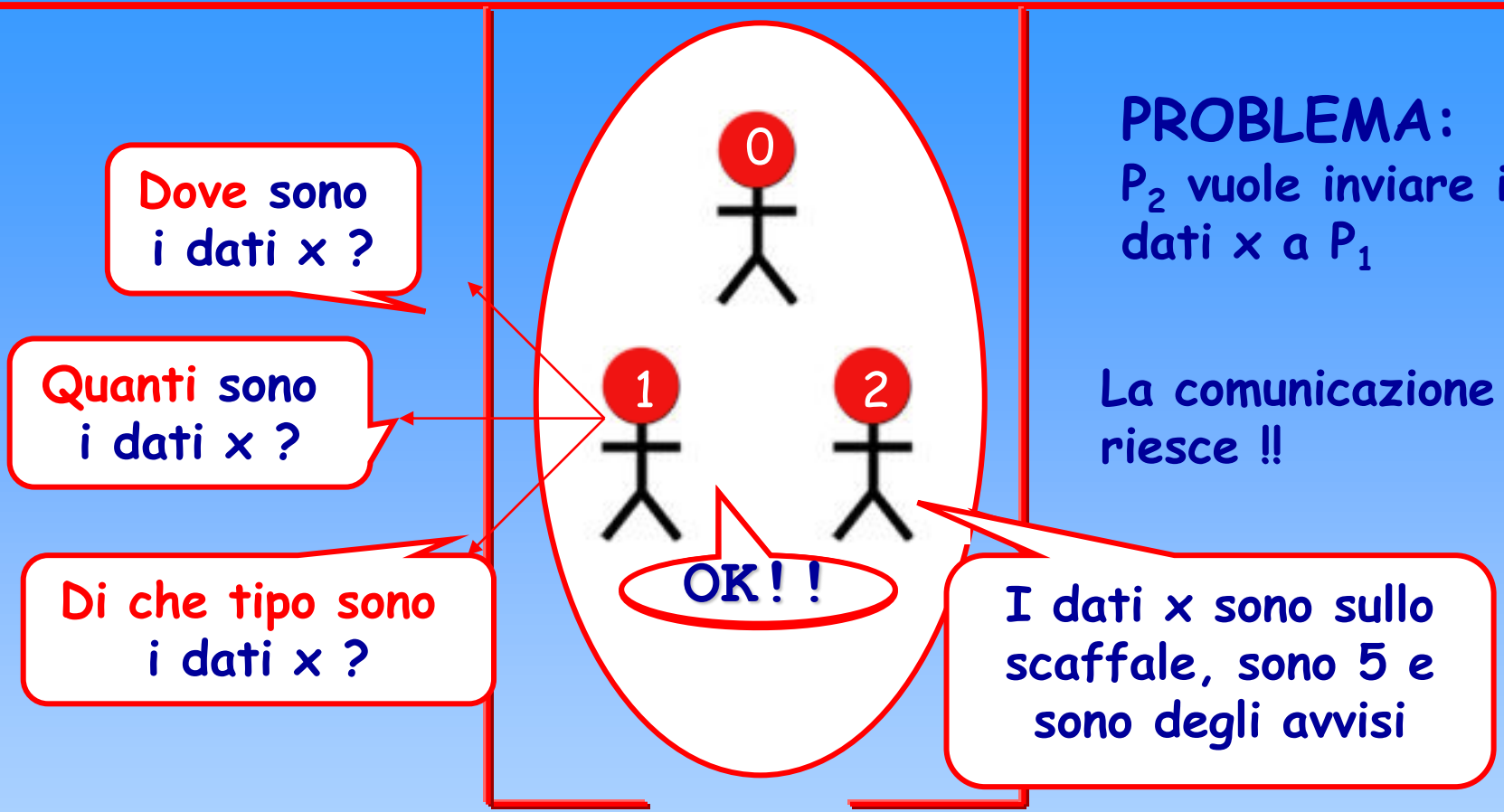
Message Passing Interface
MPI

Comunicazione di un messaggio.

Il Communicator MPI :



PROBLEMA:
P_2 vuole inviare i dati x a P_1

La comunicazione non riesce !!

COMMUNICATOR_A1

Comunicazione Riuscita !

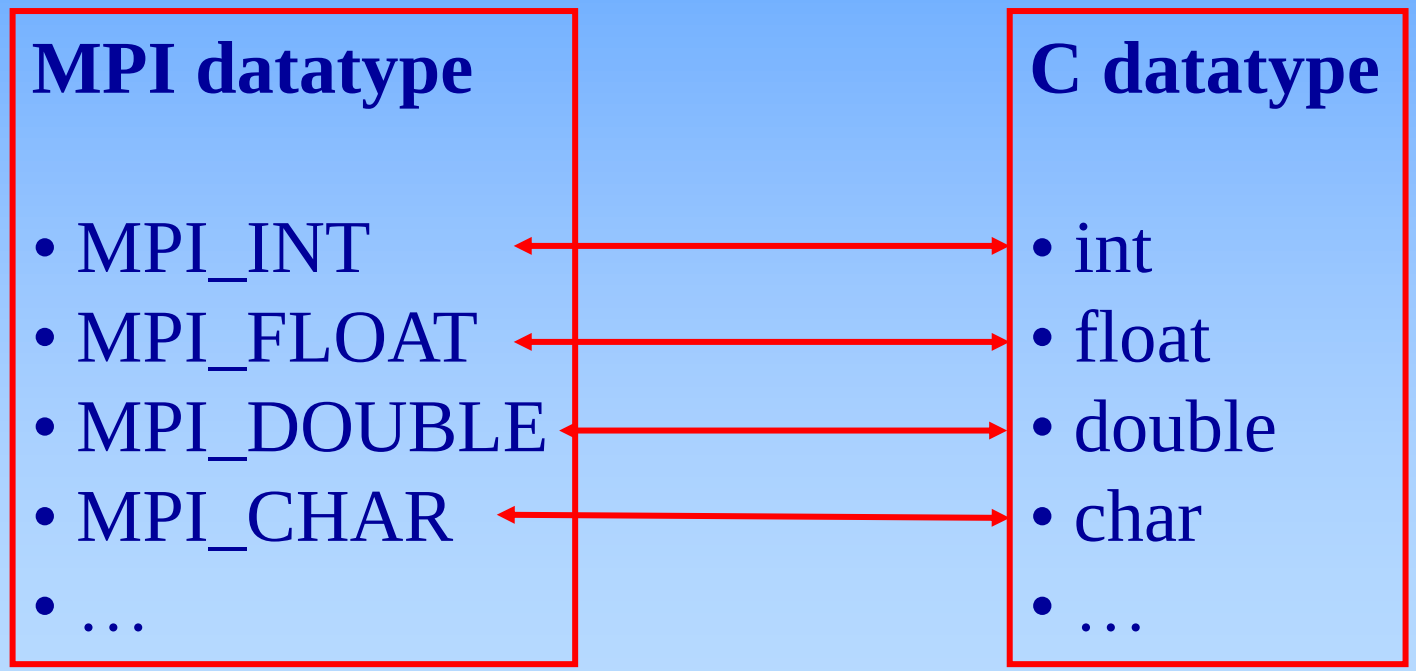
Caratteristiche di un messaggio

Un dato che deve essere spedito
o ricevuto attraverso un messaggio di MPI
è descritto dalla seguente tripla
(address, count, datatype)

Un datatype di MPI è ***predefinito*** e
corrisponde univocamente ad un tipo di dato
del linguaggio di programmazione utilizzato.

