

# Programmazione parallela con MPI

# Sviluppo di programmi paralleli

A seconda delle caratteristiche architettureali del sistema utilizzato, nello sviluppo di software parallelo è possibile fare riferimento ai 2 modelli di interazione:

- Se i nodi dell'architettura non condividono memoria (es. Cluster HPC) lo sviluppo dei programmi paralleli si fonda sul **modello a scambio di messaggi**. In questo ambito lo standard è **rappresentato dalle librerie MPI**.
- Se, invece, è prevista la condivisione di memoria tra tutti i nodi (es. sistemi multicore/multiprocessor) il modello di interazione tra processi è a **memoria comune**. In questo caso, è possibile utilizzare **OpenMP**.

# MPI: Message passing Interface

**MPI** è uno standard di fatto, che stabilisce un protocollo per la comunicazione tra processi in sistemi paralleli (MPI\_forum.org).

A partire dal 1992, il protocollo si è sviluppato e diffuso, definendo Language Independent Specifications (LIS) alle quali tutte le realizzazioni si uniformano.

**Concretamente:** Lo standard è implementato in diverse librerie di funzioni C/C++, Fortran, Python per lo sviluppo di programmi paralleli nel modello a scambio di messaggi

## Implementazioni di MPI:

- MPICH
- openMPI
- IntelMPI (→Cineca)
- ecc.

Lo standard garantisce **portabilità** sulle diverse architetture/implementazioni.

# Caratteristiche di MPI:

- è basato sul paradigma **SPMD**: l'esecuzione di una applicazione parallela MPI viene portata avanti da molteplici istanze dello stesso programma, ognuna in esecuzione contemporanea su un nodo distinto. Ogni istanza è rappresentata da un **processo MPI**.
- offre un ricco set di funzioni per esprimere **comunicazione** tra processi sia **punto-punto** che **collettive**, con semantiche sia **sincrone** che **asincrone**.
- offre potenti strumenti per **data partitioning** (distribuzione di dati tra nodi) e **data collecting** (raccolta di risultati).
- **gestione di processi statica e implicita**: il grado di parallelismo viene definito a tempo di caricamento

# Struttura di un programma MPI

```
#include <mpi.h>

main()
{
    ...           // parte sequenziale
    MPI_Init(...);
    <codice con chiamate alla libreria MPI> // parte parallela
    MPI_Finalize();
    ...           // parte sequenziale
}
```

**MPI\_Init** e **MPI\_Finalize** delimitano la parte del programma che verrà eseguita su più nodi in parallelo.

👉 Fuori dal blocco MPI\_Init/MPI\_Finalize non possono essere chiamate funzioni MPI.

# MPI\_Init e MPI\_Finalize

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

i parametri possono puntare ad argc e argv del main, oppure NULL.

```
int MPI_Finalize(void) ;
```

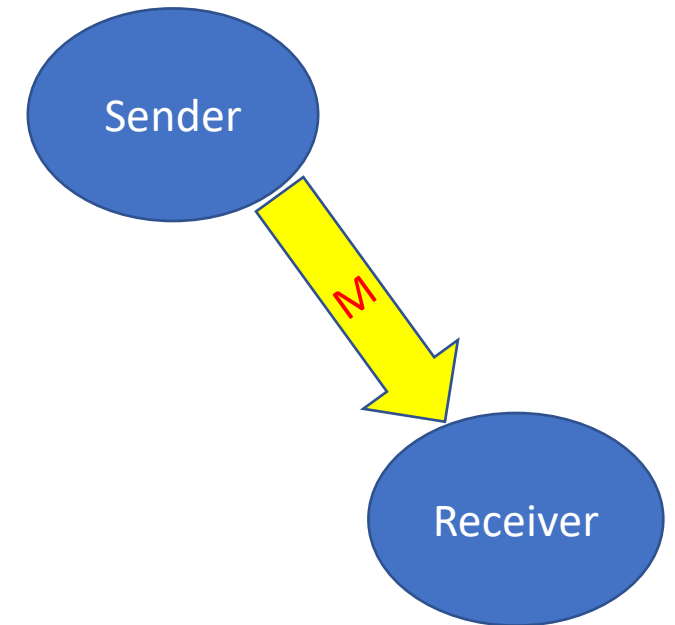
In entrambi i casi il valore restituito, in caso di successo è MPI\_SUCCESS.

# Struttura programma MPI

MPI adotta il modello **SPMD**: ogni processo esegue lo stesso programma su un nodo diverso; per differenziarne il comportamento si usa il **conditional branching**.

```
// provacomm.c
#include <mpi.h>

main()
{
    ...           // parte sequenziale
    MPI_Init(NULL, NULL);
    if (<sono il Sender>) //branching
        <invia messaggio M a Receiver>
    else // receiver
        <ricevi il messaggio M>
    MPI_Finalize();
    ...           // parte sequenziale
}
```



# Gestione dei processi

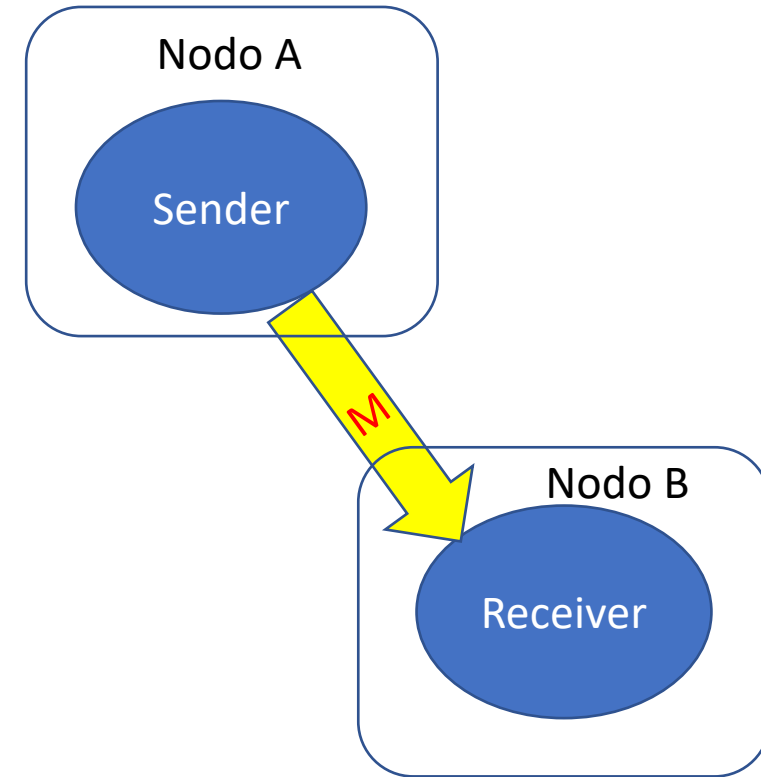
## Compilazione:

```
$mpicc -o provacomm provacomm.c
```

## Esecuzione:

```
$mpirun -n 2 provacomm
```

→ vengono lanciate **2 istanze** del programma su **due nodi** distinti (-n 2) .





# Communicator

Il **communicator** è un'astrazione che definisce un **dominio di comunicazione**, ovvero un insieme di processi che possono comunicare tra loro → **due processi possono scambiarsi messaggi se e solo se appartengono allo stesso communicator.**

**Communicator MPI\_COMM\_WORLD** : Per ogni programma che usa MPI, esiste il **communicator di default MPI\_COMM\_WORLD** che viene automaticamente creato e al quale appartengono tutti i processi creati con **mpirun**.

A partire da MPI\_COMM\_WORLD, è possibile creare nuovi communicator. Ad esempio, si vedano le funzioni:

- **MPI\_Comm\_create(...)**
- **MPI\_Comm\_spawn(..)**

# Esempio

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

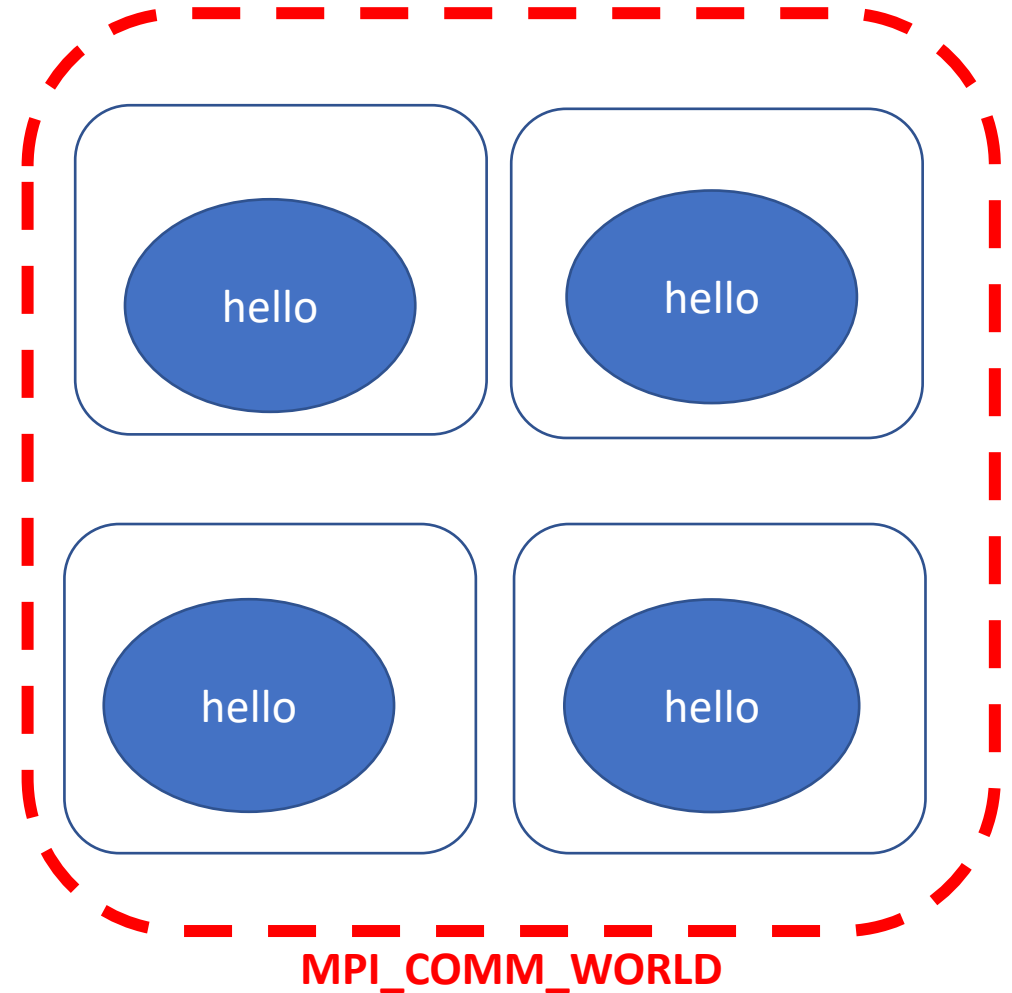
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    printf("Ciao Mondo!\n");
    MPI_Finalize();
}
```

## Compilazione:

```
$mpicc -o hello hello.c
```

## Esecuzione:

```
$mpirun -n 4 hello
```



# Communicator: size e rank

**Dimensione:** E' possibile conoscere la **dimensione** di un communicator con la funzione:

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

assegna a **\*size** il numero dei componenti del gruppo **comm**.

**Rank:** Ogni processo può ottenere il suo id (**rank**) all'interno di un communicator di cui fa parte con la funzione:

```
int MPI_Comm_rank(MPI_Comm Comm, int* rank) ;
```

assegna a **\*rank** il l'ID del processo che invoca la funzione all'interno del communicator **Comm**.

Restituiscono la costante **MPI\_SUCCESS** in caso di successo, altrimenti un codice d'errore.

# Communicator size e rank

## Esempio:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
// esecomm.c
int main(int argc, char* argv[])
{
    int comm_size, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if(my_rank == 0) // branching
        printf("[processo %d] Ci sono %d processi in MPI_COMM_WORLD.\n",
            my_rank, comm_size);
    else
        printf("[processo %d] ciao!\n", my_rank);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

# Communicator

## Esempio:

```
bash-3.2$ mpicc -o esecomm esecomm.c
```

```
bash-3.2$ mpirun -n 5 esecomm
```

```
[processo 0] Ci sono 5 processi in MPI_COMM_WORLD.
```

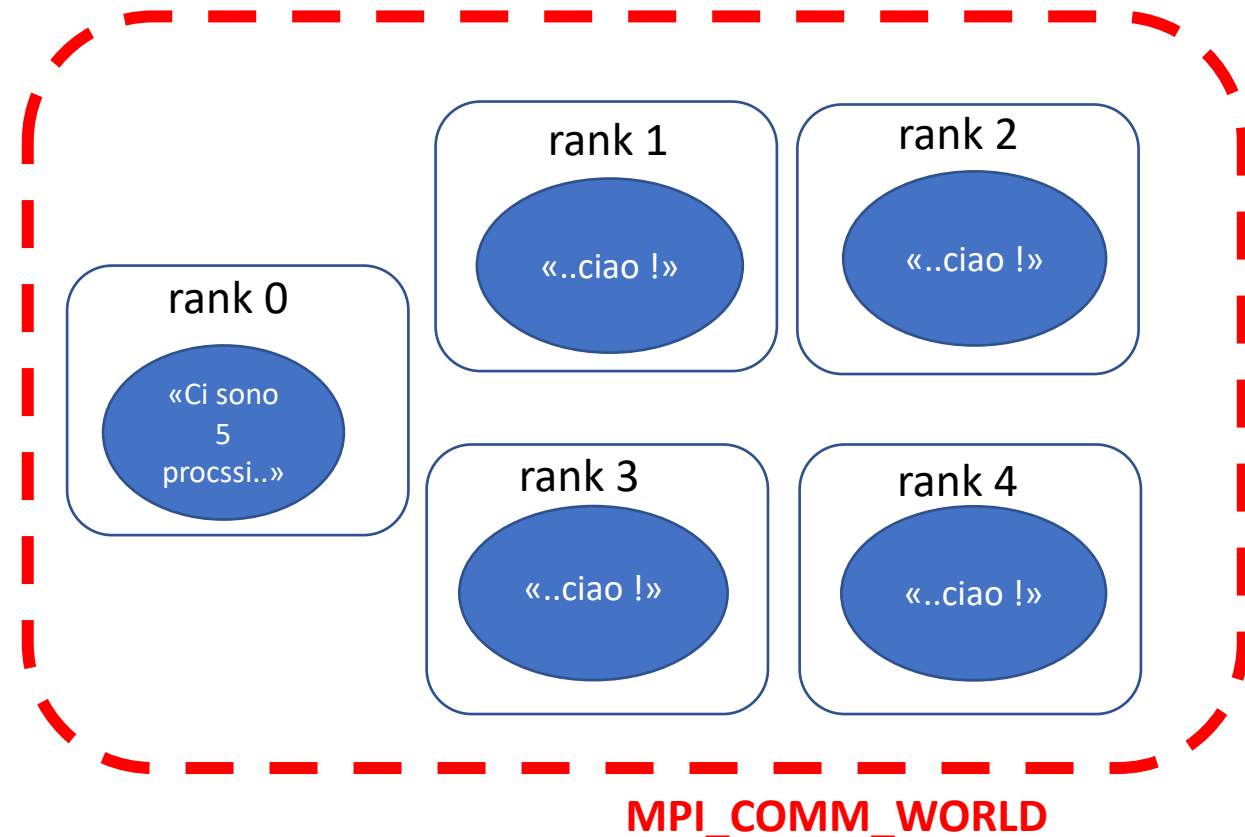
```
[processo 1] ciao!
```

```
[processo 2] ciao!
```

```
[processo 3] ciao!
```

```
[processo 4] ciao!
```

```
bash-3.2$
```



# MPI: funzioni di comunicazione

Lo standard MPI definisce un'ampia scelta di primitive di comunicazione:

- diversi schemi di comunicazione (**singole/collettive**)
- diverse semantiche (**sincrone/asincrone**)

## Primitive singole vs. collettive

**Primitive di comunicazione singole:** consentono l'invio di messaggi da 1 mittente a 1 destinatario; implementano lo schema «uno a uno».

Esempio: **MPI\_send**, **MPI\_recv**

**Primitive di comunicazione collettive:** mittenti e destinatari possono essere molteplici;

Ad esempio: distribuzione/raccolta di dati/risultati verso/da un insieme di processi «workers»

# Comunicazione singola: MPI\_Send

## Sintassi:

```
int MPI_Send(const void* buffer,  
             int count,  
             MPI_Datatype datatype,  
             int dest,  
             int tag,  
             MPI_Comm comm) ;
```

### MPI\_Datatype:

MPI\_INT  
MPI\_CHAR  
MPI\_FLOAT  
MPI\_DOUBLE  
ecc...

## 6 parametri:

- **buffer** è l'indirizzo del buffer contenente il messaggio.
- **count** è il numero di elementi del buffer da inviare.
- **datatype** è il tipo del singolo elemento del buffer (v. tipi ammessi in MPI\_Datatype).
- **dest**: è il rank of del processo destinatario.
- **tag**: l'etichetta (tag) assegnata alla comunicazione.
- **comm**: il communicator nel quale avviene la comunicazione.

# MPI\_Send: semantica

