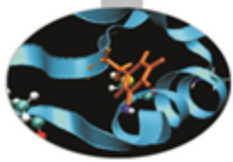
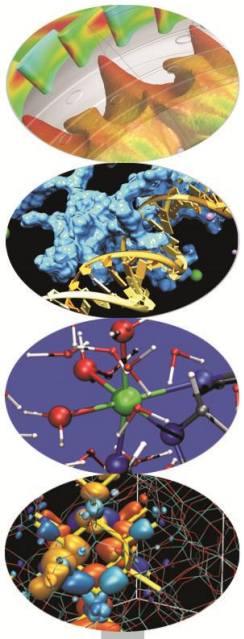
CINECA SCAI
SuperComputing Applications and Innovation



MPI
Laboratorio 1



Isabella Baccarelli
i.baccarelli@cineca.it

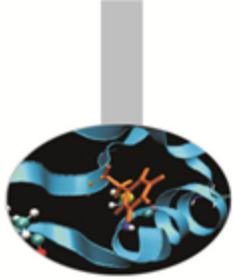
Cristiano Padrin
c.padrin@cineca.it

Mariella Ippolito
m.ippolito@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

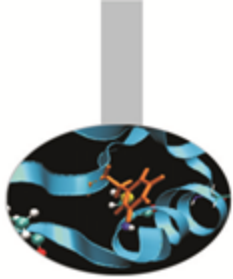
CINECA

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)

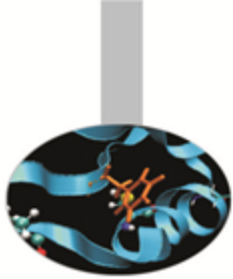
Compilare un sorgente MPI



- 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)



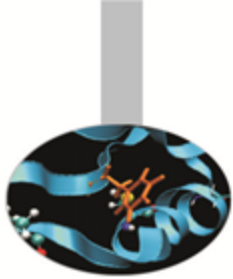
Per compilare il sorgente `sample.f` usando:

- 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
```

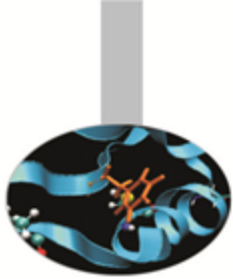
Compilare un sorgente MPI (3)



- … 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

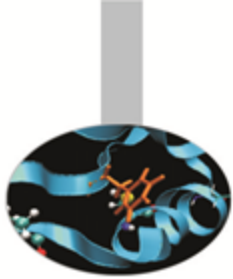


- Per eseguire un programma MPI è necessario lanciare tutti i processi (*process spawn*) con cui si vuole eseguire il calcolo in parallelo

- 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

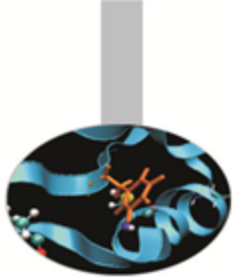


Il nome dei processori che ospiteranno i processi può essere scritto in un file con una specifica sintassi accettata dal *launcher* MPI

- 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



- 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
```

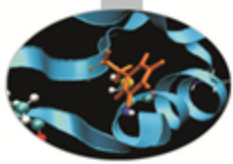
  - C++ source:

