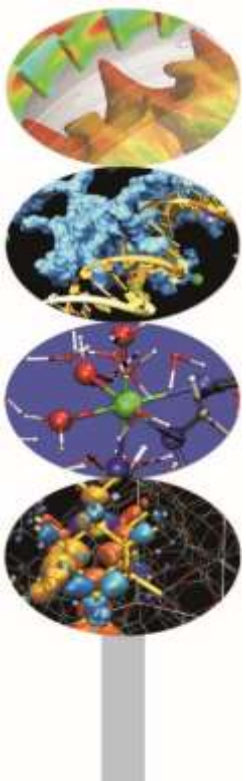
MPI Quick Refresh



Super Computing Applications and Innovation Department

Courses Edition 2017

Cos'è MPI

MPI acronimo di Message Passing Interface

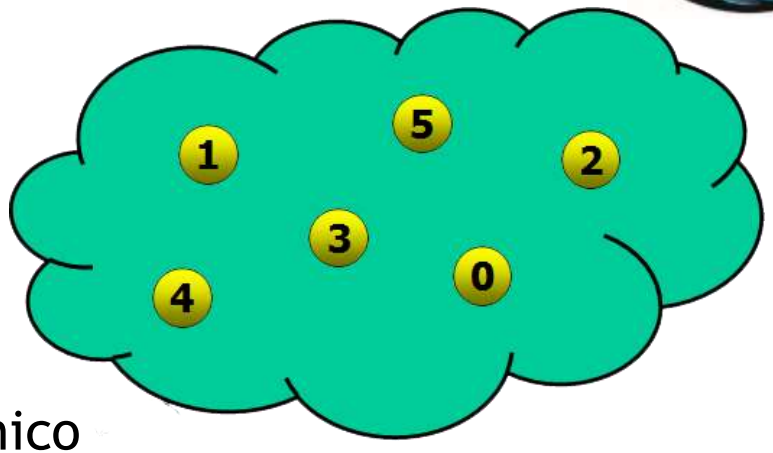
http://www.mpi-forum.org

MPI è una specifica, non un'implementazione

MPI è uno strumento di programmazione che permette di implementare il modello di programmazione message-passing

- controllo dei processi e gruppi di comunicazione (comunicatori)
- pattern di comunicazioni tra processi sincrone e asincrone