```
MPI_Send(buffer, count, datatype, dest, tag, comm);
```

invia i **count** elementi di **buffer** al destinatario di rank **dest**, all'interno del communicator **comm** con etichetta **tag**.

## Sincronizzazione:

- **MPI\_Send** può essere sia **sincrona** che **asincrona**, dipendentemente dall'implementazione. La primitiva è «**buffer-safe**» ovvero: lo standard stabilisce che **MPI\_Send** **blocchi il processo sender** almeno fino a quando il messaggio non viene prelevato dal **buffer** per essere inviato al destinatario; pertanto, al termine della **MPI\_Send** si può eventualmente riutilizzare il buffer in maniera «safe».



# MPI\_Send: varianti

- Se si desidera una semantica **sincrona**:

**MPI\_Ssend(buffer, count, datatype, dest, tag, comm) ;**

👉 Il mittente attende fino a che il destinatario non ha ricevuto il messaggio (MPI\_Recv).  
MPI\_SSend è buffer-safe.

- Per una semantica **asincrona**:

**MPI\_Isend(buffer, count, datatype, dest, tag, comm) ;**

👉 la chiamata ritorna immediatamente. (ImmediateSend).

👉 NB: **MPI\_Isend non è buffer-safe**: il messaggio rimane nel buffer fino a che i meccanismi di comunicazione non lo prelevano per recapitarlo al destinatario. Per testare la disponibilità del buffer dopo una MPI\_Isend, si può usare **MPI\_Wait** (sospensiva) oppure **MPI\_Test**.

In alternativa: **MPI\_Bsend**: send asincrona buffer-safe. Può comportare attesa da parte del mittente.

# Comunicazione singola: MPI\_Recv

## Sintassi:

```
int MPI_Recv(void* buffer,  
             int count,  
             MPI_Datatype datatype,  
             int sender,  
             int tag,  
             MPI_Comm communicator,  
             MPI_Status* status);
```

### MPI\_Datatype:

MPI\_INT  
MPI\_CHAR  
MPI\_FLOAT  
MPI\_DOUBLE  
ecc...

## 7 Parametri:

- **buffer** è l'indirizzo del buffer dove verrà memorizzato il messaggio.
- **count** è il numero di elementi nel buffer ricevuto.
- **datatype** è il tipo del singolo elemento del buffer (v. tipi ammessi in MPI\_Datatype).
- **sender**: è il rank del processo mittente. è possibile il naming esplicito del mittente, oppure specificare la costante MPI\_ANY\_SOURCE
- **tag**: l'etichetta (tag) assegnata alla comunicazione. E' possibile specificare il tag indicato dal mittente nell'invio (receive selettiva) oppure MPI\_ANY\_TAG
- **communicator**: il gruppo di processi (communicator) nel quale avviene la comunicazione.
- **status**: indirizzo della variabile in cui vengono salvati gli attributi del messaggio.

# MPI\_Recv: semantica

```
MPI_Recv(buffer, count, datatype, sender, tag, comm, &stato);
```

riceve i **count** elementi di **buffer** del tipo **datatype** dal sender di rank **sender**, con etichetta **tag** all'interno del communicator **comm**.

In **stato** vengono memorizzate informazioni relative al messaggio ricevuto; in particolare:

**stato.MPI\_SOURCE** contiene il **rank** del mittente

**stato.MPI\_TAG** contiene il **tag** associato al messaggio

**stato.MPI\_ERROR** contiene il **codice d'errore** generato dalla ricezione (se 0, successo).

.

## Sincronizzazione:

- **MPI\_Recv** è **bloccante**: il processo attende che il messaggio gli venga recapitato.

In alternativa:

**int MPI\_Irecv(..)** non è **bloccante**: se il messaggio non è arrivato, ritorna immediatamente.

# MPI\_Recv: wildcards

Attraverso i parametri **sender** e **tag** è possibile esprimere una **ricezione selettiva**.

Ad esempio:

```
MPI_Recv(buffer, 10, MPI_INT, 0, 5, MPI_COMM_WORLD, status);
```

riceve un messaggio solo se proviene dal processo di rank 0 ed ha tag uguale a 5.

In alternativa, per i parametri sender e tag è possibile usare le wildcards:

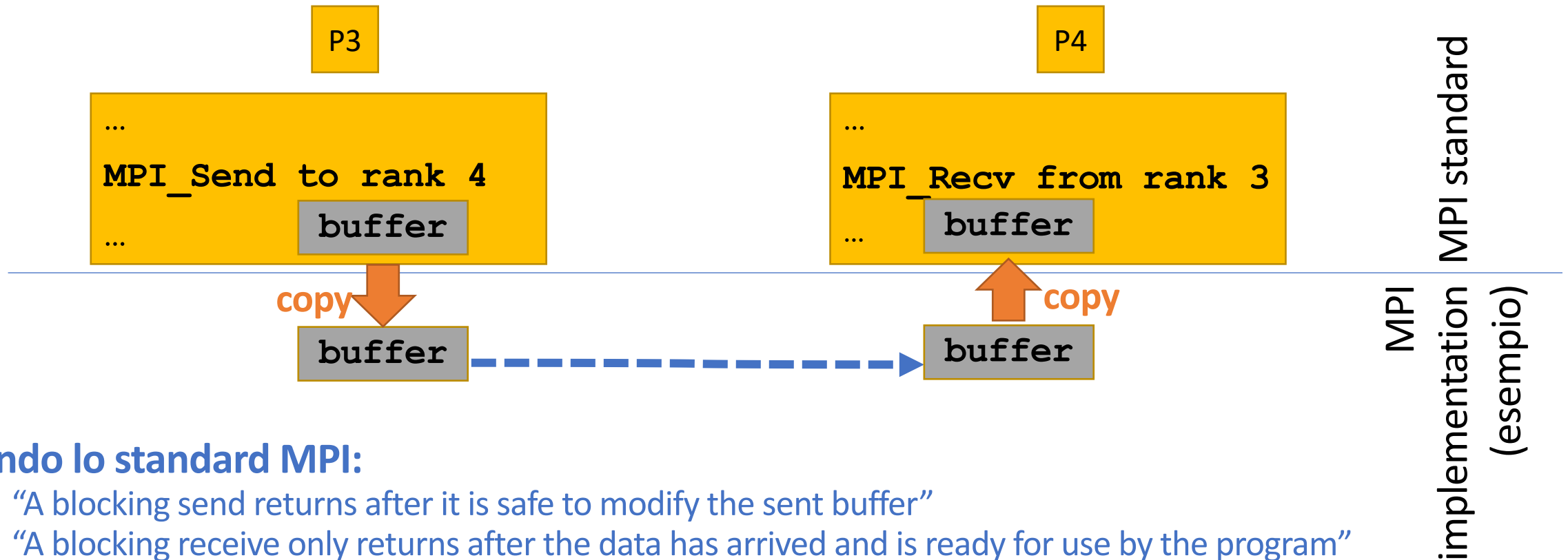
- **MPI\_ANYSOURCE** indica qualunque sender -> schema molti a uno
- **MPI\_ANYTAG** indica qualunque tag

**Esempio:**