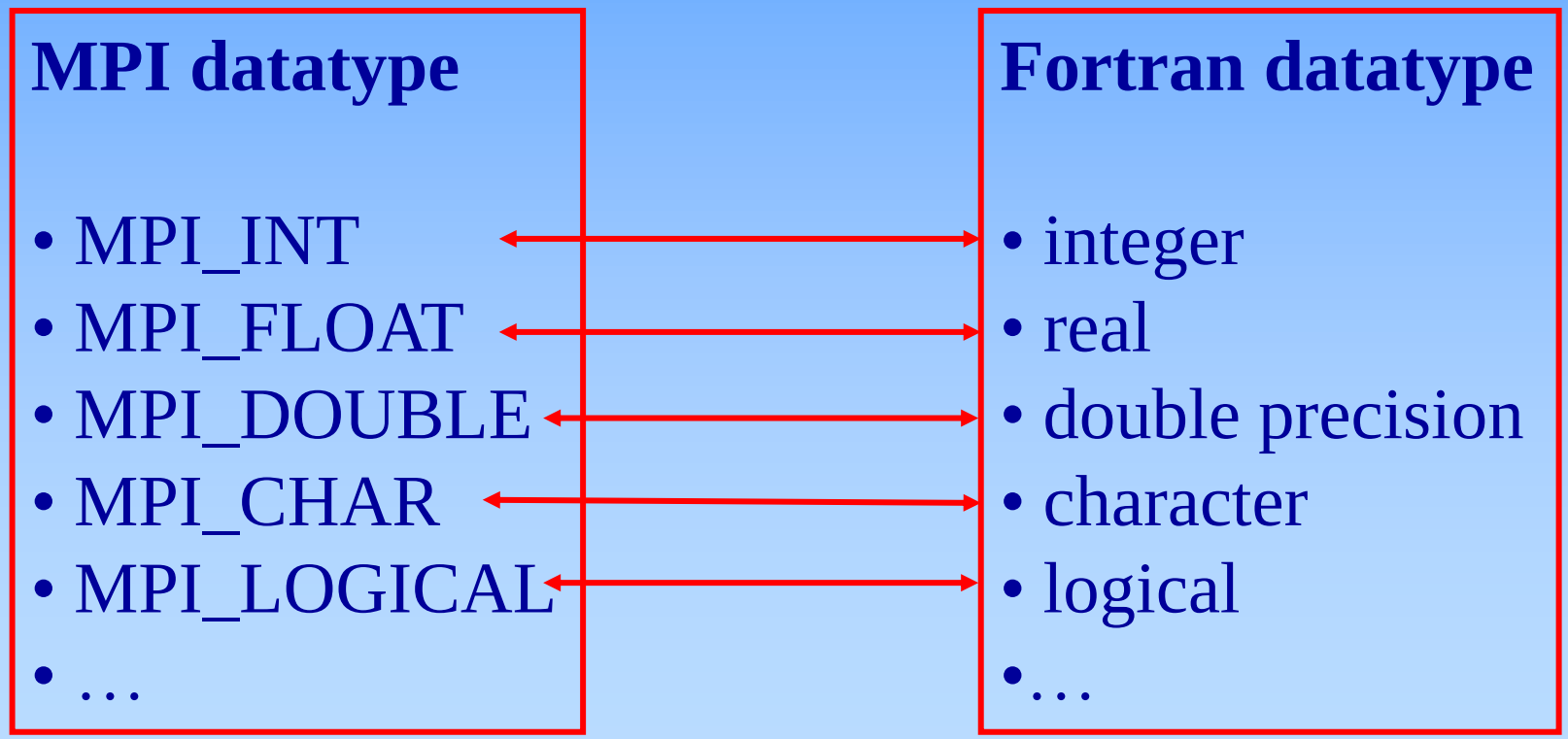
Esempio 1: linguaggio C

Ogni tipo di dato di MPI corrisponde univocamente ad un tipo di dato del linguaggio C.



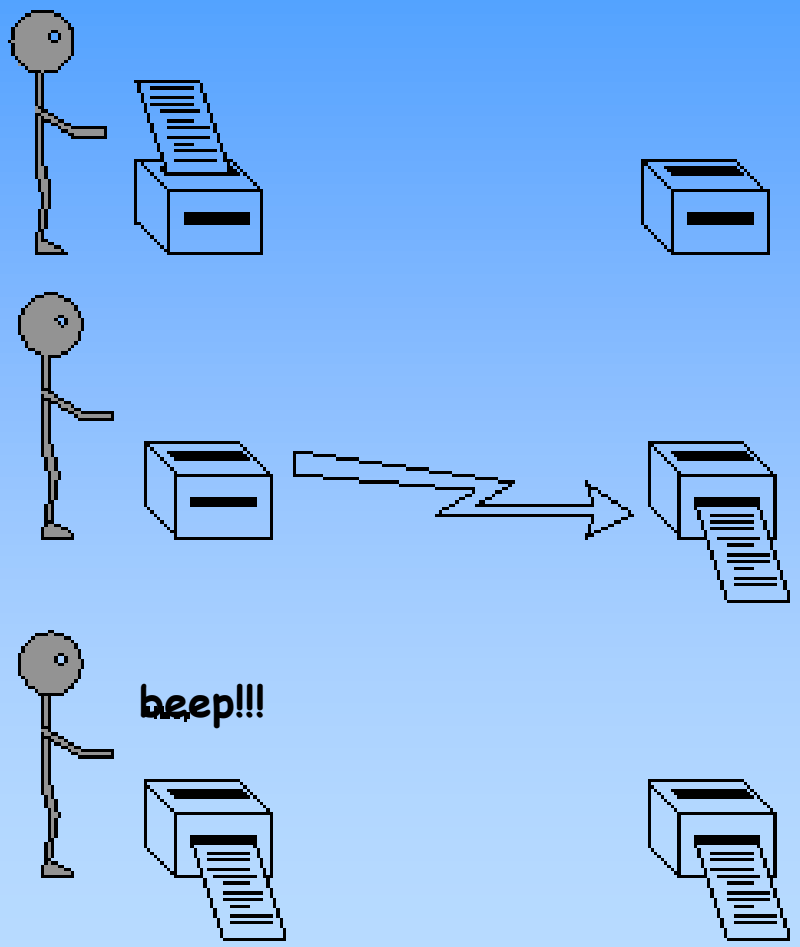
Esempio 2: linguaggio Fortran

Ogni tipo di dato di MPI corrisponde
univocamente
ad un tipo di dato del linguaggio Fortran.