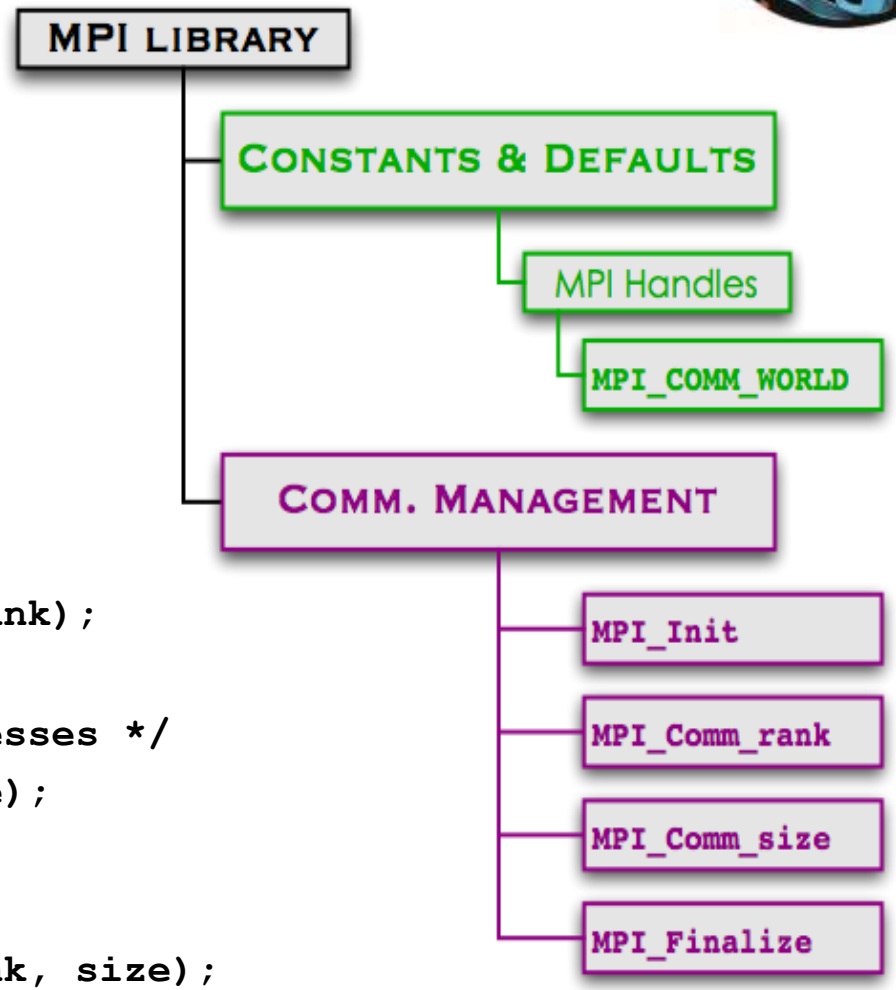
Comunicatori



- 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

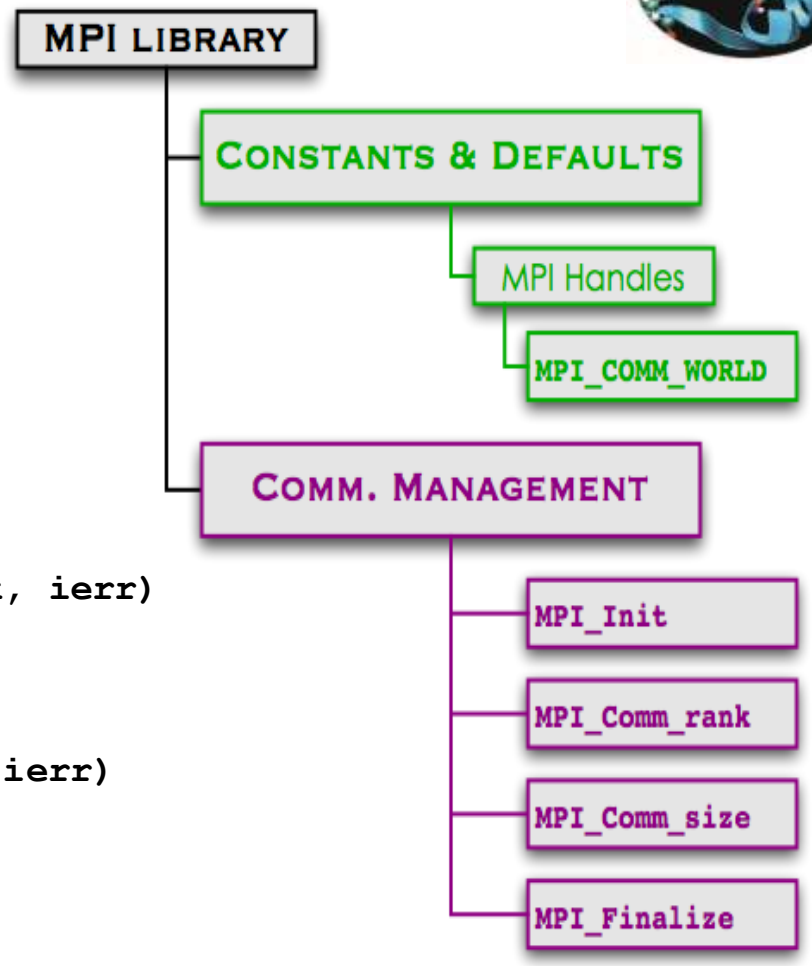
La versione in C …

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

int 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

   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
```



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:

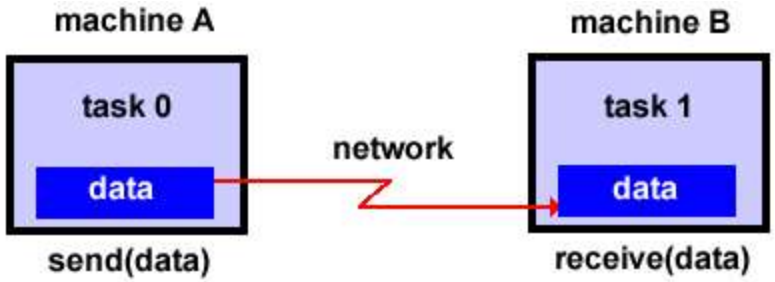
```
call MPI_XXXX(parameter,…, err)
```

  - **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**)

La comunicazione interprocesso

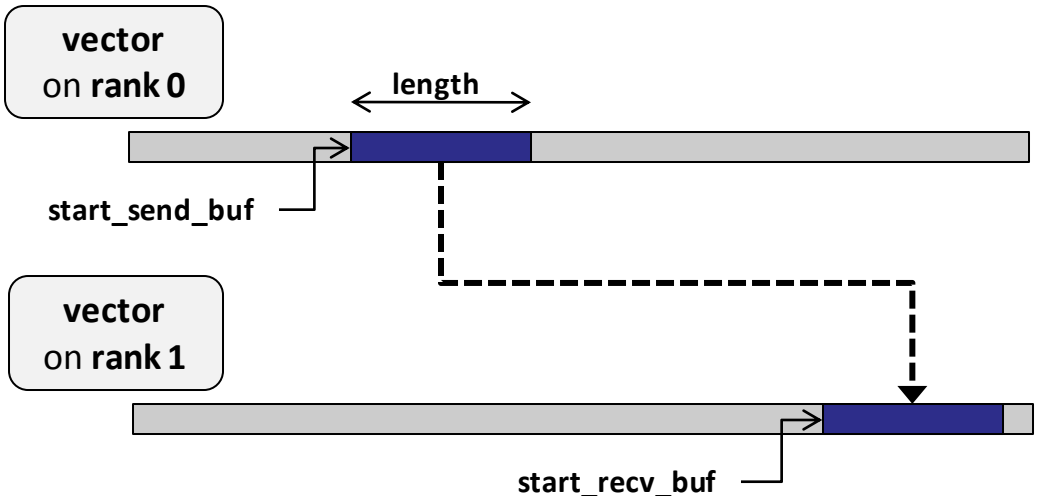
- Nella programmazione parallela ***message passing***
  la cooperazione tra processi avviene attraverso operazioni esplicite di comunicazione interprocesso