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

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

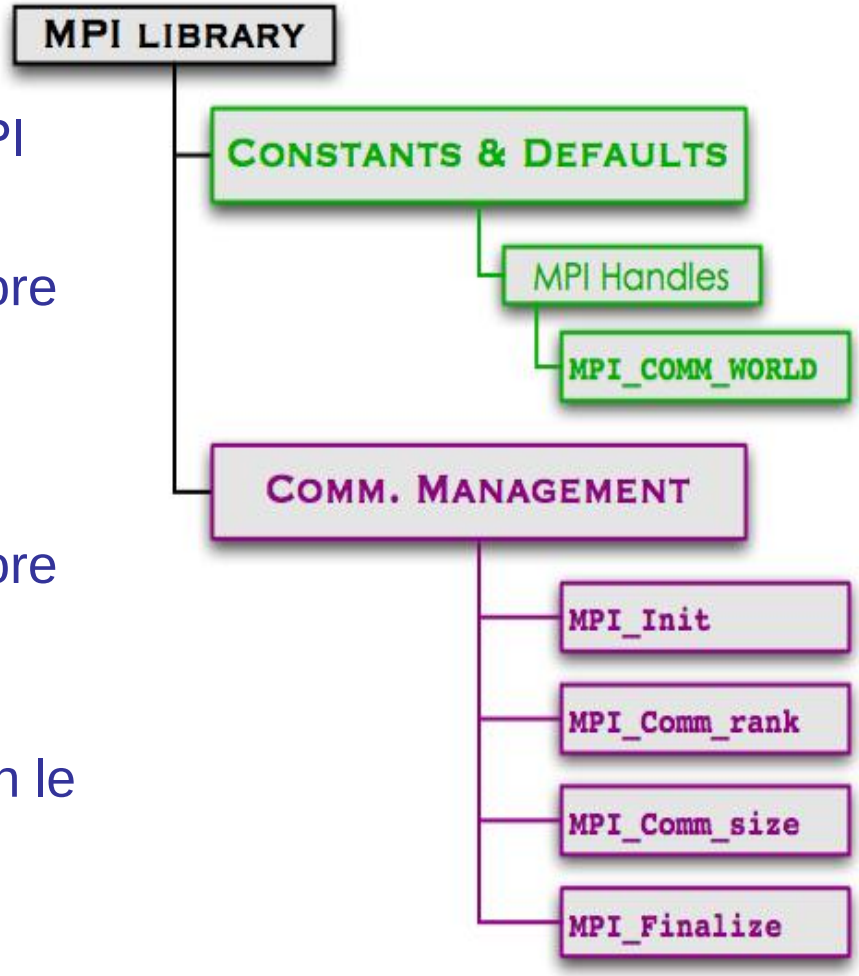
```
mpiexec -hostfile my_hostfile -n 4 ./sample.x
```

Il primo programma MPI

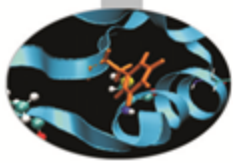


Operazioni da eseguire:

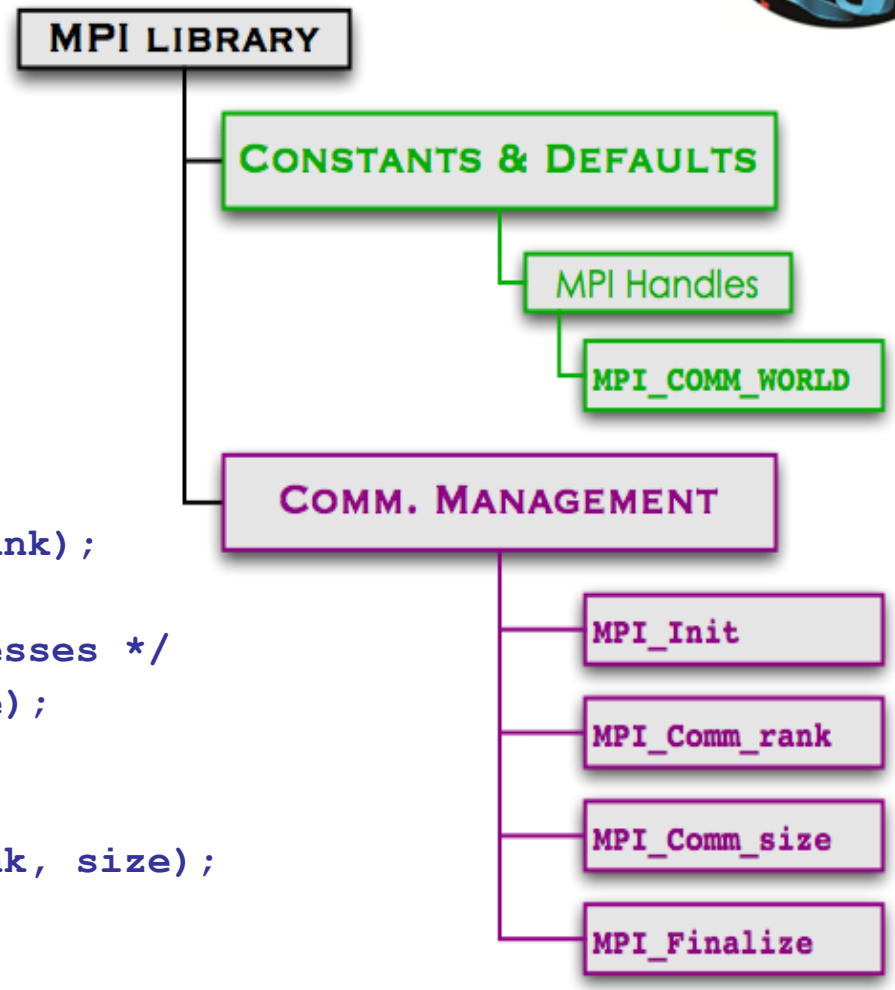
1. Inizializzare l'ambiente MPI
2. Richiedere al comunicatore di default il *rank* del processo
3. Richiedere al comunicatore di default la sua *size*
4. Stampare una stringa con le informazioni ottenute
5. Chiudere l'ambiente MPI