Tipi di comunicazioni (Esempio 1) :

Trasmissione di un fax

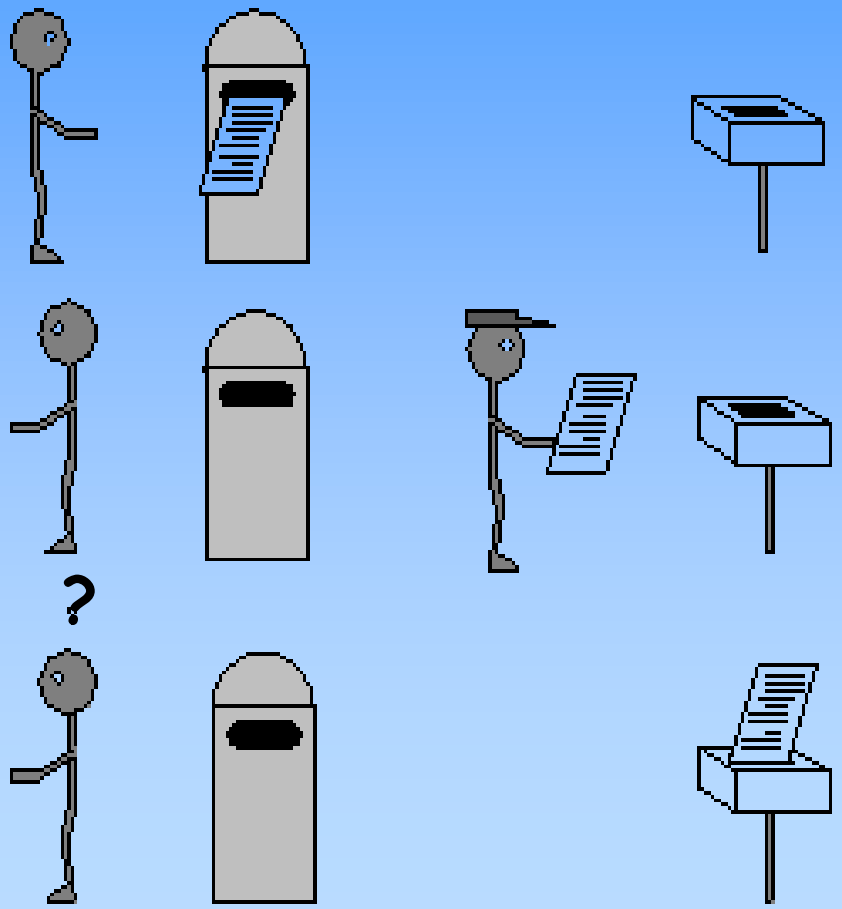


Il fax trasmittente **termina l'operazione** quando il fax ricevente ha ricevuto completamente il messaggio.

L'operazione di ricezione del messaggio **è stata completata** .

Tipi di comunicazioni (Esempio 2) :

Spedizione di una lettera tramite servizio postale

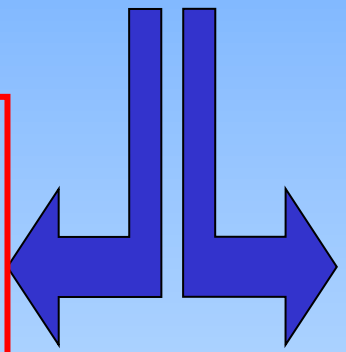


Il mittente spedisce la lettera, ma non può sapere se è stata ricevuta .

Il mittente **non sa** se l'operazione di ricezione del messaggio è stata completata.

Tipi di comunicazioni:

La spedizione o la ricezione
di un messaggio
da parte di un processo può essere
bloccante o **non bloccante**



Se un processo
esegue una
comunicazione
bloccante
si arresta fino a
conclusione
dell'operazione.

Se un processo
esegue una
comunicazione
non bloccante
prosegue senza
preoccuparsi della
conclusione
dell'operazione.

Comunicazioni bloccanti in MPI

Funzione ***bloccante*** per la ***spedizione*** di un messaggio:

MPI_Send

Funzione ***bloccante*** per la ***ricezione*** di un messaggio:

MPI_Recv

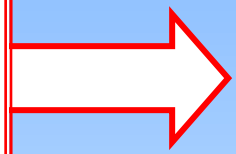
Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=10;
       MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     }else { tag=10;
       MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
      }
     MPI_Get_count(&info,MPI_INT,&num);
     MPI_Finalize();
     return 0;
}
```

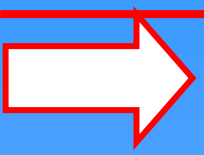
FUNZIONA SOLO SE LANCIATO CON 2 PROCESSI!!

Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=10;
       MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     }else { tag=10;
       MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
      }
     MPI_Get_count(&info,MPI_INT,&num);
     MPI_Finalize();
     return 0;
}
```



Nel programma … :



MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD)

- Con questa routine il processo chiamante P_0 spedisce il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, al processo P_1

- i due processi appartengono entrambi al communicator **MPI_COMM_WORLD**.

- Il parametro **tag** individua univocamente tale spedizione.

In generale (comunicazione uno ad uno bloccante) :

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

- Il processo che esegue questa routine spedisce i primi **count** elementi di **buffer**, di tipo **datatype**, al processo con identificativo **dest**.

- L'identificativo **tag** individua univocamente il messaggio nel contesto **comm**.

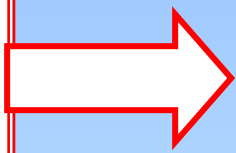
In dettaglio…

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

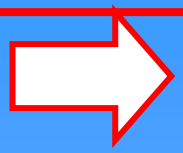
`*buffer`	indirizzo del dato da spedire
`count`	numero dei dati da spedire
`datatype`	tipo dei dati da spedire
`dest`	identificativo del processo destinatario
`comm`	identificativo del communicator
`tag`	identificativo del messaggio

Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=10;
       MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     }else { tag=10;
       MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
       }
     MPI_Get_count(&info,MPI_INT,&num);
     MPI_Finalize();
     return 0;
}
```



Nel programma … :



MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info)

- Con questa routine il processo chiamante P_1 riceve il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, dal processo P_0 ; i due processi appartengono entrambi al communicator **MPI_COMM_WORLD**.

- Il parametro **tag** idividua univocamente tale spedizione.

- Il parametro **info**, di tipo **MPI_Status**, contiene informazioni sulla ricezione del messaggio.

In generale (comunicazione uno ad uno bloccante) :

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

- Il processo che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processo con identificativo **source**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- **status** è un tipo predefinito di MPI che racchiude informazioni sulla ricezione del messaggio.

In dettaglio…

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

***buffer** indirizzo del dato in cui ricevere

count numero dei dati da ricevere

datatype tipo dei dati da ricevere

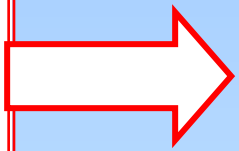
source identificativo del processo da cui ricevere

comm identificativo del communicator

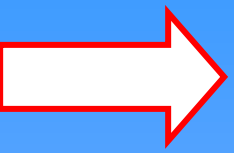
tag identificativo del messaggio

Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=10;
       MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     }else { tag=10;
       MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
       MPI_Get_count(&info,MPI_INT,&num); }
     MPI_Finalize();
     return 0;
}
```



Nel programma … :



```
MPI_Get_count(&info,MPI_INT,&num);
```

- num : numero di elementi ricevuti

- Questa routine permette al processo chiamante di conoscere il numero num di elementi ricevuti, di tipo MPI_INT, nella spedizione individuata da info.