- 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



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;
}
```



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

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`

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

- 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]

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

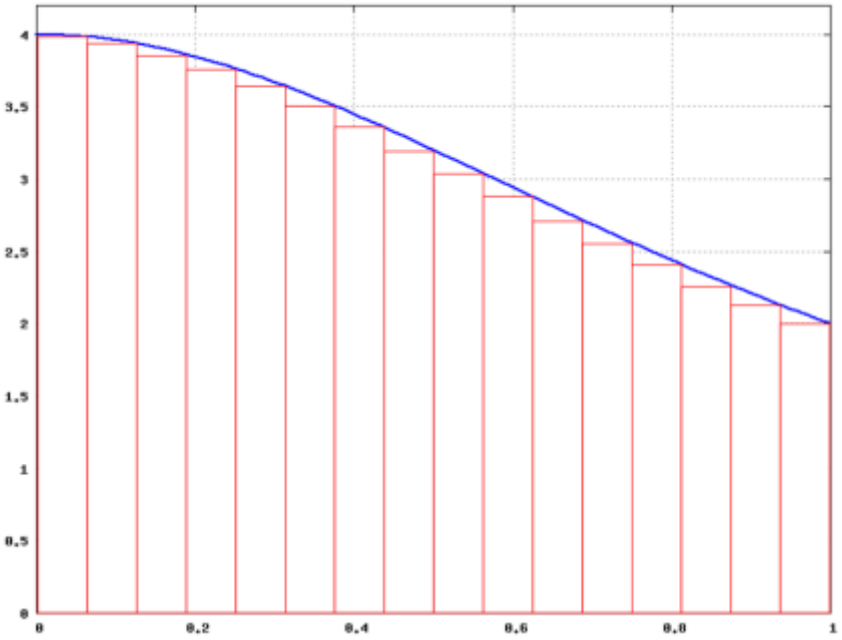
- Dunque, per N sufficientemente grande

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

Calcolo di π in seriale con il metodo integrale

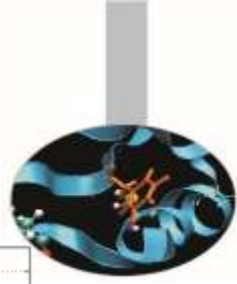
- L'intervallo [0,1] è diviso in N sotto intervalli, di dimensione h=1/N
- L'integrale può essere approssimato con la somma della serie