```
MPI_Recv(buff, count, T, MPI_ANYSOURCE, MPI_ANYTAG, MPI_COMM_WORLD, status);
```

riceve un messaggio che proviene da qualunque sender e con qualunque tag.

# Semantica MPI\_Send e MPI\_Recv



## Secondo lo standard MPI:

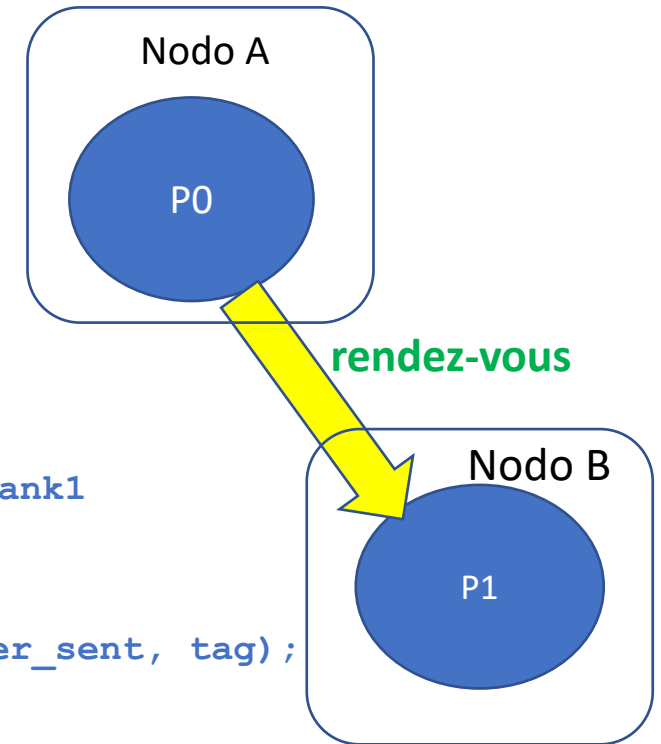
- “A blocking send returns after it is safe to modify the sent buffer”
- “A blocking receive only returns after the data has arrived and is ready for use by the program”

## In pratica:

- la MPI\_Send **non è necessariamente sincrona** (→ per send sincrona usare MPI\_Ssend !)
- la MPI\_Recv è sempre **bloccante**

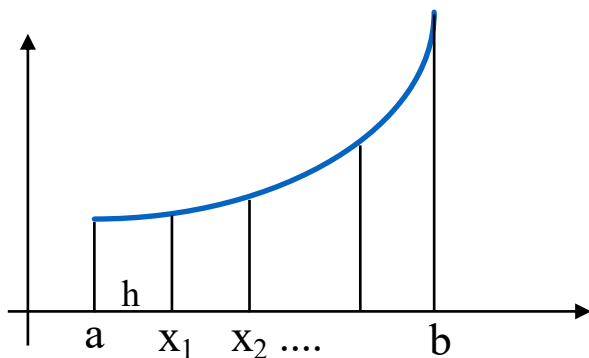
# Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{ int my_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if(my_rank == 0) // "master" MPI process: invia un messaggio al processo di rank1
  { int buffer_sent = 12345;
    int tag = 67890;
    printf("processo %d: inviato messaggio %d con tag %d.\n", my_rank, buffer_sent, tag);
    MPI_Ssend(&buffer_sent, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  }
  else if (my_rank==1) // "slave" MPI process: riceve il messaggio
  { int buffer_received;
    MPI_Status status;
    MPI_Recv(&buffer_received, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("processo %d: ricevuto messaggio %d dal processo %d, con tag %d e error code %d.\n",
          my_rank, buffer_received, status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```



# Esempio: calcolo dell'integrale di una funzione

- Si vuole calcolare  $I = \int_a^b f(x)dx$
- In ogni soluzione numerica :



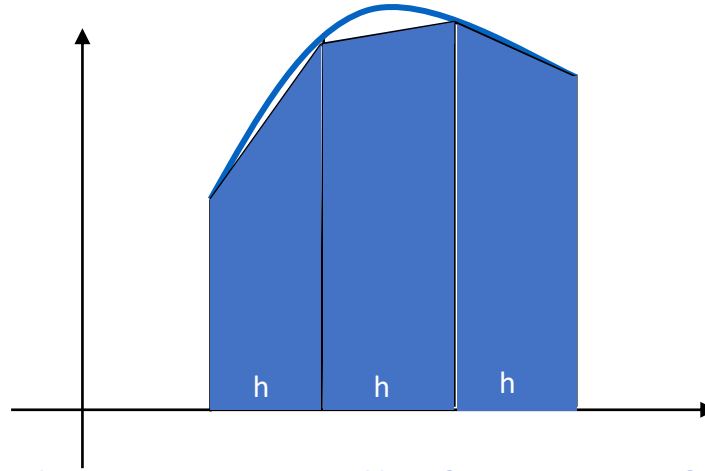
- Si suddivide l'intervallo  $(a,b)$  in  **$n$**  sub-intervalli di ampiezza  **$h$** :  
 **$h=(b-a)/n$**
- Si ottiene la successione di valori:  
 $X_0 = a, X_1, X_2, \dots, X_n = b$   
dove  $X_i = X_{i-1} + h$  per  $i=1,2,\dots,n$
- In ogni intervallo  $i$ -simo  $(X_{i-1}, X_i)$  si approssima  $f(X)$  con una funzione  $F_i(X)$ :

$$I \approx \sum_{i=1}^n \int_{x_{i-1}}^{x_i} F_i(x) dx$$

- Ogni metodo numerico per il calcolo dell'integrale adotta una funzione approssimante  $F(X)$  specifica.

# Metodo dei trapezi (Bezout):

$F_i(X)$  = retta passante per i punti di coordinate  $(X_{i-1}, f(X_{i-1}))$ ,  $(X_i, f(X_i))$



Si approssima l'i-sima areola sottesa dalla funzione  $f$  con il trapezio ottenuto collegando i due punti estremi con una retta:

$$F_i(X) = f(X_{i-1}) + ((f(X_{i-1}) - f(X_i)) / h) \cdot (X - X_{i-1})$$

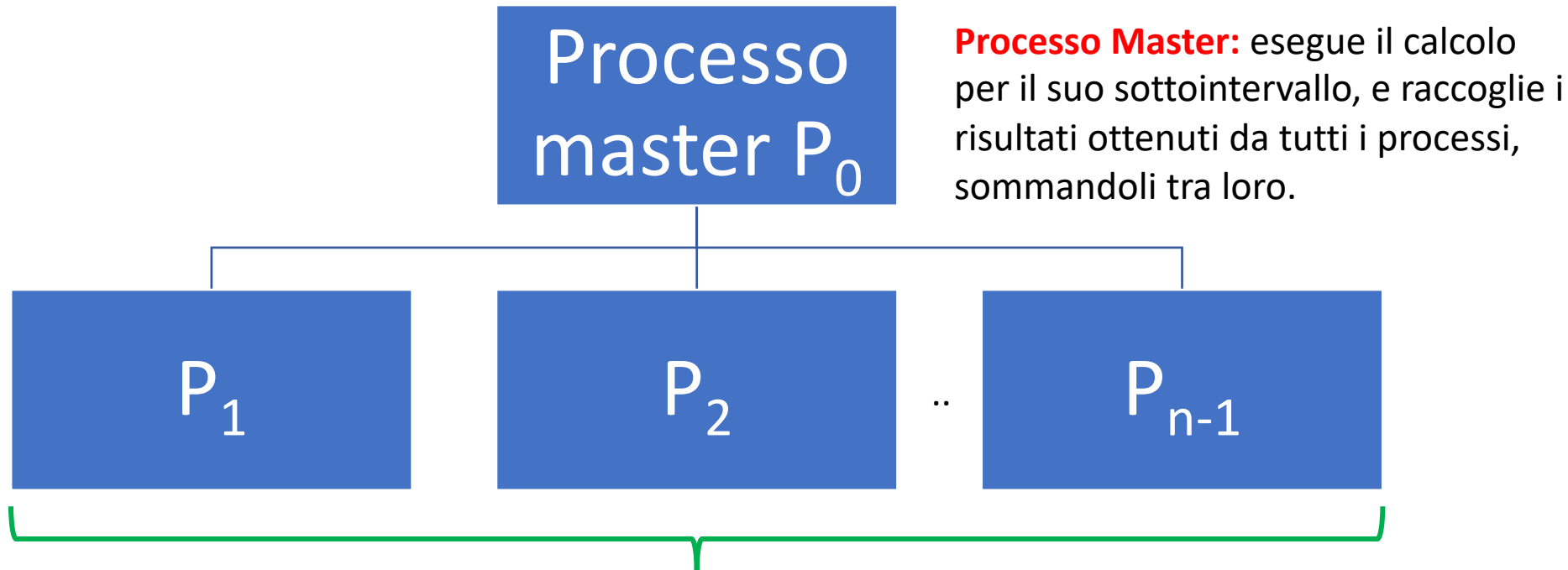
$$I_i \approx \frac{h}{2} \cdot (f(X_i) + f(X_{i-1})) \quad i = 1, 2, \dots, n$$

$$\begin{aligned} \Rightarrow I &\approx \frac{h}{2} \cdot (f(X_0) + f(X_1)) + \frac{h}{2} \cdot (f(X_1) + f(X_2)) + \dots = \\ &= h \cdot \left( \frac{f(X_0) + f(X_n)}{2} + \sum_{i=1}^{n-1} f(X_i) \right) \end{aligned}$$



# Calcolo dell'integrale: soluzione parallela

Con **n nodi**: distribuiamo il lavoro tra  $n$  processi  $P_0, \dots, P_{n-1}$ . Suddividiamo l'intervallo dato in  $n$  sottointervalli, uno per processo. Ogni processo calcola l'integrale (somma dei trapezi) per un sottointervallo diverso. Il processo di rank 0 ( $P_0$ ) si occupa di comporre tutti i risultati parziali nel risultato finale.



**Processi Slave:** ognuno calcola la somma delle aree dei trapezi contenuti in un sottointervallo dato; al termine ogni slave comunicherà al Master il risultato ottenuto.

# Metodo dei Trapezi: Algoritmo parallelo in MPI

```
#include <stdio.h>

#include <mpi.h>

double f(double x) // funzione integranda (es: quadrato di x)
{
    return x*x;
}

// funzione che calcola l'integrale di f nell'intervallo [left_endpt, right_endpt]:
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len)
{
    double estimate, x;
    int i;

    estimate = (f(left_endpt) + f(right_endpt))/2.0;
    for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i * base_len;
        estimate += f(x);
    }

    estimate = estimate*base_len;
    return estimate;
}
```

# Metodo dei Trapezi: Algoritmo parallelo in MPI

