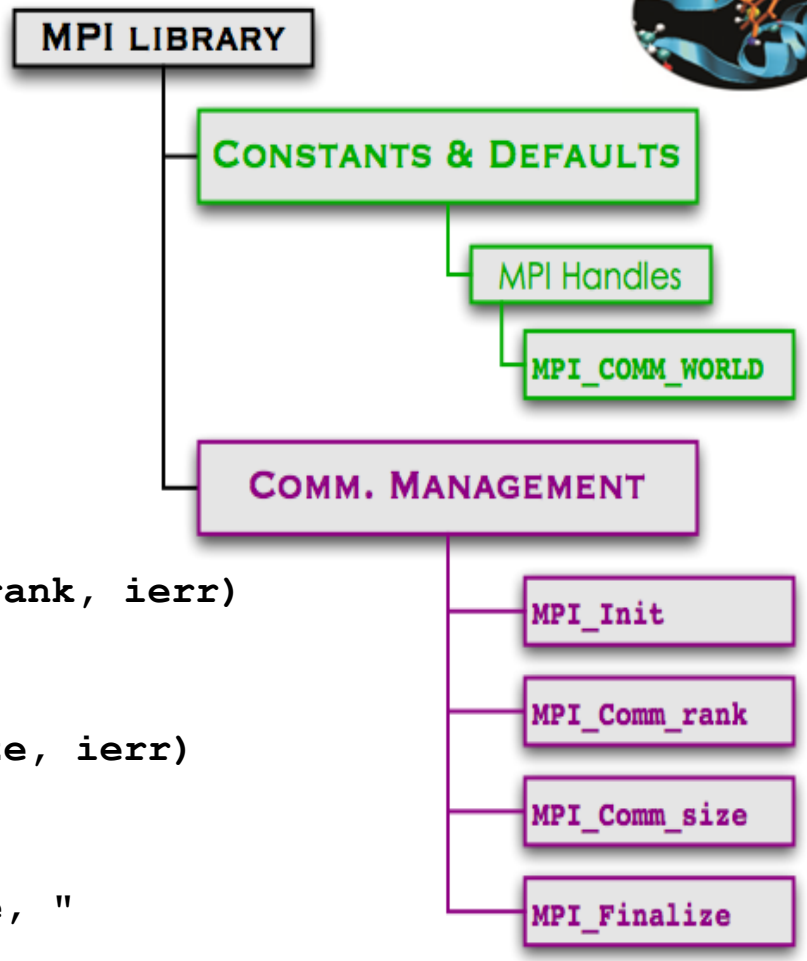


```c
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {

  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();
}
```



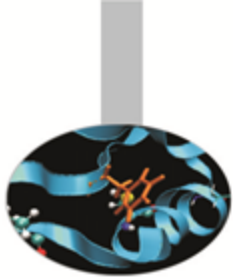
.. e quella in Fortran





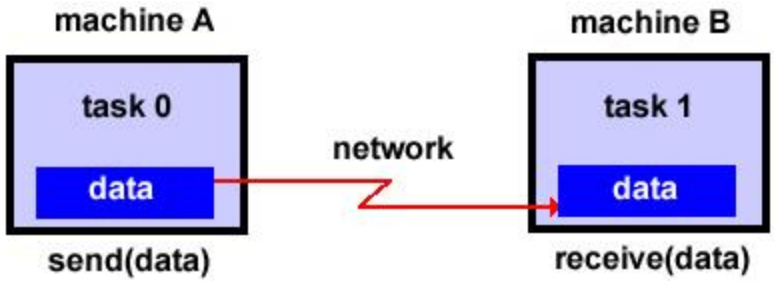

```
 PROGRAM hello

    use mpi
    INTEGER myrank, size, ierr

! 1. Initialize MPI:
    call MPI_INIT(ierr)

! 2. Get my rank:
    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

! 3. Get the total number of processes:
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

! 4. Print myrank and size
    PRINT *, "Process", myrank, "of", size, "

! 5. Terminate MPI:
    call MPI_FINALIZE(ierr)
    END
```