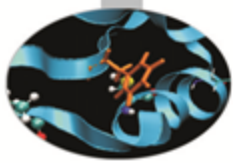
La versione in C …



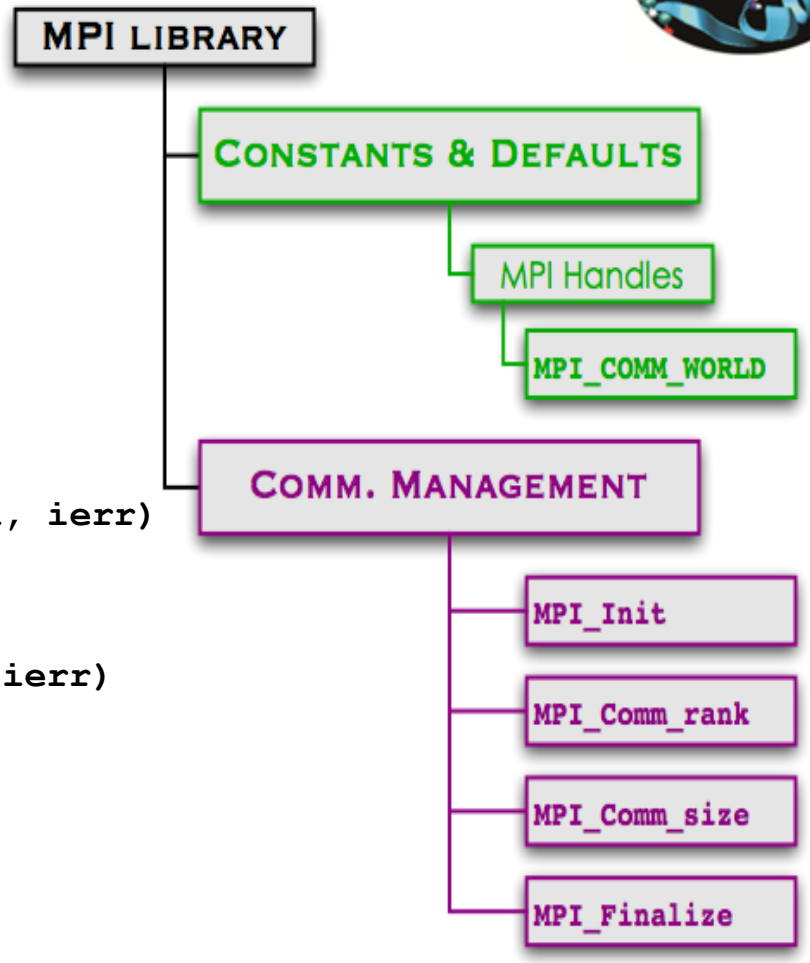
```c
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {

  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();
}
```



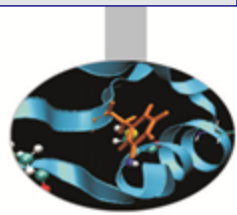
.. e quella in Fortran 77



```
 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
```



MPI Hello World



- Come si compila il codice:
  - In C:

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

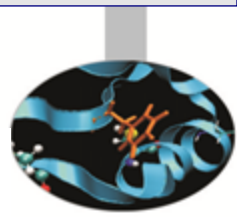
  - In Fortran:

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

- Come si manda in esecuzione utilizzando 4 processi:

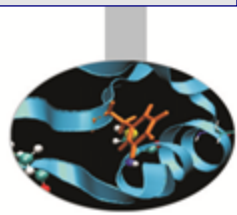
```
mpiexec -n 4 ./hello.x
```

send/receive: intero (C)



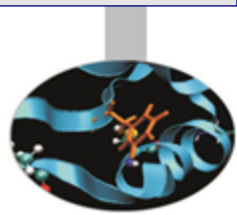
```c
#include <stdio.h>
#include <mpi.h>

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)



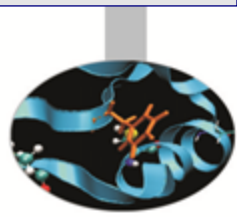
```
      program main

      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)

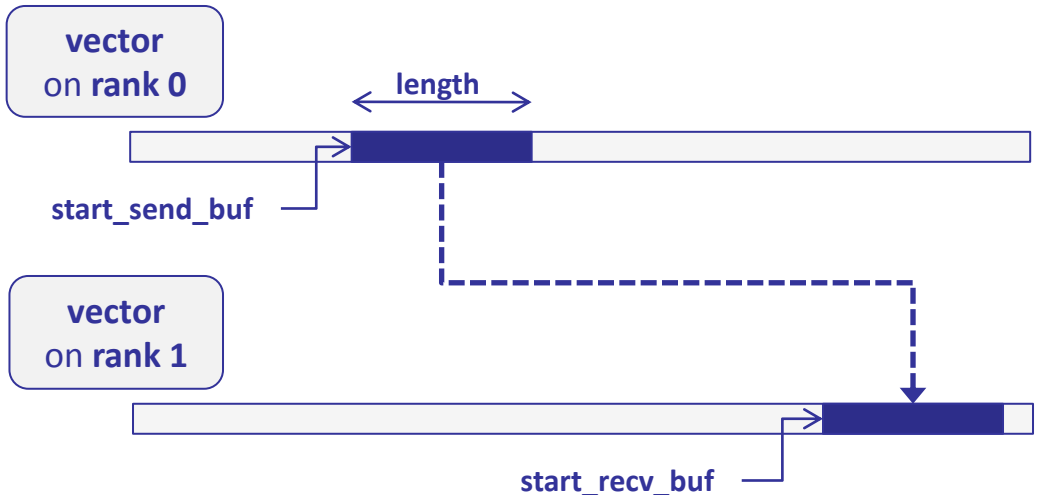


```c
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

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;
}
```

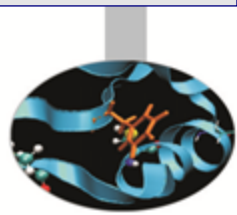
send/receive: porzione di array (C)



```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;
}
```

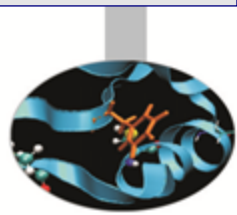


Send/Receive di un intero e di un array



- 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

Calcolo di π con il metodo integrale



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

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

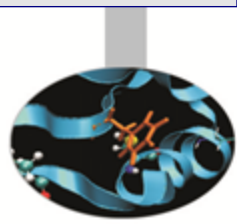
- In generale, se f è integrabile in [a,b]