```
int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    h = (b - a)/n;
    local_n = n/comm_sz;
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h); // calcolo integrale
                                                    //nel sottointervallo

    if(my_rank!=0){ // slave
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,MPI_COMM_WORLD);
    }
}
```

# Metodo dei Trapezi: Algoritmo parallelo in MPI

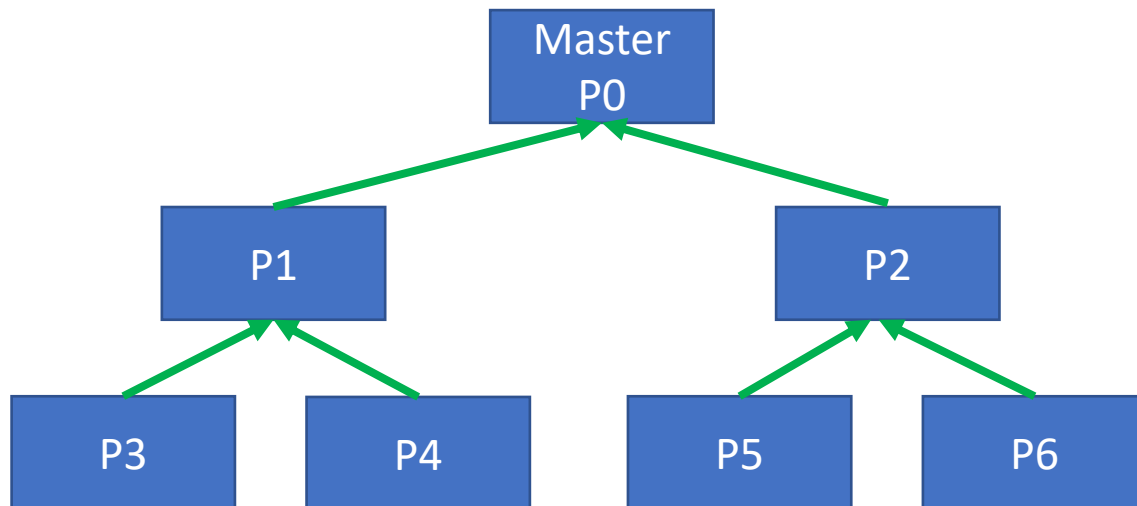
```
else { // rank=0: master: raccoglie i risultati
    total_int = local_int;
    for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_int += local_int;
    }
}
if (my_rank == 0)
{
    printf("Con n = %d trapezi, il risultato dell'\n", n);
    printf("integrale da %f a %f = %.15e\n", a, b, total_int);
}
MPI_Finalize();
return 0;
}
```

# Calcolo dell'integrale: osservazioni

**Comunicazione asimmetrica slave->master.** La soluzione è **centralizzata**: al crescere del numero dei nodi il master potrebbe rappresentare un **collo di bottiglia**, in quanto deve eseguire tante MPI\_Recv quanti sono i nodi.

- Per mitigare il problema, si potrebbe distribuire il carico di comunicazione tra più nodi, utilizzando, invece che comunicazioni punto-punto tra ogni slave ed il master, degli schemi di comunicazione gerarchici che coinvolgano tutti i nodi.

**Ad esempio:** schema di comunicazione ad albero



ogni nodo dell'albero riceve messaggi dai nodi figli e manda un messaggio «cumulativo» al padre.  
In questo modo il master viene alleggerito.

# Distribuzione del carico tra i nodi

La soluzione proposta è efficace e scalabile, ma aggiunge complessità al lavoro del programmatore.

👉 MPI offre una soluzione trasparente per il programmatore: le **primitive di comunicazione collettive**. Sono funzioni per la **comunicazione asimmetrica** implementate in modo tale da distribuire il carico di comunicazione e di calcolo tra i nodo coinvolti.

# Primitive di comunicazione collettiva

La gamma di funzioni MPI che consentono di esprimere comunicazioni collettive è ampia.

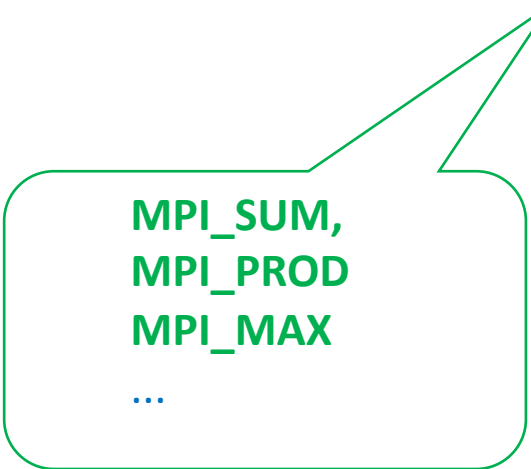
## Principali primitive:

- **MPI\_Reduce()** → multi-a-uno
- **MPI\_Bcast()** → uno-a-molti
- **MPI\_Scatter()** → uno-a-molti
- **MPI\_Gather()** → multi-a-uno
- **MPI\_AllGather()** → multi-a-molti

# MPI\_Reduce

Nell'esempio del calcolo dell'integrale, il master ha il ruolo di **collettore**, cioè deve accumulare tutti i risultati ricevuti componendoli in un'unica somma. Questa operazione può essere realizzata con la funzione **MPI\_Reduce**:

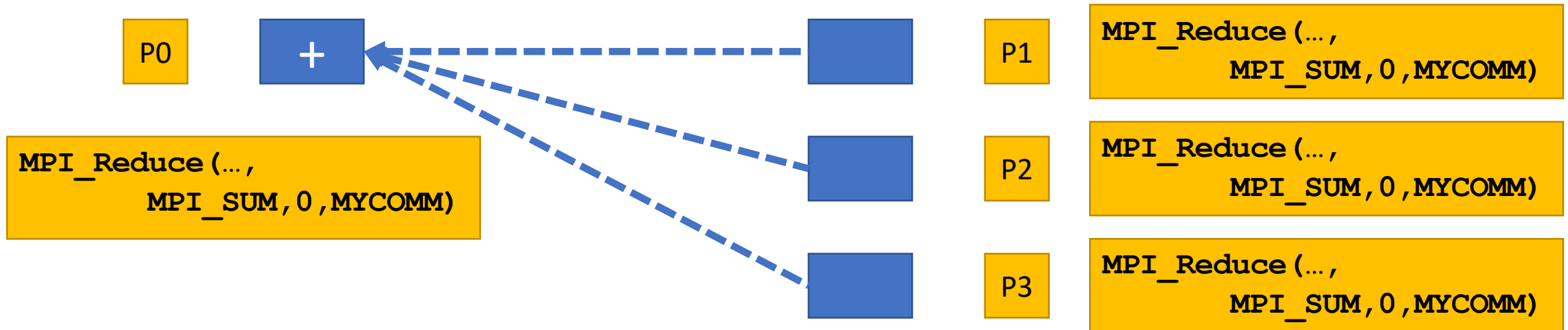
```
int MPI_Reduce(const void* send_buffer,    // dato in ingresso
               void* receive_buffer,      // risultato
               int count,                  // dimensione dato di ingresso
               MPI_Datatype datatype,      // tipo dato in ingresso
               MPI_Op operation,           // operatore di composizione
               int root,                   // rank del collettore
               MPI_Comm comm);             // communicator
```



MPI\_SUM,  
MPI\_PROD  
MPI\_MAX  
...



# MPI\_Reduce



Ogni processo chiama MPI\_Reduce:

- ogni processo del communicator (compreso P0) invia un valore
- il collettore (P0) riceve i valori e compone (es. somma:MPI\_SUM)

Semantica di ricezione **bloccante**.

# MPI\_Reduce: esempio trapezi

```
#include <stdio.h>
#include <mpi.h>
double f(double x)
{
    return x*x; }

double Trap(double left_endpt, double right_endpt, int trap_count, double base_len){...}