In Generale… :

```
MPI_GET_COUNT(MPI_Status *status
              MPI_Datatype datatype, int *count);
```

`MPI_Status` in C è un tipo di dato strutturato, composto da tre campi:

- identificativo del processo da cui ricevere
- identificativo del messaggio
- indicatore di errore

• Il processo che esegue questa routine, memorizza nella variabile count il numero di elementi, di tipo datatype, che riceve dal messaggio e dal processo indicati nella variabile status.

Comunicazioni non bloccanti in MPI: modalità *Immediate*

Funzione ***non bloccante*** per la ***spedizione*** di un messaggio:

`MPI_Isend`

Funzione ***non bloccante*** per la ***ricezione*** di un messaggio:

`MPI_Irecv`

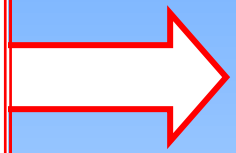
Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc, n, tag;
     MPI_Request rqst;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=20;
       MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
     }else {
       tag=20;
       MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
      }
     MPI_Finalize();
     return 0;
}
```

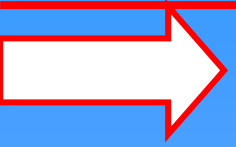
FUNZIONA SOLO SE LANCIATO CON 2 PROCESSI!!

Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc, n, tag;
     MPI_Request rqst;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=20;
       MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
     }else {
       tag=20;
       MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
      }
     MPI_Finalize();
     return 0;
}
```



Nel programma … :



MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst)

- Il processo chiamante P_0 spedisce il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, al processo P_1 ; i due processi appartengono entrambi al communicator **MPI_COMM_WORLD**.

- Il parametro **tag** idividua univocamente tale spedizione.

- Il parametro **rqst** contiene le informazioni dell'intera spedizione.
- Il processo P_0, appena inviato il parametro n, è **libero** di procedere nelle successive istruzioni.

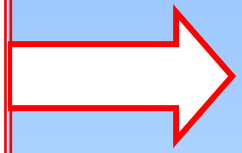
In generale (comunicazione uno ad uno non bloccante) :

```
MPI_Isend(void *buffer, int count,
          MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm,
          MPI_Request *request);
```

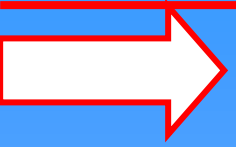
- Il processo che esegue questa routine spedisce i primi **count** elementi di **buffer**, del tipo **datatype**, al processo con identificativo **dest**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- L'oggetto **request** crea un nesso tra la trasmissione e la ricezione del messaggio

Un semplice programma con 2 processi:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{   int menum, nproc, n, tag;
    MPI_Request rqst;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    if(menum==0)
    { scanf("%d",&n);
      tag=20;
      MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
    }else {
      tag=20;
      MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
     }
    MPI_Finalize();
    return 0;
}
```



Nel programma … :



MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst)

- Il processo chiamante P_1 riceve il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, dal processo P_0 ; i due processi appartengono entrambi al communicator **MPI_COMM_WORLD**.

- Il parametro **tag** individua univocamente tale ricezione.

- Il parametro **rqst** contiene le informazioni dell'intera spedizione.

- Il processo P_1, appena ricevuto il parametro n, è **libero** di procedere nelle successive istruzioni.

Ricezione di un messaggio (comunicazione uno ad uno)

```
MPI_Irecv(void *buffer, int count,
          MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm,
          MPI_Request *request);
```

- Il processo che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processo con identificativo **source**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- L'oggetto **request** crea un nesso tra la trasmissione e la ricezione del messaggio.

In particolare: request

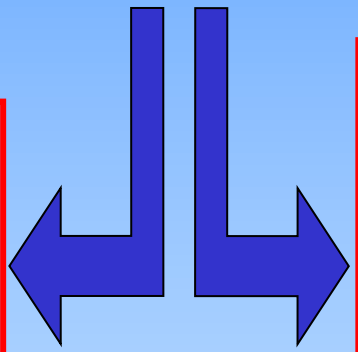
Le operazioni non bloccanti utilizzano
l'oggetto **request**
di un tipo predefinito di MPI: **MPI_Request.**

Tale oggetto **collega**
l'operazione che *inizia* la comunicazione in esame
con l'operazione che la *termina*.

Osservazione

L'oggetto **request** ha nelle comunicazioni, un ruolo simile a quello di **status**.

status
contiene informazioni sulla ***ricezione*** del messaggio.

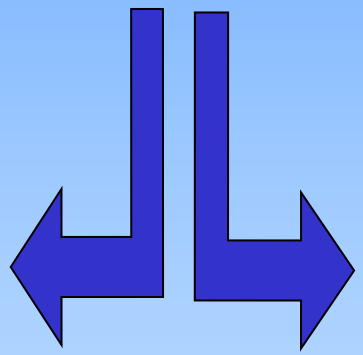


request
contiene informazioni su ***tutta la fase di trasmissione*** o di ***ricezione*** del messaggio.

Osservazione: termine di un'operazione non bloccante

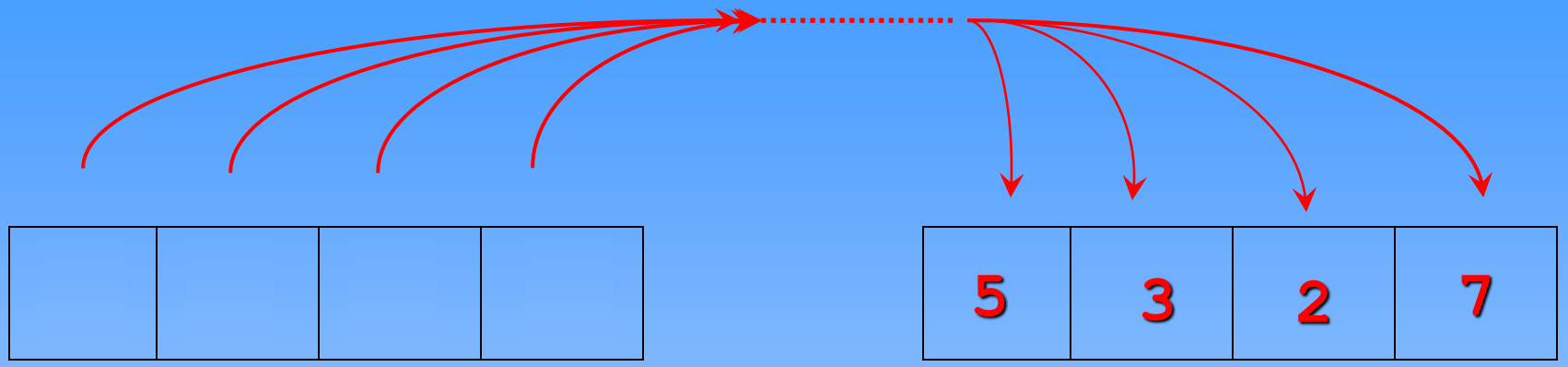
Un'operazione non bloccante
è terminata
quando:

Nel caso della spedizione, quando il **buffer** è nuovamente riutilizzabile dal programma.



Nel caso della ricezione quando il messaggio è stato interamente ricevuto.

Osservazione: termine di un'operazione non bloccante



Vettore x da spedire

Vettore x da ricevere

Operazione di
spedizione
non bloccante
terminata

Operazione di
ricezione
non bloccante
terminata

Domanda

Utilizzando una comunicazione non bloccante

come si fa a sapere

se l'operazione di spedizione o di ricezione

è stata terminata

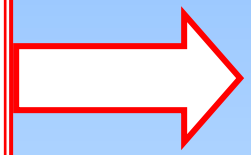
?

Risposta

MPI mette a disposizione
delle funzioni per
controllare tutta la fase
di trasmissione di un messaggio
mediante comunicazione
non bloccante.

Controllo sullo stato di un'operazione non bloccante…

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int flag,nloc;
     …
     nloc=1;
     if(menum==0)
     { scanf("%d",&n);
       tag=20;
       MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
     }
     if(menum!=0)
     {
        tag=20;
        MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
        MPI_Test(&rqst,&flag,&info);
        if(flag==1){
           nloc+=n;}else{
        MPI_Wait(&rqst,&info);
        nloc+=n;}
        printf("nloc=%d \n",nloc);
     }
     …
}
```