La comunicazione interprocesso

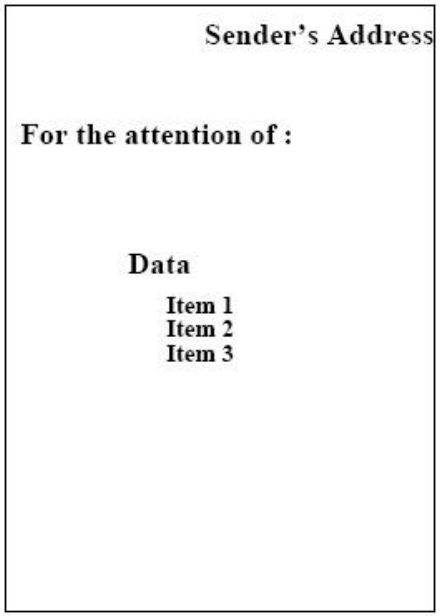


- Nella programmazione parallela ***message passing***
  la cooperazione tra processi avviene attraverso operazioni esplicite di comunicazione interprocesso

- 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

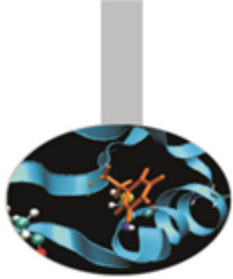


Cos'è un messaggio



- 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*



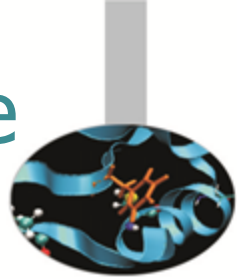
In C

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

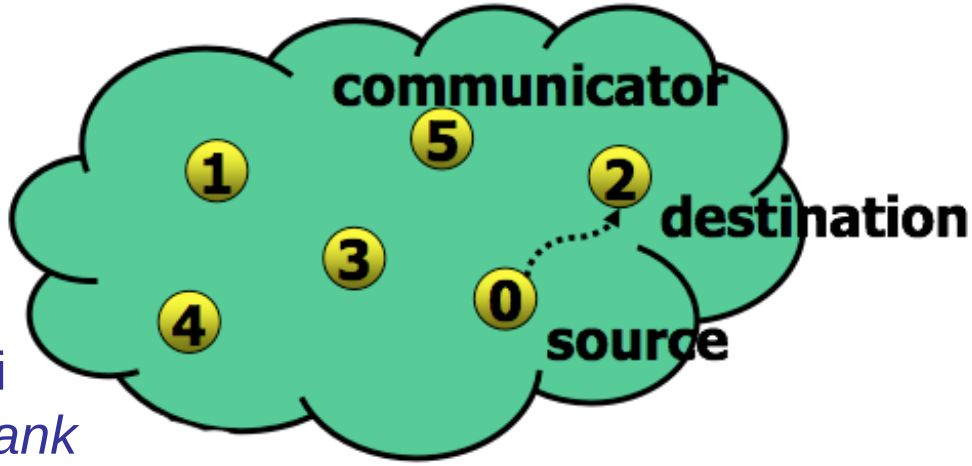
In Fortran

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

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



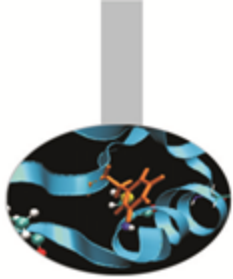
Per inviare/ricevere un messaggio:



- 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*

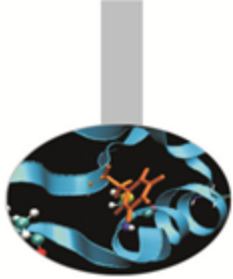
Inviare un messaggio



- 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 utilizzare

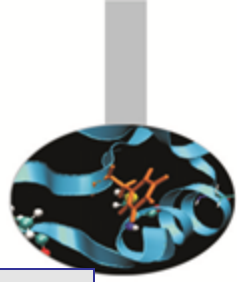
    sono determinati esplicitamente dagli argomenti che il processo sorgente passa alla funzione di *send*