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

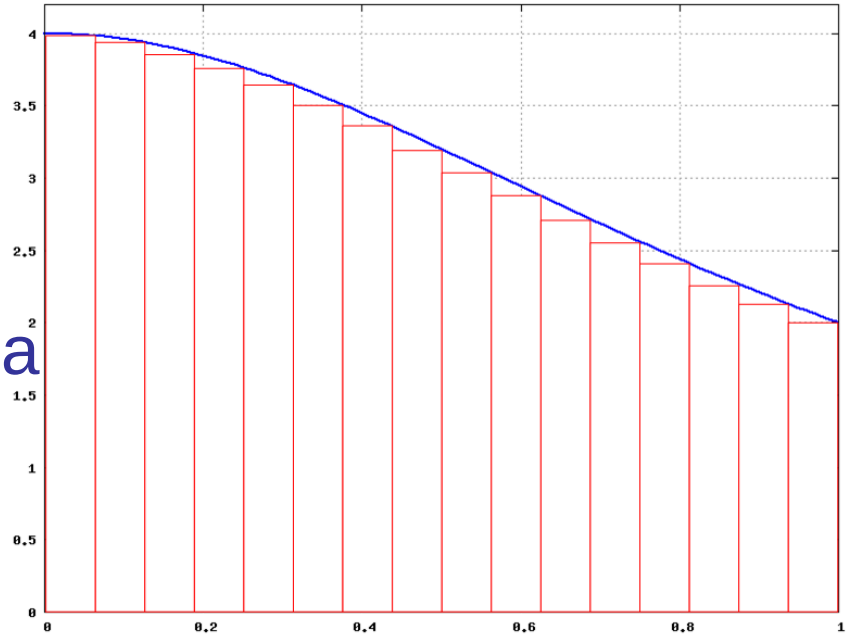
- Dunque, per N sufficientemente grande

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

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

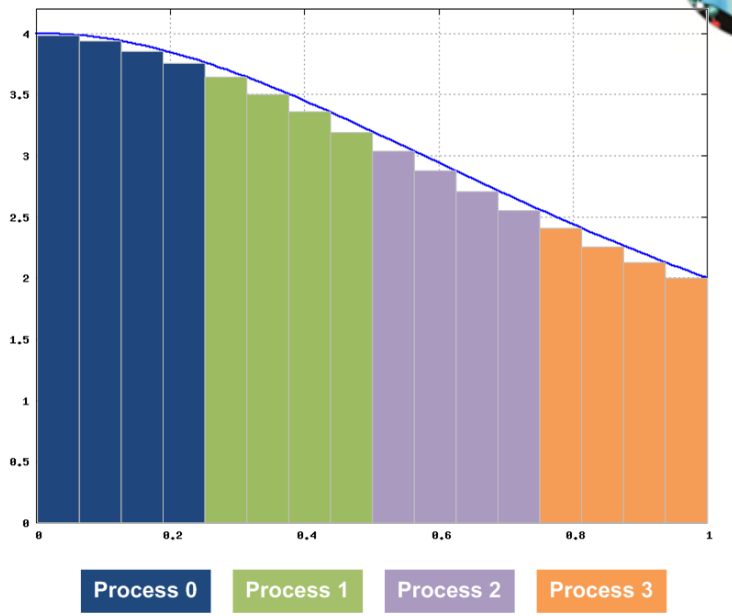
$$\sum_{i=1}^{N} \frac{4 \cdot h}{1+(ih)^2} \quad \text{con} \quad h = \frac{1}{N}$$

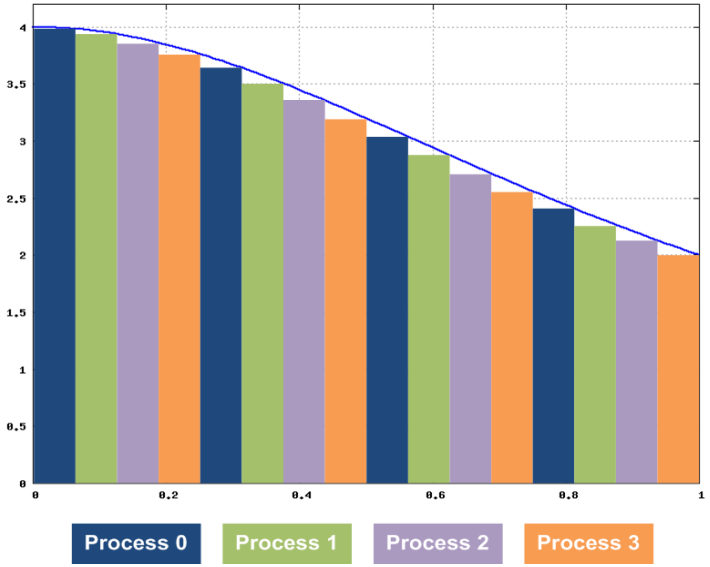
che è uguale alla somma delle aree dei rettangoli in rosso