int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    h = (b - a)/n;
    local_n = n/comm_sz;
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (my_rank == 0)
    {
        printf("Con n = %d trapezi, il risultato dell'\n", n);
        printf("integrale da %f a %f = %.15e\n", a, b, total_int);
    }
    MPI_Finalize();
    return 0;
}
```

operazione

rank collettore

tutti i processi eseguono  
la MPI\_Reduce:

- il collettore riceve e aggrega i dati in total\_int
- tutti gli altri processi inviano il proprio risultato (local\_int).

Implementazione  
scalabile

# MPI\_Reduce: varianti

## MPI\_Ireduce:

Rende la ricezione non bloccante; per la verifica successiva sul completamento della primitiva: MPI\_Wait/MPI\_Test.

## MPI\_AllReduce:

Si può usare quando tutti i processi necessitano del risultato:

- tutti i processi inviano
- tutti ricevono e compongono il risultato

Carico di comunicazione più elevato, anche se l'implementazione è ottimizzata.

# MPI\_Bcast

Se uno stesso dato deve essere distribuito a tutti i processi si può usare la funzione:

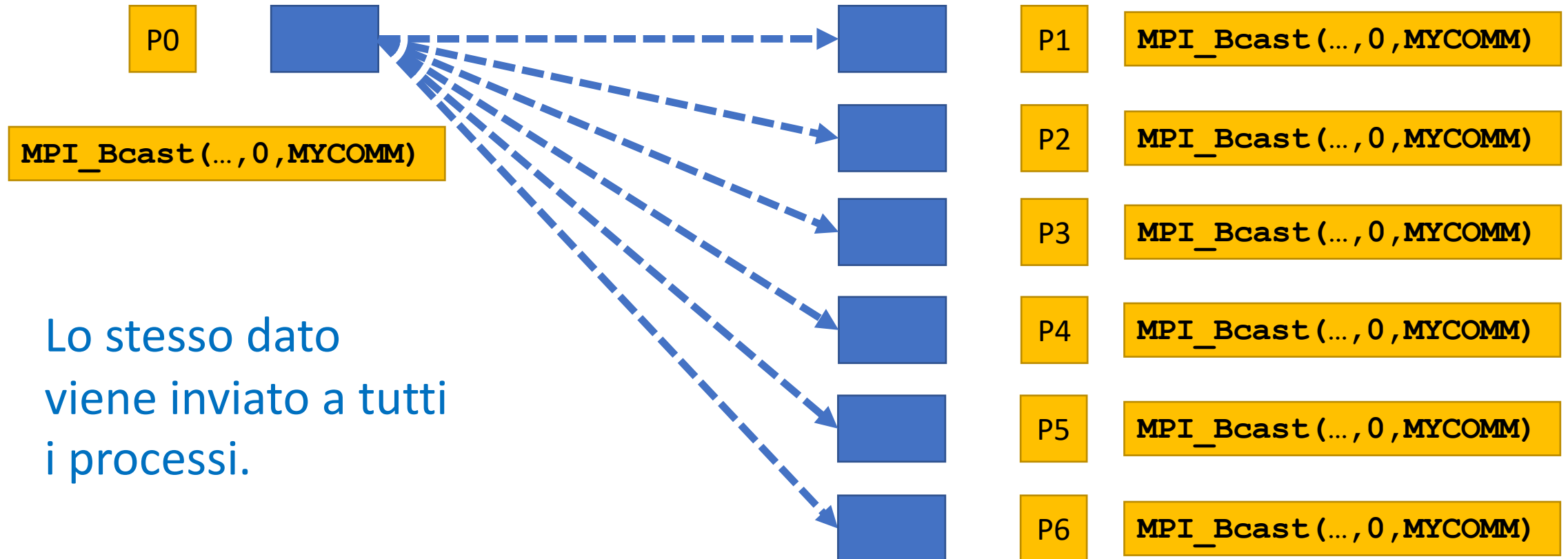
```
int MPI_Bcast(void* buffer, //buffer per il dato da inviare/ricevere
              int count,    //elementi da inviare/ricevere
              MPI_Datatype datatype, //tipo associato al dato
              int emitter_rank,    //rank del mittente (emitter)
              MPI_Comm communicator); // communicator
```

MPI\_Bcast viene eseguita da tutti i processi del gruppo:

- Il processo «**emitter**» invia il dato;
- tutti gli altri lo ricevono (recv bloccante).

Anche in questo caso l'implementazione è ottimizzata (tree structured communication)

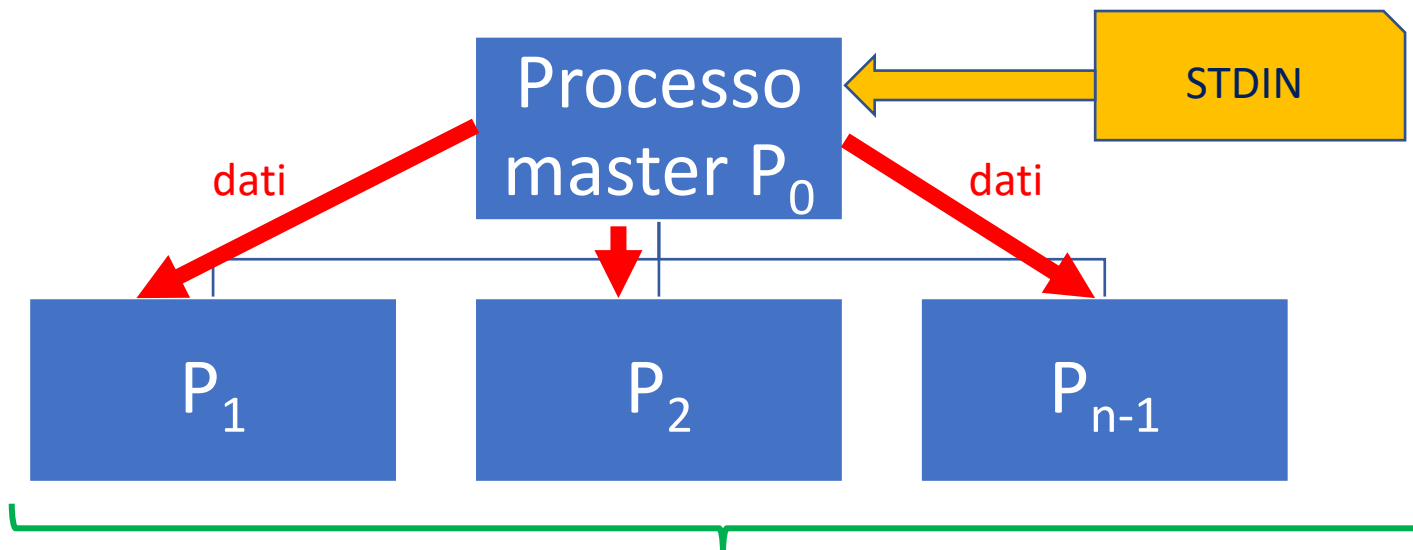
# MPI\_Bcast



# Esempio MPI\_Bcast: metodo dei trapezi

Consideriamo il caso in cui i **dati** del problema vengano **forniti da standard input**:

1. il processo di rank 0 acquisisce i dati: estremi dell'intervallo e numero di trapezi in cui scomporre l'integrale
2. il processo 0 invia i dati a tutti gli altri



## Processo Master:

1. legge i dati da stdin e li invia a tutti gli altri processi (Bcast)
2. esegue il calcolo per il suo sottointervallo
3. raccoglie i risultati ottenuti da tutti i processi, sommandoli tra loro.

**Processi Slave:** ognuno attende i dati dal master, calcola la somma delle aree dei trapezi contenuti in un sottointervallo dato; al termine ogni slave comunicherà al Master il risultato ottenuto.

# Esempio MPI\_Bcast: metodo dei trapezi

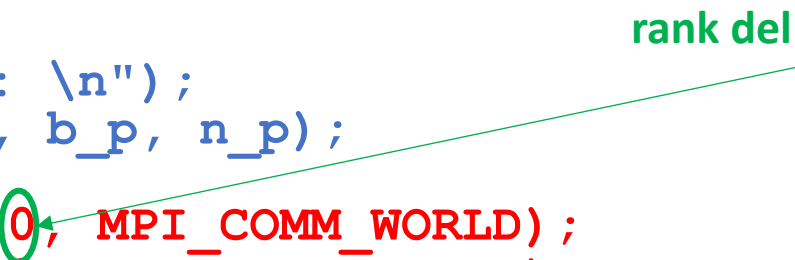
```
#include <stdio.h>
#include <mpi.h>

double f(double x) //funzione integranda
{
    return x*x;
}

double Trap(double left_endpt, double right_endpt, int trap_count, double
base_len){...}

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p)
{
    int dest;
    if(my_rank==0){
        printf("Immetti a, b, n: \n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

rank del mittente



```
int main(void) {
    int my_rank, comm_sz, n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;
    int source;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    Get_input(my_rank, comm_sz, &a, &b, &n);
    h = (b - a)/n;
    local_n = n/comm_sz;
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (my_rank == 0)
    {
        printf("Con n = %d trapezi, il risultato\n", n);
        printf("dell'integrale da %f a %f = %.15e\n", a, b, total_int);
    }
    MPI_Finalize();
    return 0;
}
```



# Scatter & Gather

L'esigenza di **distribuire dati** e **raccogliere risultati** verso/da processi paralleli è comune per moltissime applicazioni HPC.

## **Esempio:** somma di due vettori $C=A+B$

Dati due vettori A e B di uguale dimensione, calcolare il vettore somma C.

Partizioniamo ogni vettore in tanti sottovettori quanti sono i processori utilizzati; ad ogni processo  $P_i$  in esecuzione sul nodo  $N_i$  viene affidato il calcolo della somma di due sottovettori omologhi  $C_i=A_i+B_i$ .

Pertanto:

1. all'inizio dell'esecuzione è necessario **distribuire** a tutti i nodi  $N_i$  i sottovettori di A e di B
2. alla fine, occorre **raccogliere** in un unico vettore C i risultati  $C_i$  ottenuti dai singoli nodi.