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

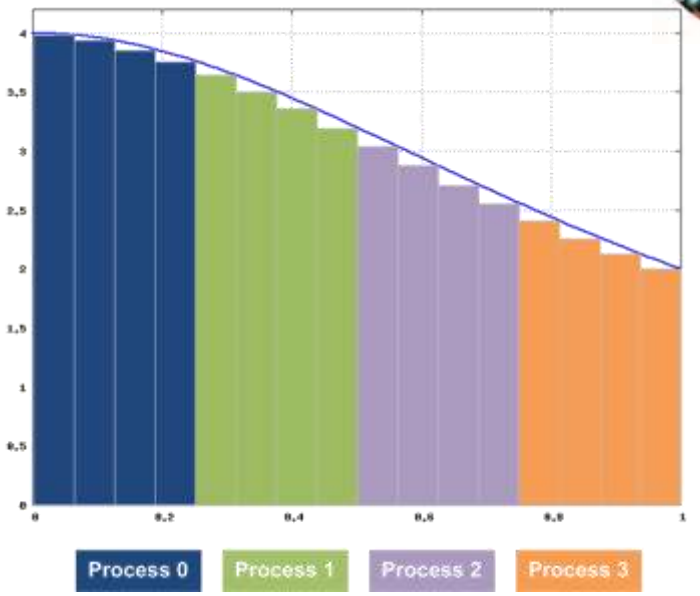


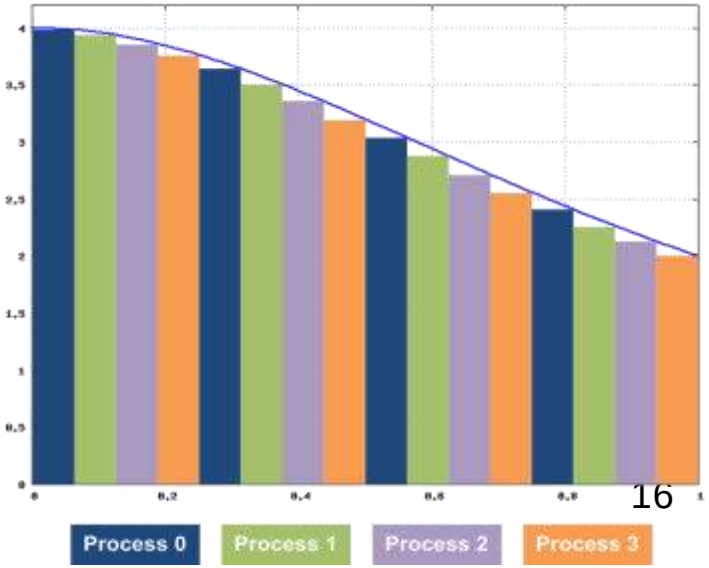
- che è uguale alla somma delle aree dei rettangoli in rosso
- Al crescere di N si ottiene una stima sempre più precisa di π

Il Calcolo di p: l'algoritmo parallelo



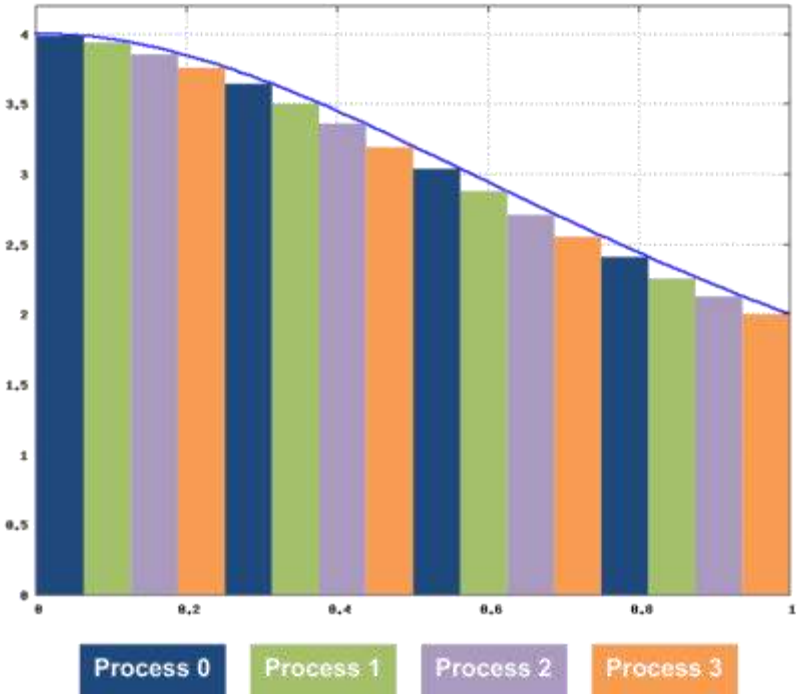
- Ogni processo calcola la somma parziale di propria competenza rispetto alla decomposizione scelta
- Ogni processo con rank ≠ 0 invia al processo di rank 0 la somma parziale calcolata
- Il processo di rank 0
  - Riceve le P-1 somme parziali inviate dagli altri processi