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

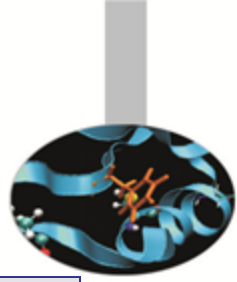
```
int MPI_Send(void *buf,int count,MPI_Datatype dtype,
             int dest,int tag, MPI_Comm comm)
```

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

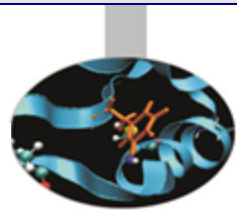
```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status)
```

In Fortran

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

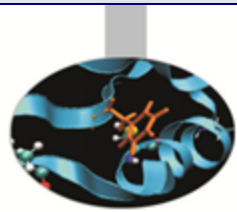
- [OUT] **buf** è l'indirizzo iniziale del *receive* buffer
- [IN] **count** è di tipo **int** e contiene il numero di elementi del *receive* buffer
- [IN] **dtype** è di tipo **MPI_Datatype** e descrive il tipo di ogni elemento del *receive* buffer
- [IN] **src** è di tipo **int** e contiene il *rank* 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)



```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;
}
```

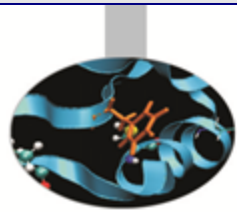
send/receive: *array* di double (FORTRAN)



```
      program main

      use mpi
      implicit none

      integer ierr, rank, size
      integer i, j, status(MPI_STATUS_SIZE)

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

!--- Start up MPI  ----------- -----------------------------------------
      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
            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
```

CINECA SCAI
SuperComputing Applications and Innovation

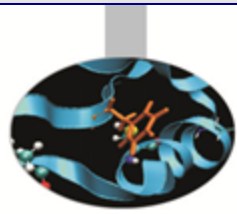
send/receive: *array* di *float* (C)



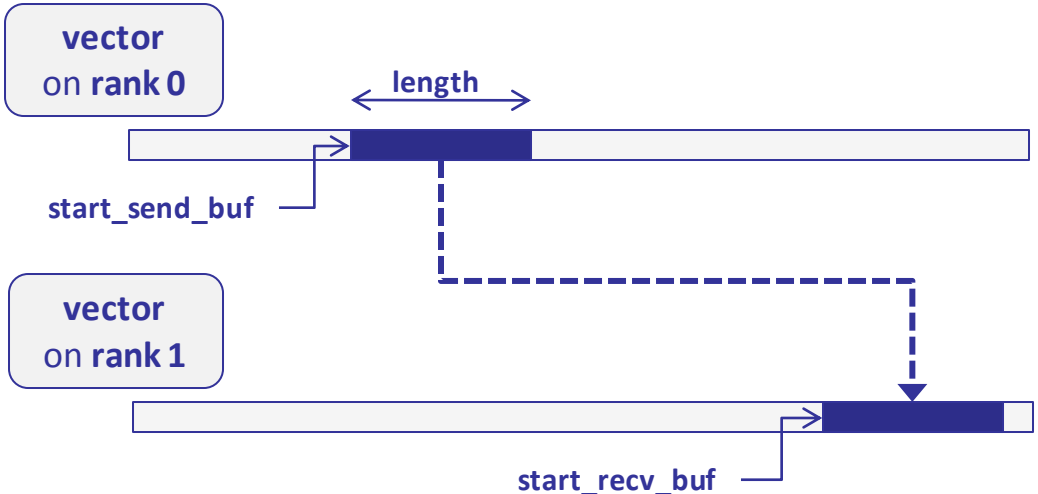
```c
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

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

    MPI_Status status;
    int rank, size;
    int i, 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;
}
```


CINECA

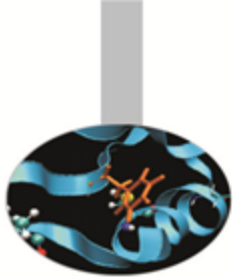
send/receive: porzione di array (C)



```c
#include <stdio.h>
#include <mpi.h>
#define VSIZE  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 = VSIZE - BORDER;
    int length = 10;
    int vector[VSIZE];

    /* 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 < VSIZE; 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;
}
```