A questo scopo, MPI definisce alcune primitive collettive che realizzano l'operazione 1 (**MPI\_Scatter**) e l'operazione 2 (**MPI\_Gather**)

# MPI\_Scatter

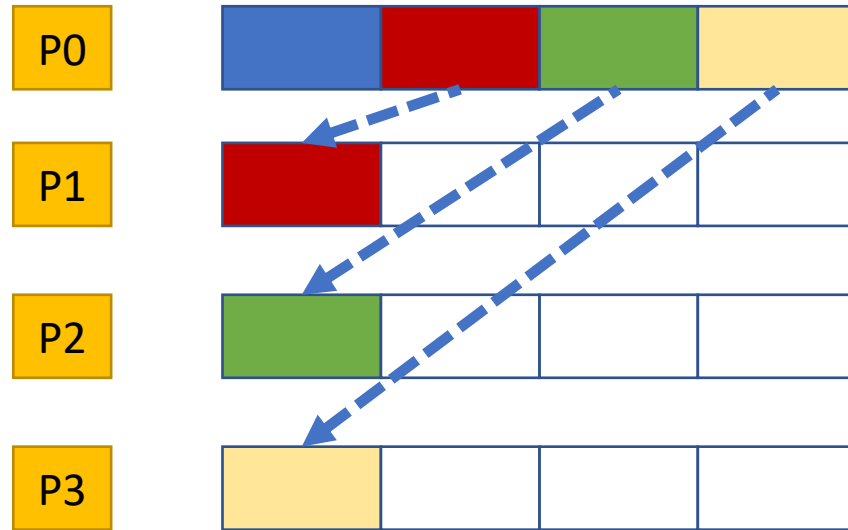
```
int MPI_Scatter(const void* sendbuf,           // buffer mittente
               int count_send,                // numero elementi/processo
               MPI_Datatype datatype_send,    // tipo del singolo elem.
               void* recvbuf,                 // buffer destinatario
               int count_recv,                // numero elementi ricevuti
               MPI_Datatype datatype_recv,    // tipo elem. ricevuto
               int root,                      // rank del sender
               MPI_Comm C) ;                 // communicator
```

Se **size** è la dimensione di C, il vettore **sendbuf** viene diviso in **size blocchi** di uguale dimensione:

- il primo blocco viene inviato al processo di rank 0
- il secondo blocco al processo di rank 1
- ...
- l'ultimo blocco al processo di rank (size-1)

Semantica ricezione bloccante.

# MPI\_Scatter



I blocchi vengono smistati ai processi seguendo l'ordine dei rank.

La semantica di ricezione è bloccante.

## Varianti MPI\_Scatter:

- MPI\_Scatterv: scatter con blocchi di dimensione variabile
- MPI\_Isscatter, MPI\_Isscatterv: ricezione non bloccante (verifica successiva con MPI\_Wait/MPI\_test)

# MPI\_Scatter: esempio

```
// scat.c
```

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

```
#include <stdlib.h>
```

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

```
#define DIM 20
```

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

```
{    int my_value, my_A[DIM];
```

```
    int size, my_rank;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    if(size > DIM)
```

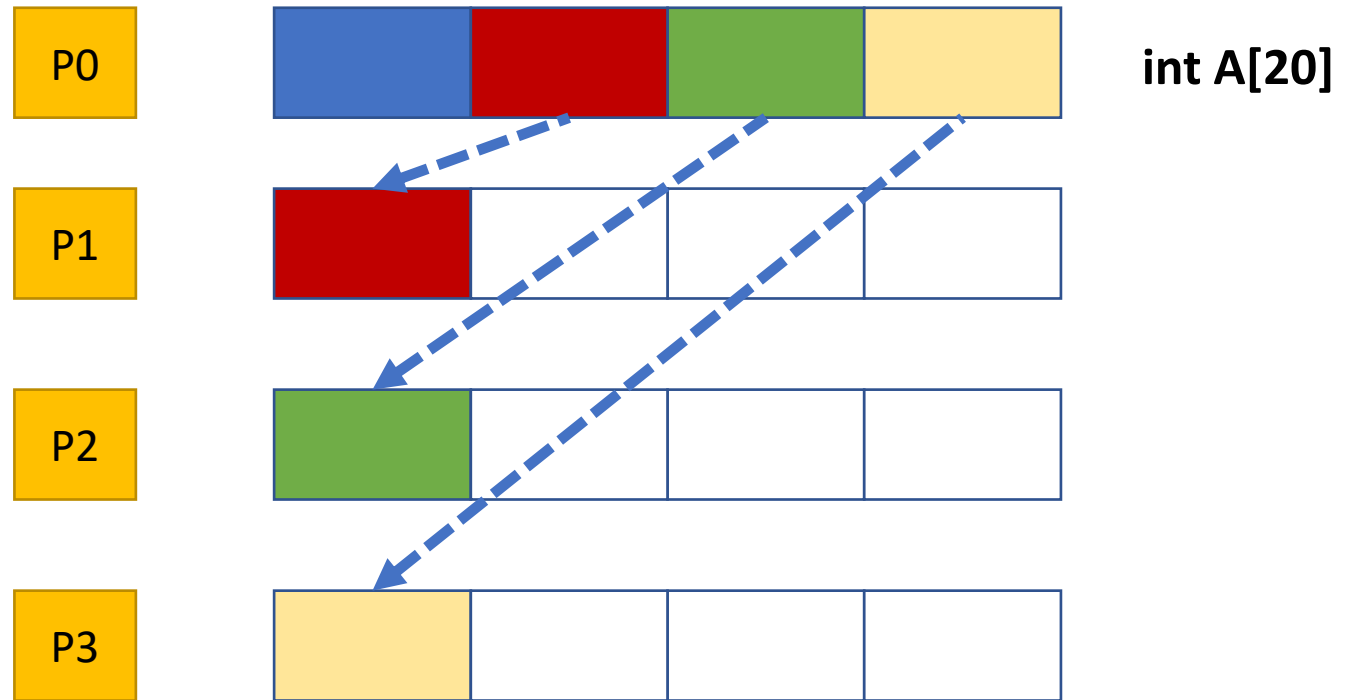
```
    {printf("il num. di processi %d è maggiore della dim. %d.\n",size, DIM);
```

```
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
```

```
    }
```

```
if (my_rank == 0)
{
    int A[DIM], i ;
    for(i=0; i<DIM; i++)
        A[i]=i*10; //inizializzazione A[]
    MPI_Scatter(A, DIM/size, MPI_INT, &my_A, DIM/size, MPI_INT, 0,
        MPI_COMM_WORLD);
}
else
    MPI_Scatter(NULL, DIM/size, MPI_INT, &my_A, DIM/size, MPI_INT, 0,
        MPI_COMM_WORLD);
printf("Processo %d ha ricevuto i seguenti valori :\n", my_rank);
for(int i=0; i<DIM/size; i++)
    printf("\t%d\n", my_A[i]);
MPI_Finalize();
return EXIT_SUCCESS;
} // fine main
```

# MPI\_Scatter



```
$mpicc -o scat scat.c
```

```
$ mpirun -n 4 scat
```

Il vettore A viene diviso in 4 **blocchi** di dimensione 5 ciascuno:

il primo blocco A[0,..4] viene inviato al processo P0

il secondo blocco A[5,..9] al processo P1

il terzo blocco A[10,..14] al processo P2

il quarto blocco A[15,..19] al processo P3

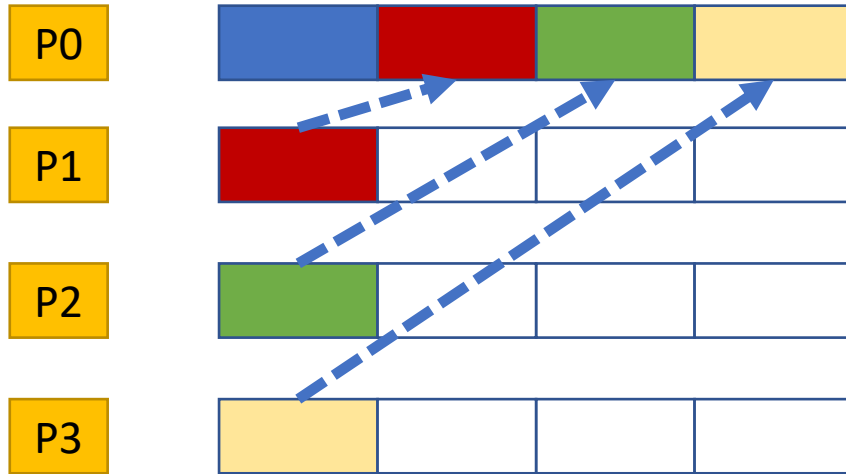
# MPI\_Gather

```
int MPI_Gather(void* sendbuf, // buffer di invio
              int count_send, // num. elementi da inviare
              MPI_Datatype datatype_send, //tipo elemento
              void* recvbuf, //buffer di ricezione (collettore)
              int count_recv, //num.elementi per messaggio ricevuto
              MPI_Datatype datatype_recv,
              int root, //rank collettore
              MPI_Comm communicator);
```

Se **size** è la dimensione di C, il vettore **recvbuf** viene riempito con size **blocchi** di uguale dimensione, ognuno ricevuto da un processo del communicator:

- il blocco ricevuto da rank 0 viene assegnato al primo blocco di recvbuf
- il blocco ricevuto da rank 1 viene assegnato al secondo blocco di recvbuf
- ...
- il blocco ricevuto da rank (size-1) viene assegnato all'ultimo blocco di recvbuf

# MPI\_Gather



- L'ordine di raccolta è quello dei rank.
- La ricezione è bloccante

## Varianti MPI\_Gather:

- MPI\_Gatherv: gather con blocchi di dimensione variabile
- MPI\_Allgather , MPI\_Allgatherv: ogni processo è collettore (oltre che sender)
- MPI\_Igather: ricezione non bloccante (verifica successiva con MPI\_Wait/MPI\_test)



# MPI\_Gather/Scatter: esempio

Si vuole calcolare la somma C di 2 vettori A e B:

```
int A[DIM], B[DIM], C[DIM];
```

Con size processi:

- il processo 0 calcola la somma dei primi 2 sottovettori di dimensione DIM/size
- il processo 1 calcola la somma dei secondi sottovettori di dimensione DIM/size
- ..
- il processo (size-1) calcola la somma degli ultimi sottovettori

Alla fine: il processo 0 raccoglie tutti i size blocchi risultato ricevuti da tutti i processi nel vettore risultato C.