Nel programma … :

```
MPI_Test(&rqst,&flag,&info);
```

- Il processo chiamante P_1 verifica se la ricezione di **n**, individuata da **rqst**, è stata completata; in questo caso **flag**=1 e procede all'incremento di nloc.
- Altrimenti flag=0.

Controllo sullo stato di un'operazione non bloccante…

```
MPI_Test(MPI_Request *request,
         int *flag, MPI_Status *status);
```

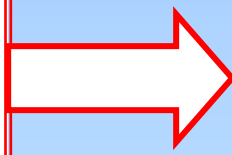
- Il processo che esegue questa routine testa lo stato della comunicazione non bloccante, identificata da **request**.
- La funzione **MPI_Test** ritorna l'intero **flag**:

flag = 1, l'operazione identificata da **request** è terminata;

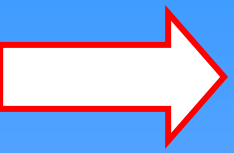
flag = 0, l'operazione identificata da **request** NON è terminata;

Controllo sullo stato di un'operazione non bloccante…

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int flag,nloc;
     …
     nloc=1;
     if(menum==0)
     { scanf("%d",&n);
       tag=20;
       MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
     }
     if(menum!=0)
     {
        tag=20;
        MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
        MPI_Test(&rqst,&flag,&info);
        if(flag==1){
           nloc+=n;}else{
        MPI_Wait(&rqst,&info);
        nloc+=n;}
        printf("nloc=%d \n",nloc);
     }
     …
}
```



Nel programma … :



MPI_Wait(&rqst,&info);

- Il processo chiamante P_1 procede nelle istruzioni solo quando la ricezione di n è stata completata.

In generale… :

```
MPI_Wait(MPI_Request *request,
         MPI_Status *status);
```

- Il processo che esegue questa routine controlla lo stato della comunicazione non bloccante, identificata da **request**, e si arresta solo quando l'operazione in esame si è conclusa.

- In status si hanno informazioni sul completamento dell'operazione di Wait.

Vantaggi delle operazioni non bloccanti

Le comunicazioni di tipo non bloccante hanno DUE vantaggi:

1) Il processo non è obbligato ad ***aspettare*** in stato di attesa.