Introduzione al calcolo parallelo con MPI (1° parte)



Cosa è il calcolo parallelo

- Serie di Fibonacci
- Serie geometrica

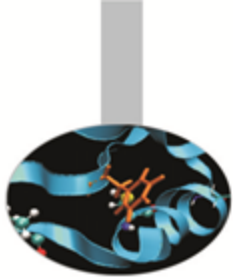
Calcolo parallelo: MPI

- Introduzione alla comunicazione point-to-point
- Le sei funzioni di base

Laboratorio 1

- Introduzione all'ambiente di calcolo
- Esercizi sulle comunicazioni point-to-point

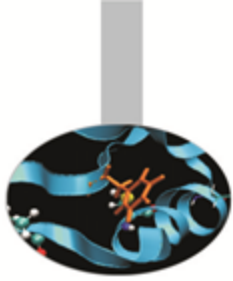
Compilare un sorgente MPI



- 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)



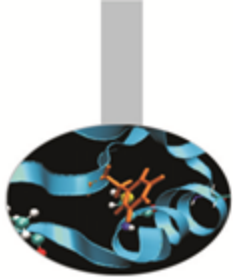
Per compilare il sorgente `sample.f90` usando:

- il compilatore gnu `gfortran` (linux)
- le librerie `libmpi_f90.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_f90 -lmpi -o sample.x
```

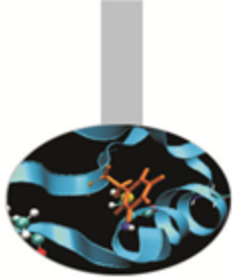
Compilare un sorgente MPI (3)



- … 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.f90`

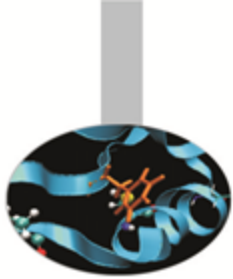
```
mpifort sample.f90 –o sample.x
```

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



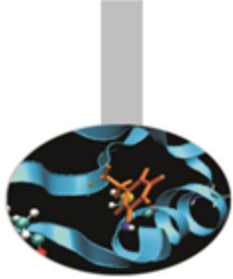
Il nome dei processori che ospiteranno i processi può essere scritto in un file con una specifica sintassi accettata dal *launcher* MPI

- Usando `OpenMPI` , 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 source:

```
mpifort sample.f90 -o sample.x  (da OpenMPI 1.7)
mpif77/mpif90 sample.f90 -o sample.x (versioni precedenti)
```

  - C source:

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

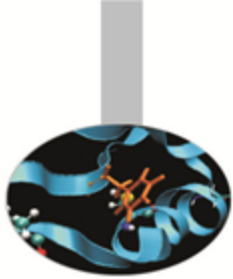
  - C++ source:

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

- Eseguire:
  - il launcher `OpenMPI` è `mpirun` (oppure `mpiexec`):

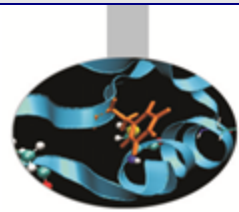
```
mpirun -hostfile my_hostfile -n 4 ./sample.x
```

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
```

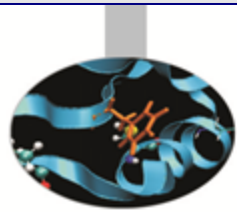
  - In Fortran:

```
mpifort helloworld.f90 -o hello.x
```

- Come si manda in esecuzione utilizzando 4 processi:

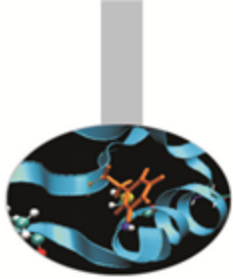
```
mpirun -n 4 ./hello.x
```

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



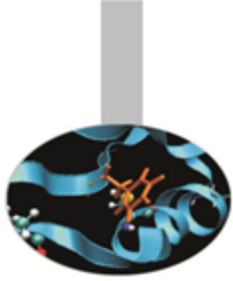
- Cosa succede se la lunghezza del messaggio ricevuto (r_count) è diversa dalla lunghezza del messaggio spedito (s_count) ?
  - Se s_count < r_count → solo le prime s_count locazioni di r_buf sono modificate
  - Se s_count > r_count → errore overflow