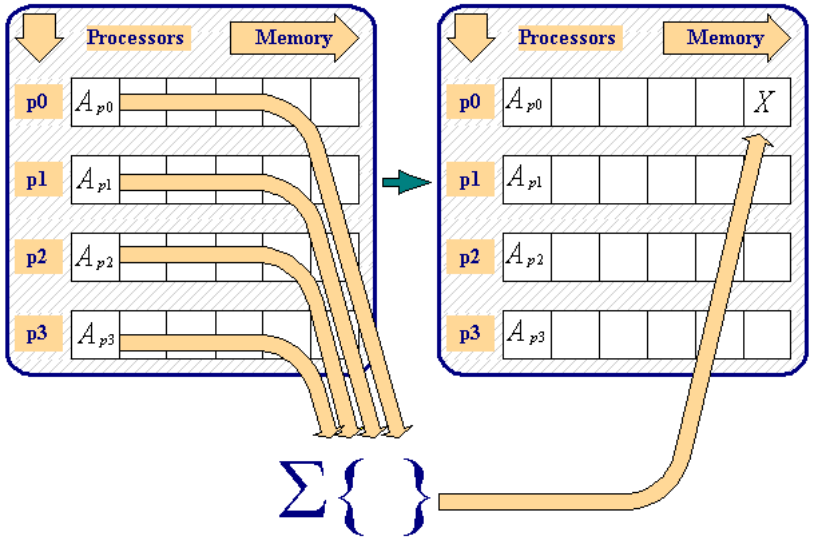


Calcolo di π con *reduce*

```c
#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;
}
```



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)
```

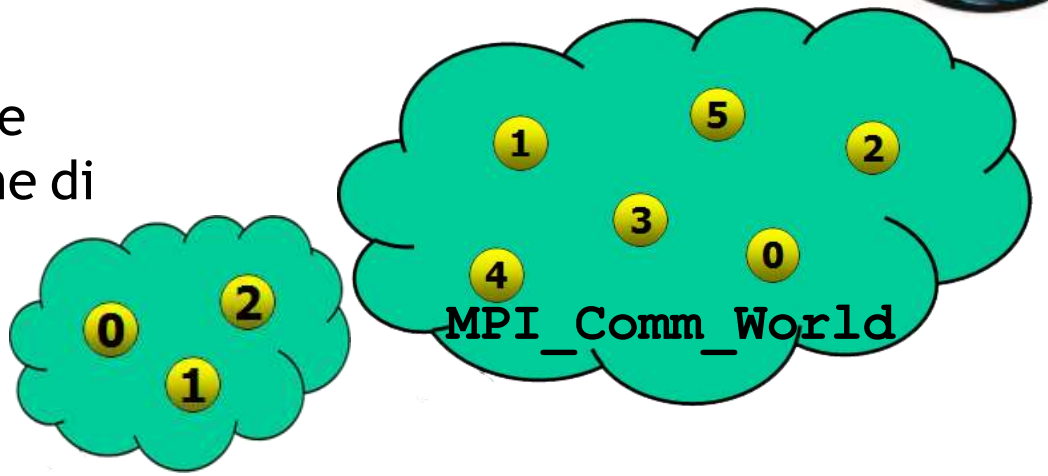
In Fortran

```
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 referenzia 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