```
…
if(menum==0)
 { scanf("%d",&n);
   tag=20;
   MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
 }
 /* P0 può procedere nelle operazioni senza dover
    attendere il risultato della spedizione di n
    al processo P1 */
 if(menum!=0)
 {
  …
 }
 …
```

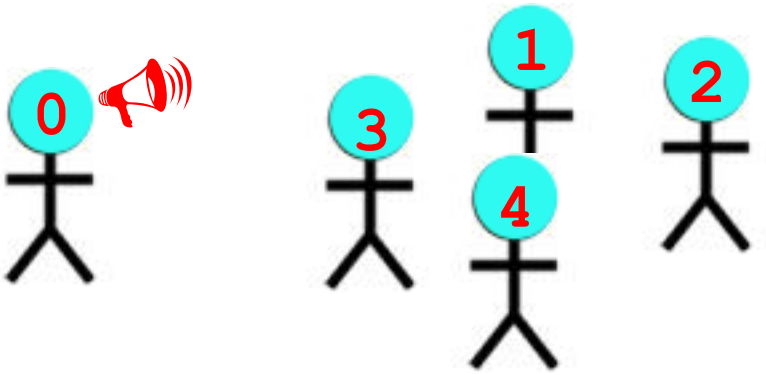
Vantaggi delle operazioni non bloccanti

2) Più comunicazioni possono avere luogo ***contemporaneamente*** sovrapponendosi almeno in parte tra loro.

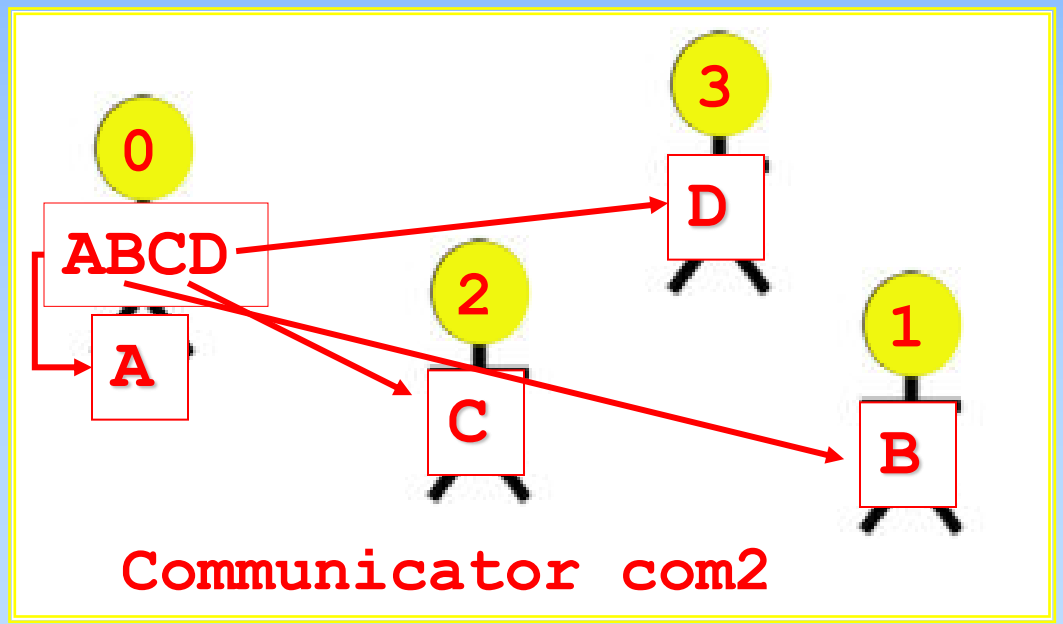
```
…
if(menum==0)
{ …
 /* P0 spedisce due elementi a P1 */
  MPI_Isend(&a,1,MPI_INT,1,0,MPI_COMM_WPRLD,&rqst1);
  MPI_Isend(&b,1,MPI_INT,1,0,MPI_COMM_WORLD,&rqst2);
} elseif(menum==1) {
/* P1 riceve due elementi da P0 secondo l'ordine di spedizione*/
  MPI_Irecv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&rqst1);
  MPI_Irecv(&b,1,MPI_INT,0,0,MPI_COMM_WORLD,&rqst2);
}
…
```

Le operazioni collettive.

Esempi di operazioni collettive :



Nel connumicator com1 P_0 comunica con tutti gli altri processi



P_0 distribuisce a tutti i processi di com2 un elemento del proprio vettore

Caratteristiche delle operazioni collettive

Le operazioni collettive sono eseguite da tutti i processi appartenenti ad un communicator.

Inoltre…

- Tutti i processi che eseguono l'operazione collettiva eseguono almeno una comunicazione.
- L'operazione collettiva può richiedere una sincronizzazione.
- Tutte le operazioni collettive sono bloccanti.

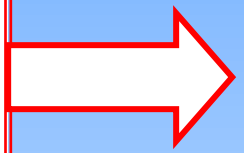
Scopo delle operazioni collettive

Le operazioni collettive permettono:

- La Sincronizzazione dei processi.
- L'esecuzione di operazioni globali (es. ricerca del massimo in un vettore distribuito fra i processi).
- Gestione ottimizzata degli input/output seguendo uno schema ad albero.

Sincronizzazione dei processi … :

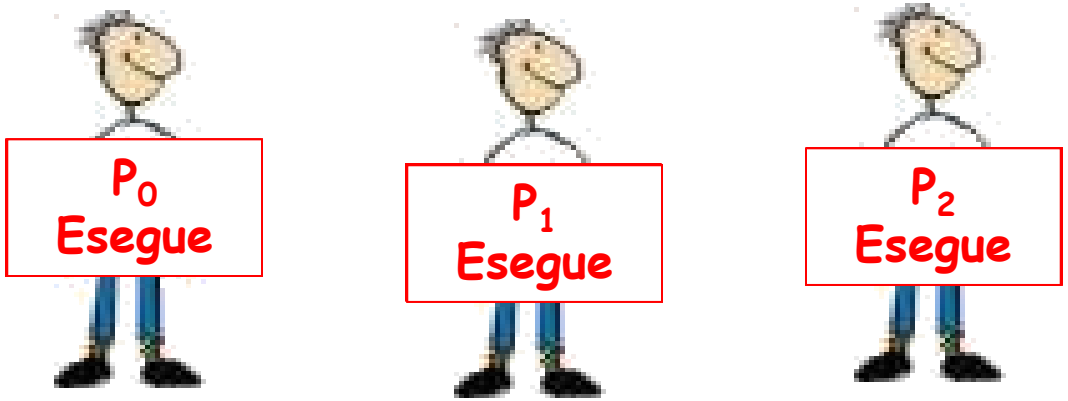
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    …
     MPI_Init(&argc,&argv);
     …
     MPI_Barrier(MPI_COMM_WORLD);
     …
   MPI_Finalize();
}
```



Nel programma … :

MPI_Barrier(MPI_COMM_WORLD);

- Ogni processo dell'ambiente MPI_COMM_WORLD può procedere solo quando tutti gli altri avranno richiamato questa routine.



In generale … :

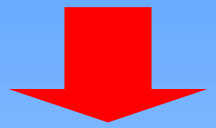
```
MPI_Barrier(MPI_Comm comm);
```

- Questa routine fornisce un meccanismo sincronizzante per tutti i processi del communicator **comm**.

- Ogni processo si *ferma* fin quando tutti i processi di **comm** non eseguono **MPI_Barrier**.

La comunicazione collettiva di un messaggio

La comunicazione
di un messaggio può coinvolgere
due o più processi.

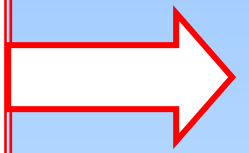


Per comunicazioni che coinvolgono solo due processi → Si considerano funzioni MPI per **comunicazioni uno a uno**

Per comunicazioni che coinvolgono più processi → Si considerano funzioni MPI per **comunicazioni collettive**