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

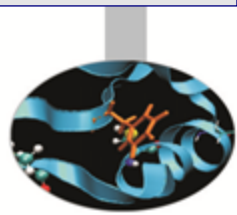
Il Calcolo di π: l'algoritmo parallelo

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

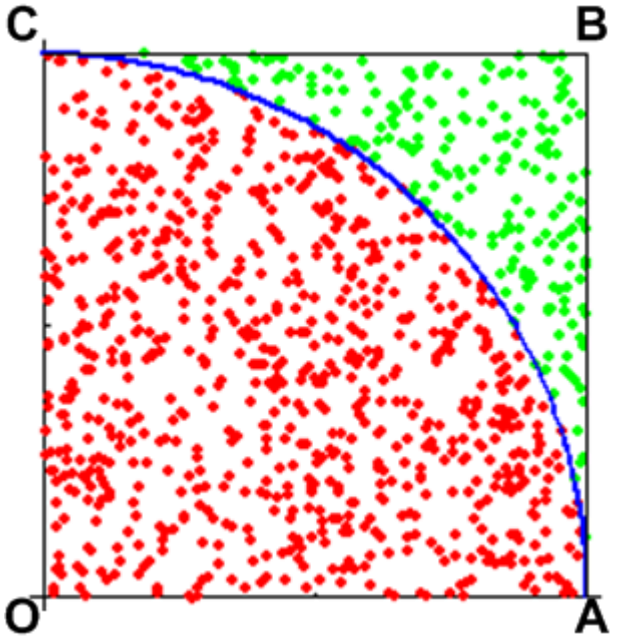




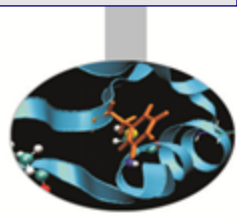
Il Calcolo di π con il metodo Monte Carlo



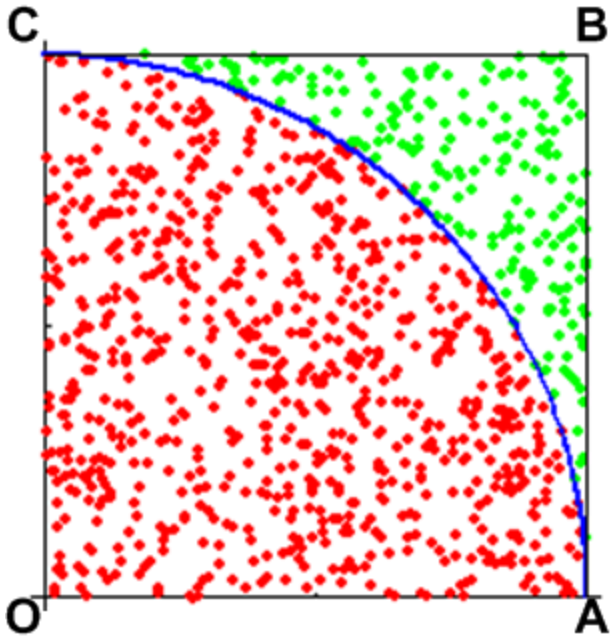
- *AOC* è il quadrante del cerchio unitario, la cui area è $\pi/4$
- Sia $Q = (x,y)$ una coppia di numeri casuali estratti da una distribuzione uniforme in $[0,1]$
- La probabilità p che il punto Q sia interno al quadrante *AOC* è pari al rapporto tra l'area di *AOC* e quella del quadrato *ABCO*, ovvero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



Il Calcolo di π in seriale (Monte Carlo)

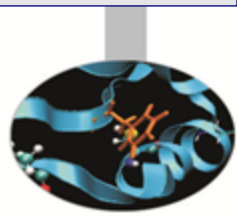


- Estrarre N coppie $Q_i=(x_i,y_i)$ di numeri pseudo casuali uniformemente distribuiti nell'intervallo [0,1]