Inoltre

- La lunghezza del messaggio ricevuto (r_count) deve essere minore o uguale alla lunghezza del receive buffer (lenght_buf)
  - Se r_count < lenght_buf → solo le prime r_count locazioni di buf sono modificate
  - Se r_count > lenght_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

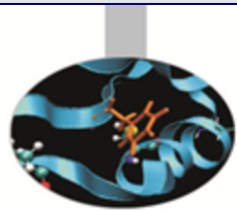
```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
  dtype, int *count)
```

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

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

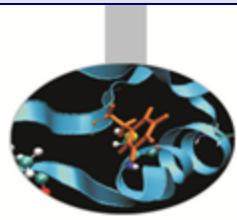
- In generale, se f è integrabile in [a,b]

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

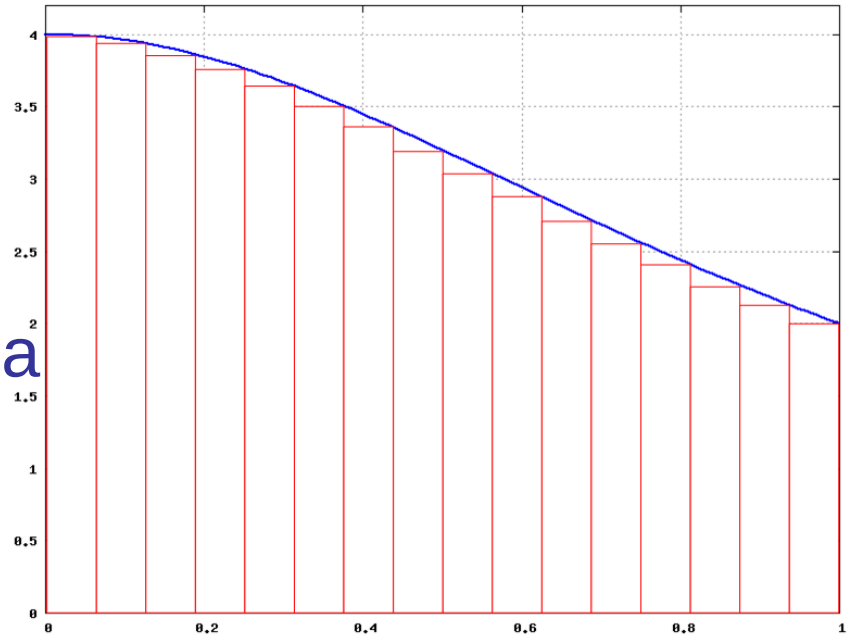
- Dunque, per N sufficientemente grande

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

Calcolo di π in seriale con il metodo integrale



- L'intervallo [0,1] è diviso in N sotto intervalli, di dimensione h=1/N



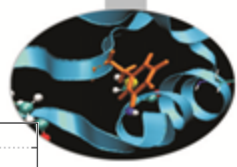
- L'integrale può essere approssimato con la somma della serie

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

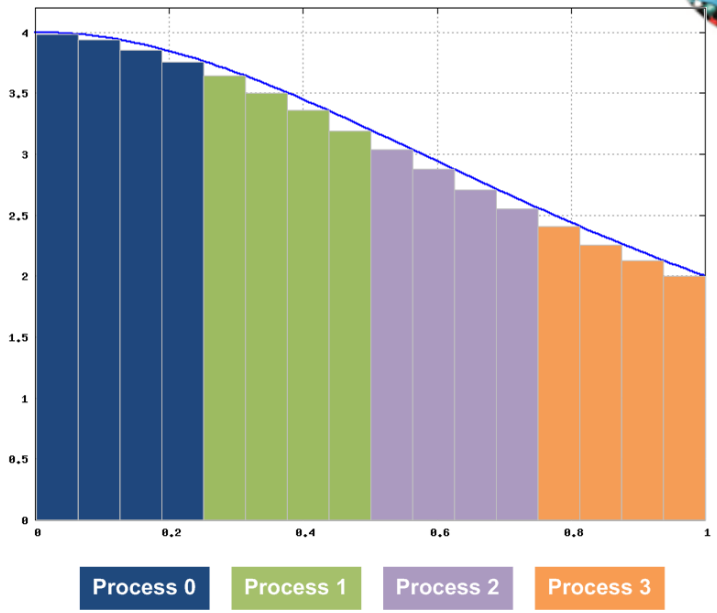
che è uguale alla somma delle aree dei rettangoli in rosso