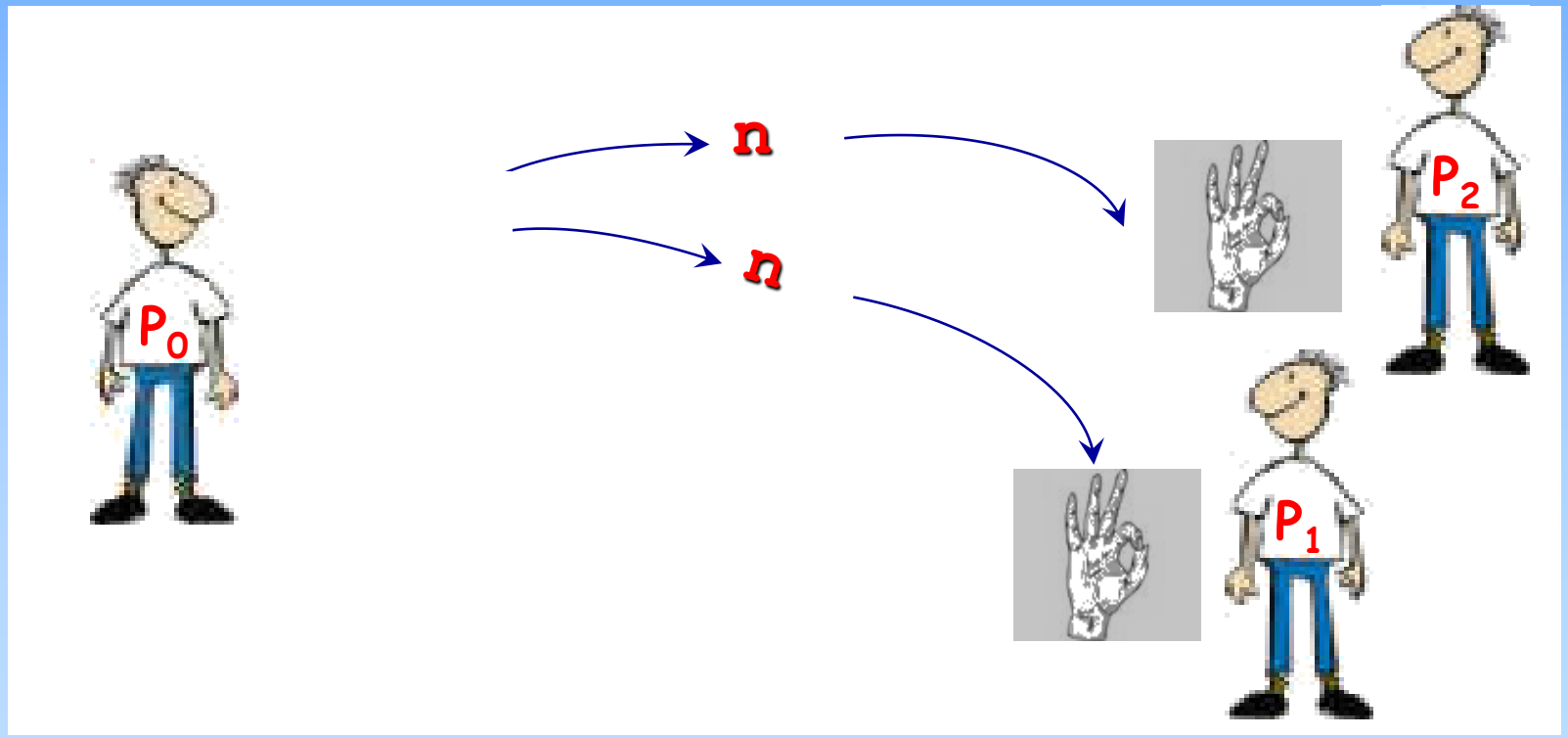
Spedizione collettiva, uno a molti: 1/4

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     {
       scanf("%d",&n);
     }
     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Finalize();
     return 0;
}
```



```
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

- Il processo P_0 spedisce **n**, di tipo **MPI_INT** e di dimensione **1**, a tutti i processi dell'ambiente **MPI_COMM_WORLD**.



In generale … :

```
MPI_Bcast(void *buffer, int count,
          MPI_Datatype datatype,
          int root, MPI_Comm comm);
```

- Il processo con identificativo `root` spedisce a tutti i processi del comunicator `comm` lo stesso dato memorizzato in `*buffer`.

- `Count, datatype, comm` devono essere uguali per ogni processo di `comm`.

In dettaglio…:


```
MPI_Bcast(void *buffer, int count,
          MPI_Datatype datatype,
          int root, MPI_Comm comm);
```

***buffer** indirizzo del dato da spedire

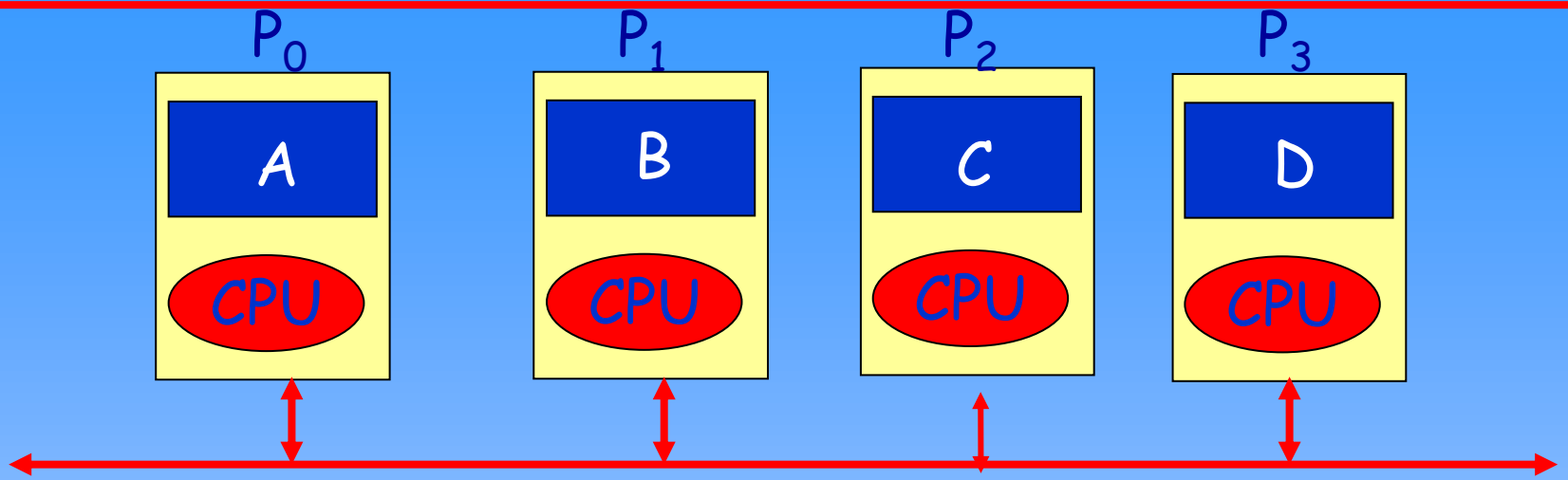
count numero dei dati da spedire

datatype tipo dei dati da spedire

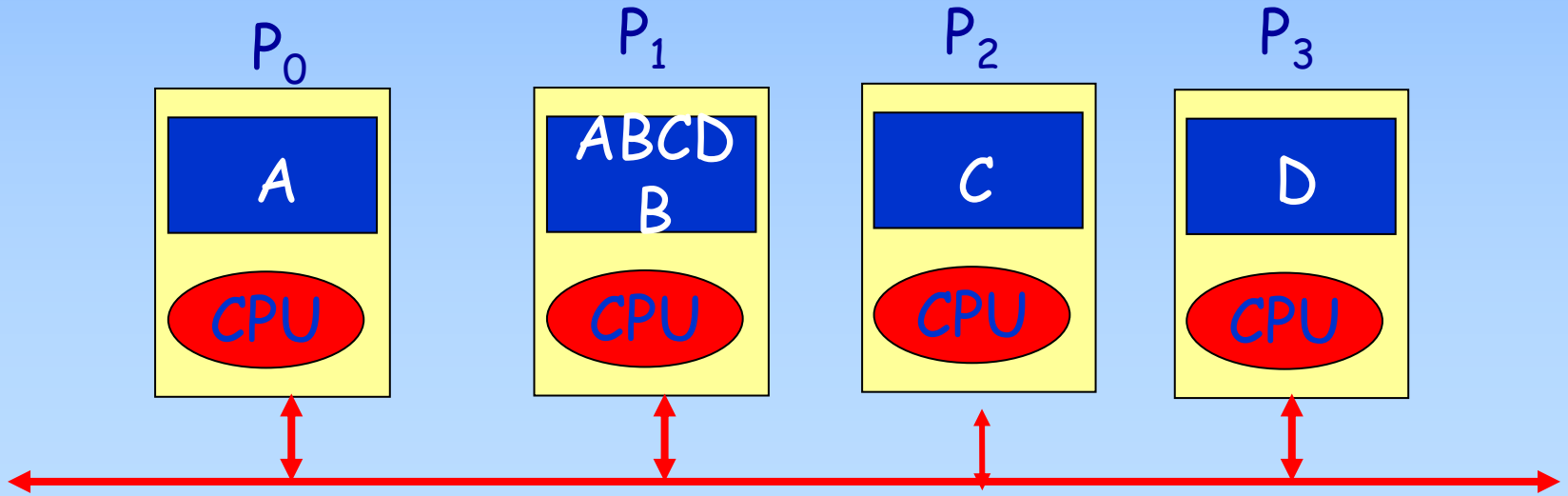
root identificativo del processo che spedisce a tutti

comm identificativo del communicator

Operazione collettiva di tipo: *data gather*



Tutti i processi spediscono i propri dati ad un processo assegnato (es al processo P_1)