**Distribuzione dei dati:** MPI\_Scatter

**Raccolta risultati:** MPI\_Gather

# Esempio: somma di vettori

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define DIM 12

int main(int argc, char* argv[])
{
    int  my_A[DIM], my_B[DIM], my_C[DIM];
    int size, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if(size > DIM)
    {
        printf("il numero di processi %d è maggiore della dimensione %d.\n",size, DIM);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
}
```

```
if (my_rank == 0) //master
{
    int A[DIM],B[DIM], i ;
    printf("inserire i %d elementi di A:\n", DIM);
    for(i=0; i<DIM; i++) //inizializzazione A
        scanf("%d", &A[i]);
    printf("inserire i %d elementi di B:\n", DIM);
    for(i=0; i<DIM; i++) //inizializzazione A
        scanf("%d", &B[i]);
    // verifica
    printf("[processo %d] vettore A:\n", my_rank);
    for(i=0;i<DIM;i++)
        printf("\t%d\n",A[i]);
    printf("[processo %d] vettore B:\n", my_rank);
    for(i=0;i<DIM;i++)
        printf("\t%d\n",B[i]);
    MPI_Scatter(A, DIM/size, MPI_INT, &my_A, DIM/size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(B, DIM/size, MPI_INT, &my_B, DIM/size, MPI_INT, 0, MPI_COMM_WORLD);
}
```

```
else // slave
{
    MPI_Scatter(NULL, DIM/size, MPI_INT, &my_A, DIM/size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(NULL, DIM/size, MPI_INT, &my_B, DIM/size, MPI_INT, 0, MPI_COMM_WORLD);
}

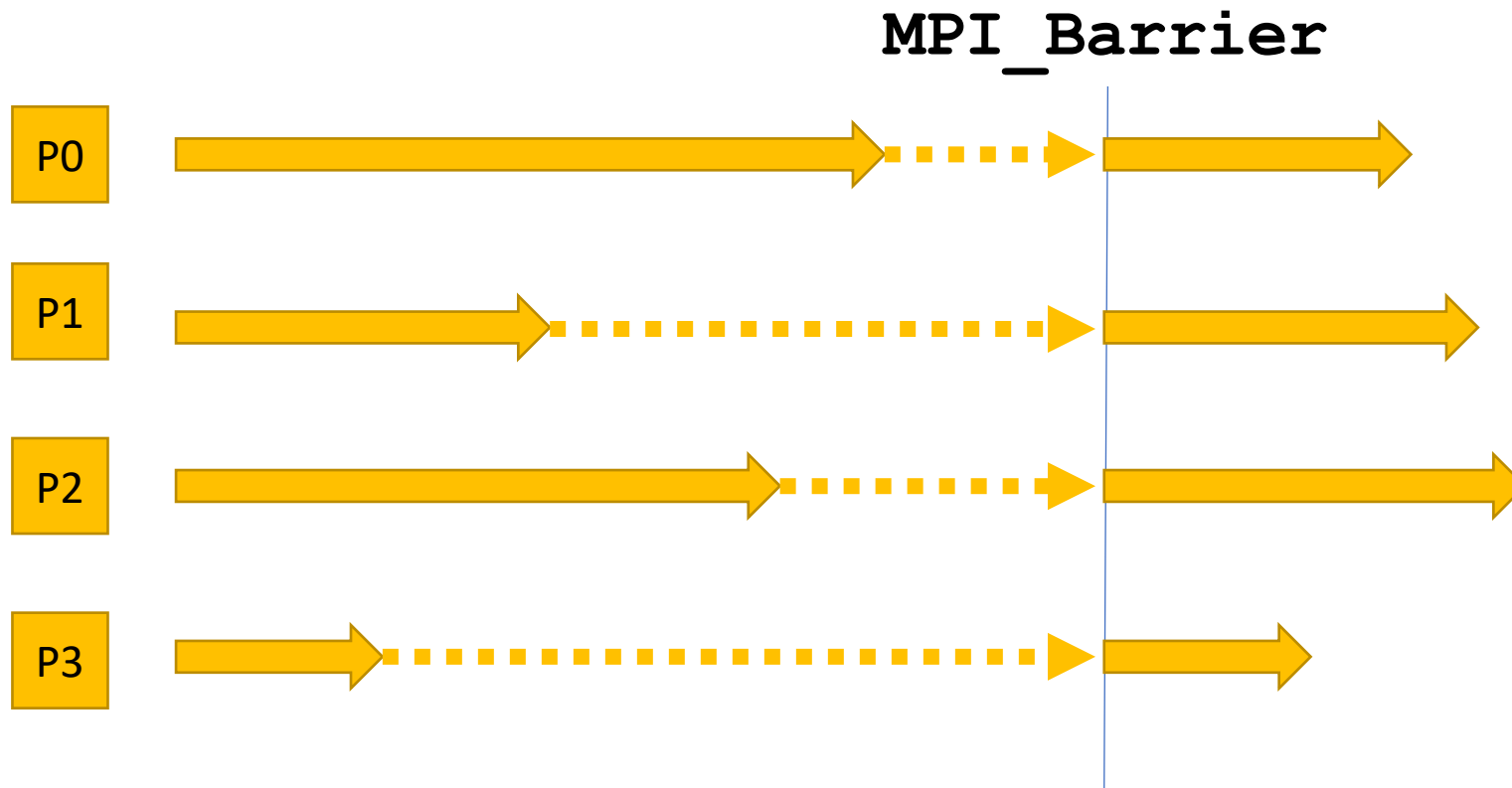
// calcolo somma sottovettori omologhi:
for(int i=0; i<DIM/size; i++)
    my_C[i]=my_A[i]+my_B[i];

if (my_rank==0) //collettore
{
    int C[DIM];
    MPI_Gather(&my_C, DIM/size, MPI_INT, C, DIM/size, MPI_INT, 0, MPI_COMM_WORLD); //raccolta
    printf("Risultato C=A+B:\n");
    for(int i=0; i<DIM; i++)
        printf("\t%d\n", C[i]);
}

else // sender
    MPI_Gather(&my_C, DIM/size, MPI_INT, NULL, 1, MPI_INT, 0, MPI_COMM_WORLD); //invio

MPI_Finalize();
return EXIT_SUCCESS;
}
```

# Barriera di sincronizzazione: MPI\_Barrier



# MPI\_Barrier

```
int MPI_Barrier(MPI_Comm comm) ;
```

Blocca ogni processo nel communicator comm fino a quando tutti non avranno chiamato la MPI\_Barrier.

# Timing

Il calcolo parallelo ha tra i suoi obiettivi primari il raggiungimento di elevate prestazioni. Per valutare le prestazioni effettive di un programma sono necessari strumenti per la misurazione del tempo:

```
double MPI_Wtime(void) ;
```

è una funzione che restituisce il valore corrente del tempo locale (Walltime).

Per la misurazione dei tempi di esecuzione di programmi, si calcola la differenza tra due valori prodotti da chiamate successive a MPI\_Wtime (es: una all'inizio dell'esecuzione e una alla fine): tale differenza esprime (in secondi) il tempo trascorso tra le 2 chiamate.

# Timing

Esempio di uso:

```
double start, end;
MPI_Init(&argc, &argv);
...
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

/* ... calcola ... */

MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
if (rank == 0)
    printf("Tempo di esecuzione = %f\n", end-start);

...
MPI_Finalize();
```



# Timing: esempio trapezi

```
#include <stdio.h>

#include <mpi.h>

double f(double x) {...} //funzione integranda

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p) {...}

double Trap(double left_endpt, double right_endpt, int trap_count, double base_len){...}

int main(void) {
    int my_rank, comm_sz, n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int, inizio, fine, local_elaps, global_elaps;
    int source;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    Get_input(my_rank, comm_sz, &a, &b, &n);
```

```

MPI_Barrier(MPI_COMM_WORLD);
inizio=MPI_Wtime();
h = (b - a)/n;
local_n = n/comm_sz;
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_int = Trap(local_a, local_b, local_n, h);
MPI_Reduce(&local_int, &total_int,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD); //sinc.
fine=MPI_Wtime();
local_elaps= fine-inizio;
printf("tempo impiegato da proc %d: %f secondi\n", my_rank, local_elaps);
MPI_Reduce(&local_elaps, &global_elaps,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
if (my_rank == 0)
{
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    printf("tempo impiegato: %f secondi\n", global_elaps);
}
MPI_Finalize();
return 0;
}

```