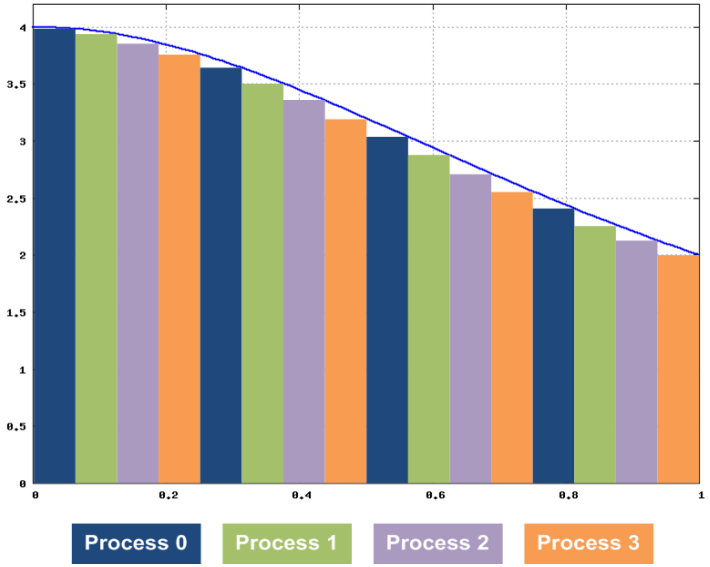
- Al crescere di N si ottiene una stima sempre più precisa di π

Il Calcolo di π: l'algoritmo parallelo

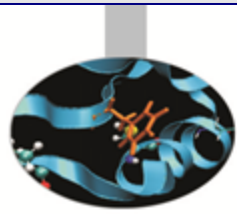


1. Ogni processo calcola la somma parziale di propria competenza rispetto alla decomposizione scelta
2. Ogni processo con *rank* ≠ 0 invia al processo di *rank* 0 la somma parziale calcolata
3. Il processo di *rank* 0
   - Riceve le P-1 somme parziali inviate dagli altri processi
   - Ricostruisce il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

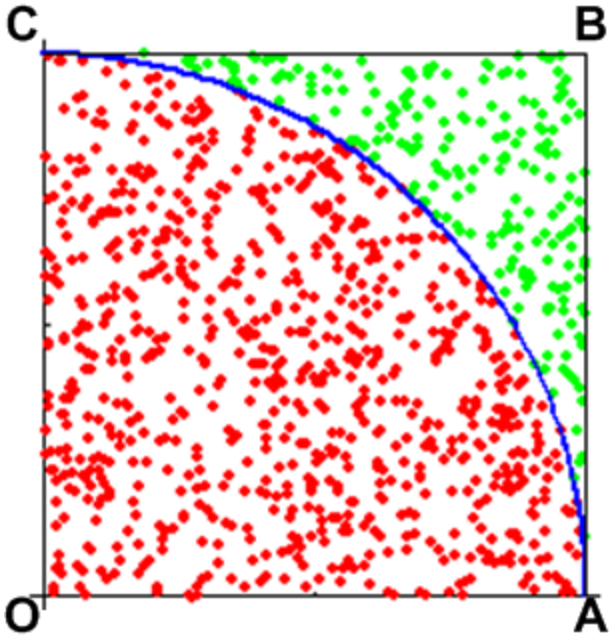




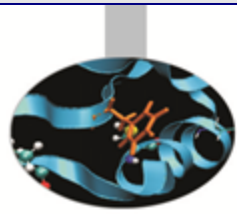
Il Calcolo di π con il metodo Monte Carlo