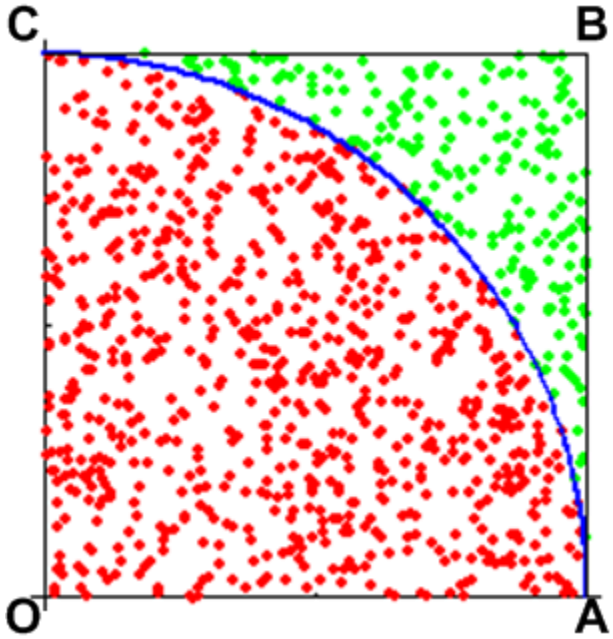


- Per ogni punto Q_i
  - calcolare $d_i = x_i^2 + y_i^2$
  - se $d_i \leq 1$ incrementare il valore di N_c, il numero di punti interni al quadrante AOC
- Il rapporto N_c/N è una stima della probabilità p
- $4*N_c/N$ è una stima di π, con errore dell'ordine 1/sqrt(N)

Il Calcolo di π con P processi (Monte Carlo)



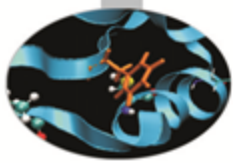
1. Ogni processo estrae N/P coppie $Q_i=(x_i,y_i)$ e calcola il relativo numero N_c di punti interni al quadrante AOC



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

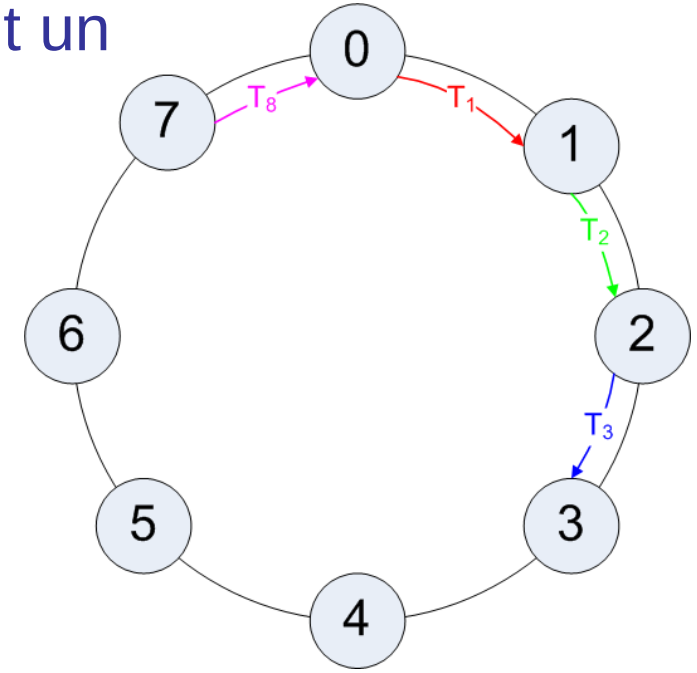
3. Il processo di *rank* 0
   - Riceve i P-1 valori di N_c inviati dagli altri processi
   - Ricostruisce il valore globale di N_c sommando i contributi ricevuti dagli altri processi con quello calcolato localmente
   - Calcola la stima di π (= $4*N_c/N$)