Oltre `MPI_Comm_World`: i comunicatori

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



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

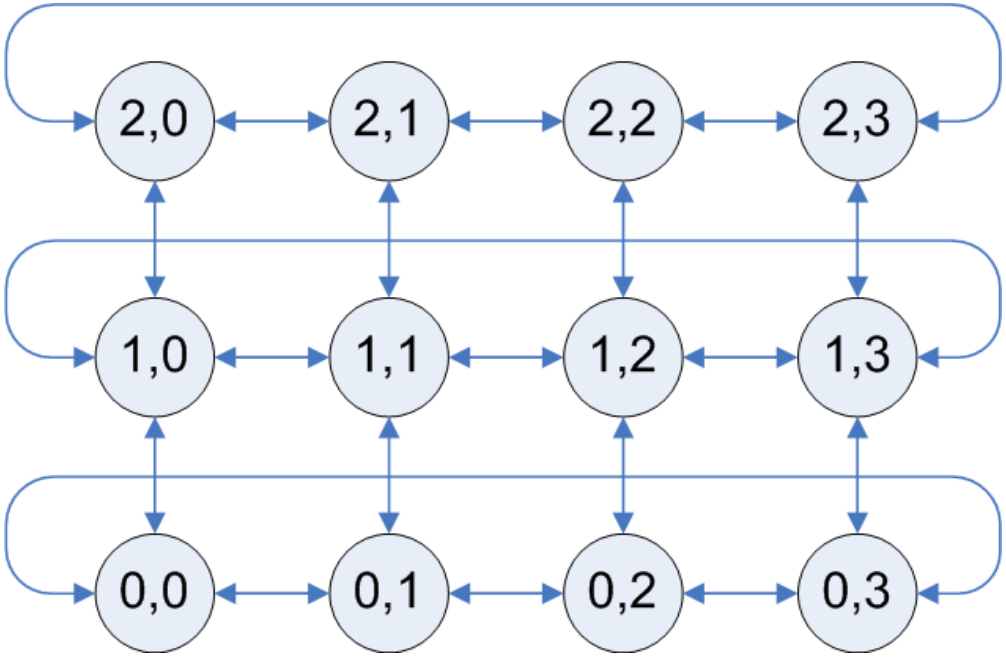
Topologie virtuali di processi

- Definizione di un nuovo schema identificativo dei processi *conveniente* per lavorare con uno specifico pattern di comunicazione:
  - semplifica la scrittura del codice
  - può consentire ad MPI di ottimizzare le comunicazioni

- Creare una topologia virtuale di processi in MPI significa definire un nuovo comunicatore, con attributi specifici

- Tipi di topologie:
  - **Cartesiane**:
    - ogni processo è identificato da un set di coordinate cartesiane ed è connesso ai propri vicini da una griglia virtuale
    - Ai bordi della griglia può essere impostata o meno la periodicità
  - **Grafo** (al di fuori di questo corso)

Topologia cartesiana 2D



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

Creare un comunicatore con topologia cartesiana

In C

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

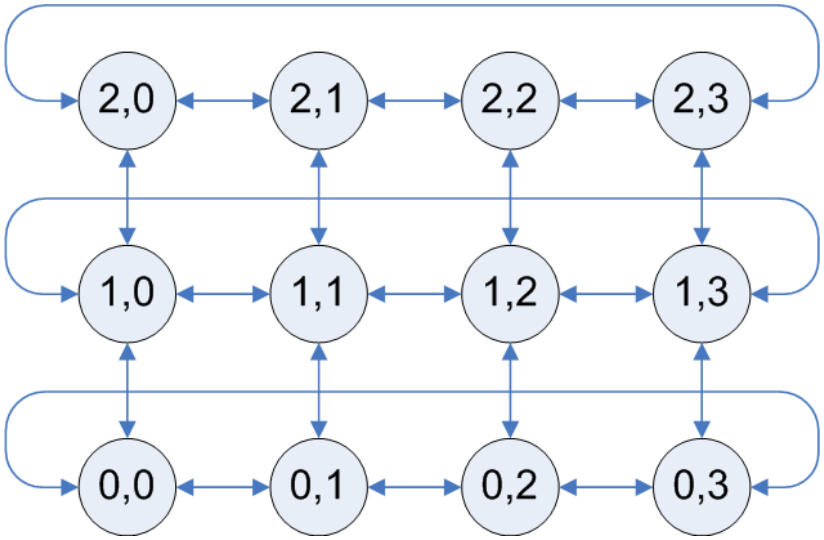
In Fortran

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

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

Come usare MPI_Cart_create

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

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

    ...

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

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

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

    ...

    return 0;
}
```



Alcune utili funzionalità

- **`MPI_Dims_Create`**:
  - Calcola le dimensioni della griglia bilanciata ottimale rispetto al numero di processi e la dimensionalità della griglia dati in input
  - Utile per calcolare un vettore `dims` di input per la funzione `MPI_Cart_Create`

- Mapping tra coordinate cartesiane e *rank*
  - **`MPI_Cart_coords`**: sulla base della topologia definita all'interno del comunicatore, ritorna le coordinate corrispondenti al processo con un fissato *rank*
  - **`MPI_Cart_rank`**: sulla base della topologia definita all'interno del comunicatore, ritorna il *rank* del processo con un fissato set di coordinate cartesiane