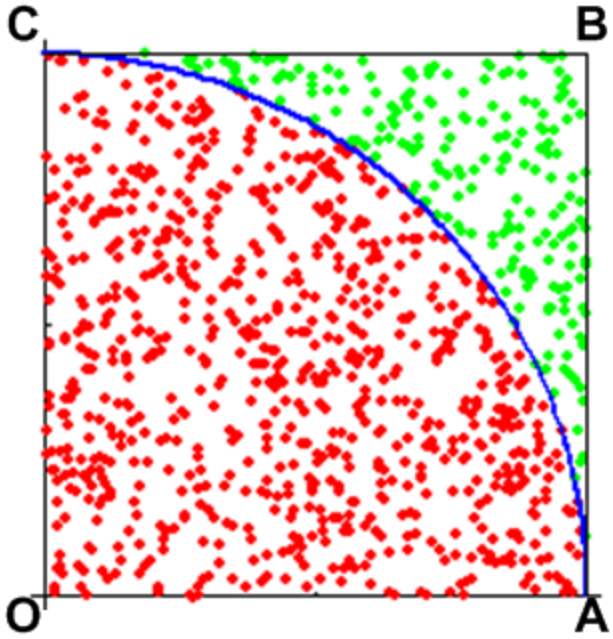
- *AOC* è il quadrante del cerchio unitario, la cui area è $\pi/4$
- Sia $Q = (x,y)$ una coppia di numeri casuali estratti da una distribuzione uniforme in $[0,1]$
- La probabilità p che il punto Q sia interno al quadrante *AOC* è pari al rapporto tra l'area di *AOC* e quella del quadrato *ABCO*, ovvero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



Il Calcolo di π in seriale (Monte Carlo)

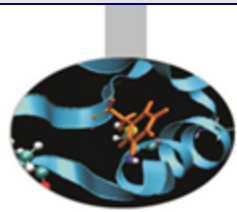


- Estrarre N coppie $Q_i=(x_i,y_i)$ di numeri pseudo casuali uniformemente distribuiti nell'intervallo [0,1]

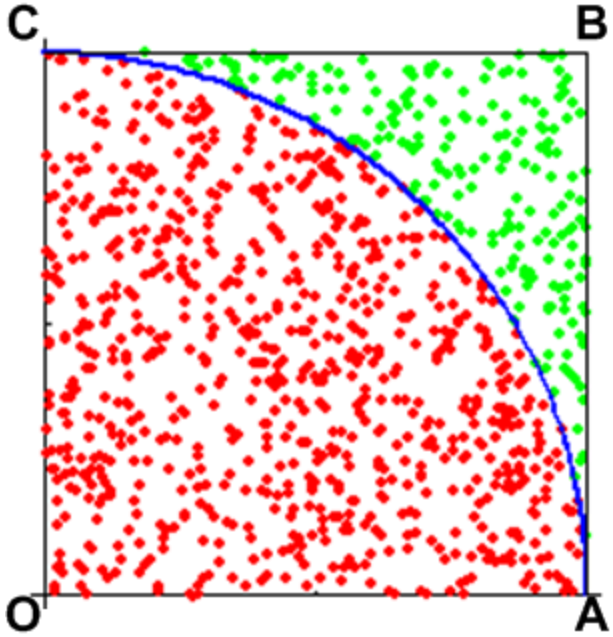


- Per ogni punto Q_i
  - calcolare $d_i = x_i^2 + y_i^2$
  - se $d_i \leq 1$ incrementare il valore di N_c, il numero di punti interni al quadrante AOC
- Il rapporto N_c/N è una stima della probabilità p
- $4*N_c/N$ è una stima di π, con errore dell'ordine 1/sqrt(N)

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

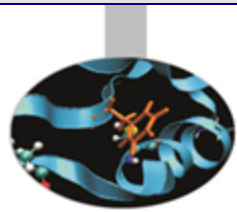


2. Ogni processo con *rank* $\neq 0$ invia al processo di *rank* 0 il valore calcolato di N_c

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 (= $4*N_c/N$)

Communication Ring

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