Communication Ring

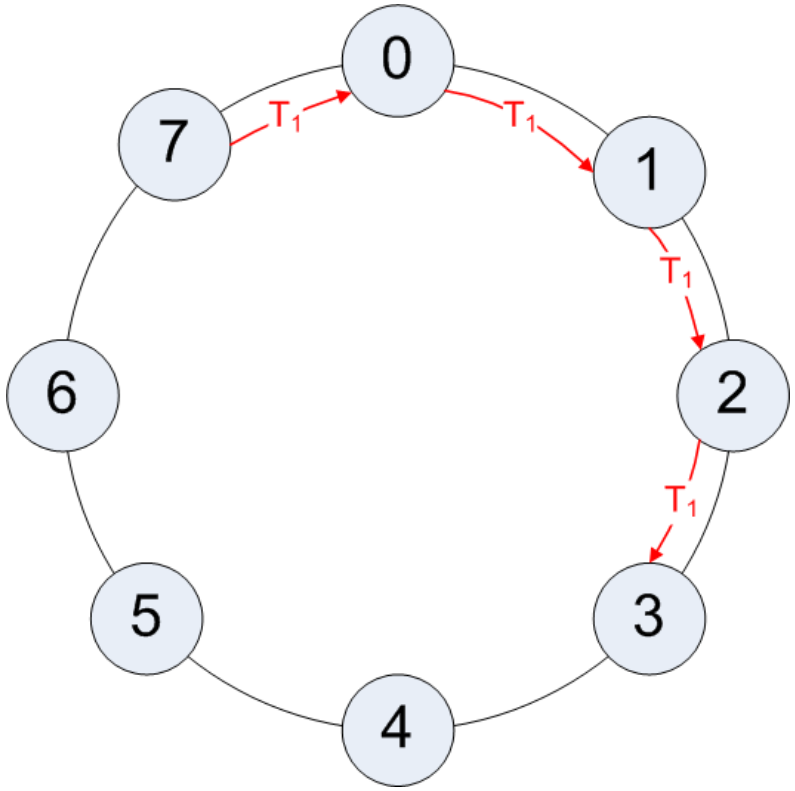


Scrivere un programma MPI in cui

- 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

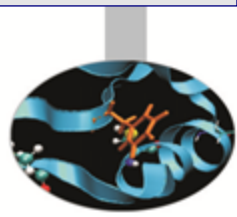


Shift Circolare periodico

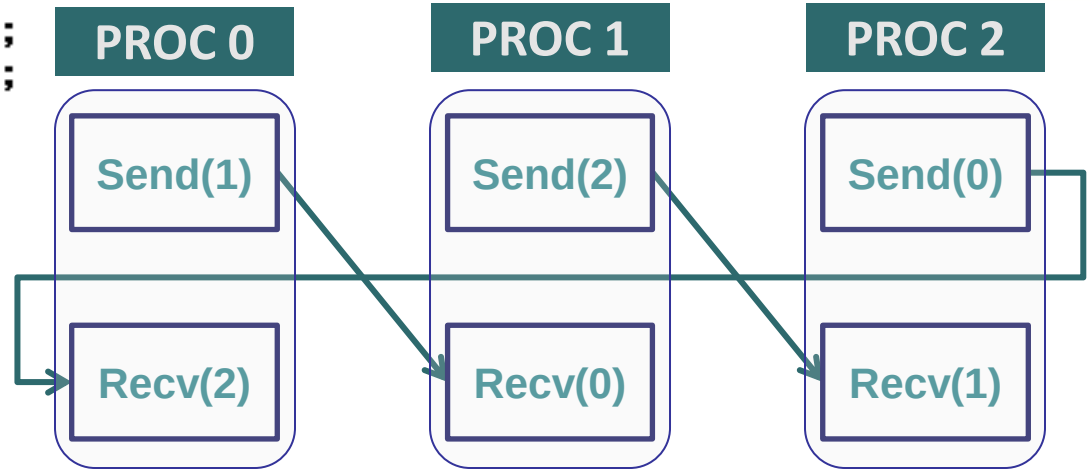


- Ogni processo genera un array `A`, popolandolo con interi pari al proprio rank
- Ogni processo invia il proprio array `A` al processo con rank immediatamente successivo
  - Periodic Boundary: l'ultimo processo invia l'array al primo processo
- Ogni processo riceve l'array `A` dal processo immediatamente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

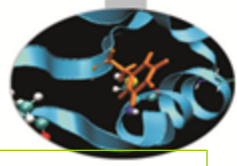
Shift circolare periodico: versione *naive*



```
/* 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*