Il gather in MPI


```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype datatype,
           void *recv_buff,int recv_count,
           MPI_Datatype recv_type,
           int root, MPI_Comm comm);
```

- Ogni processo di **comm** spedisce il contenuto di ***send_buff** al processo con identificativo **root**
- Il processo con identificativo **root** *concatena* i dati ricevuti in **recv_buff**, memorizzando prima i dati ricevuti dal processo 0, poi i dati ricevuti dal processo 1, poi quelli ricevuti dal processo 2, etc…

```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype datatype,
           void *recv_buff,int recv_count,
           MPI_Datatype recv_type,
           int root, MPI_Comm comm);
```

- Gli argomenti `recv_` sono significativi solo per il processo `root`
- L'argomento `recv_count` è il numero di dati da ricevere da ogni processo, **non è** il numero totale dei dati da ricevere.

Il gather in MPI: in dettaglio… 3/3

```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype sendtype,
           void *recv_buff,int recv_count,
           MPI_Datatype recv_type,
           int root, MPI_Comm comm);
```

***send_buff** indirizzo del dato da spedire

send_count numero dei dati da spedire

sendtype tipo dei dati da spedire

***recv_buff** indirizzo del dato in cui **root** riceve

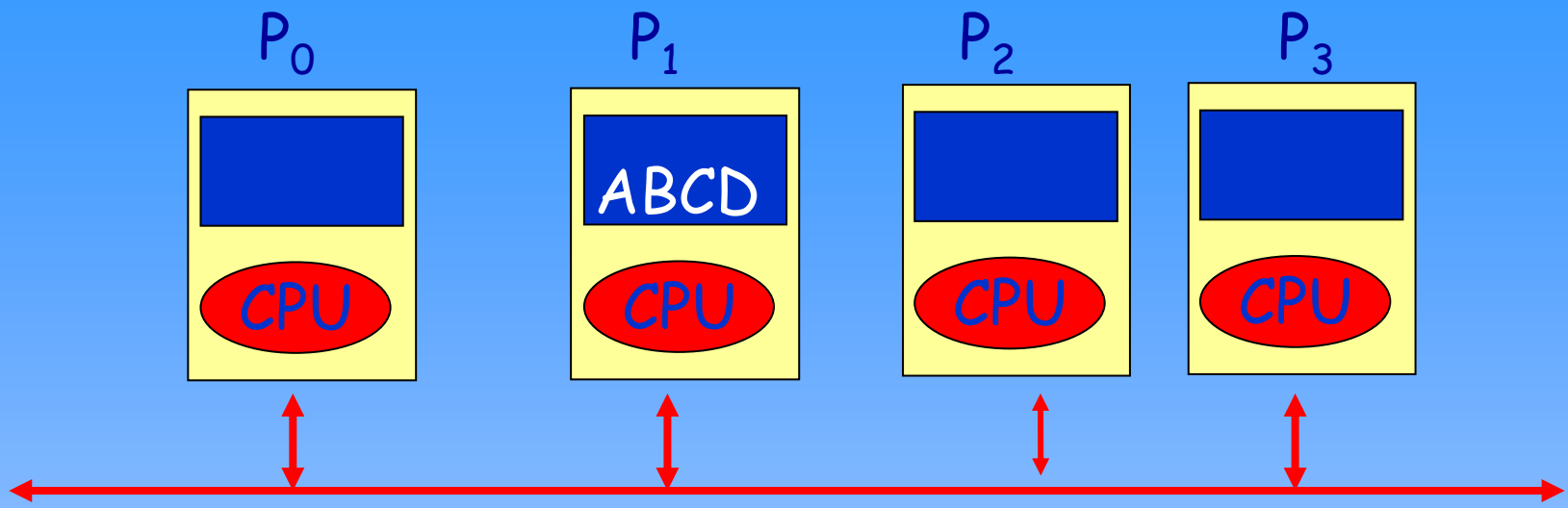
recv_count numero dei dati che root riceve

recv_type tipo dei dati che root riceve

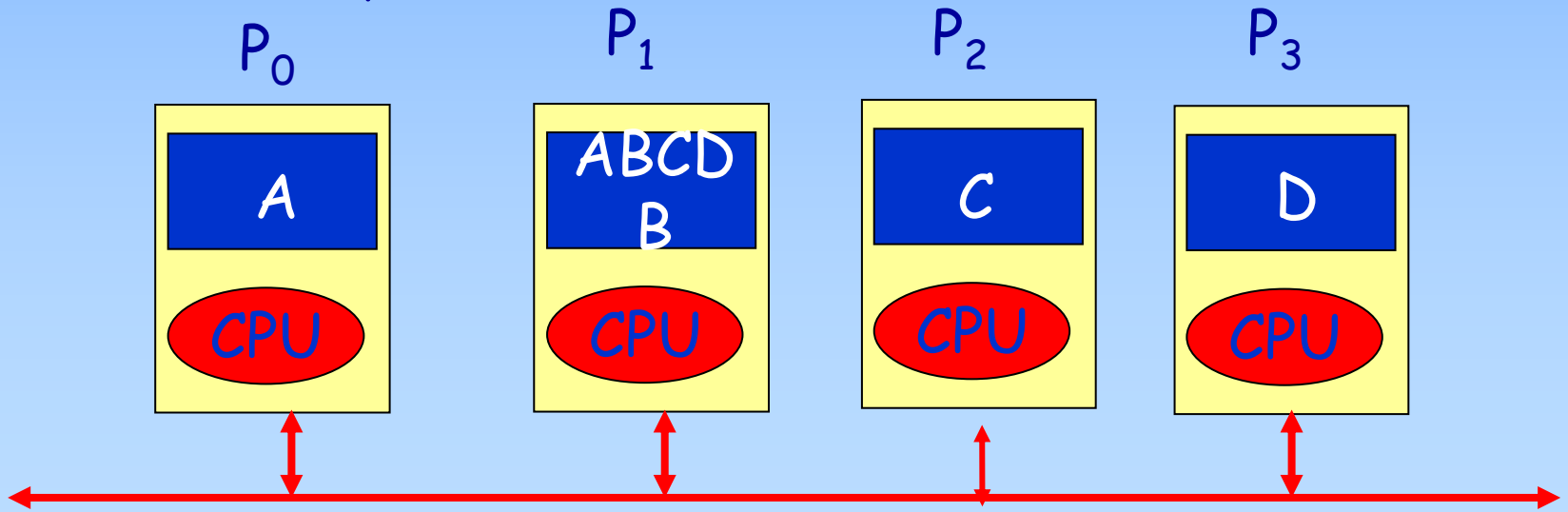
root identificativo del processo che riceve da tutti

comm identificativo del communicator

Operazione collettiva di tipo: *data scatter*



Un solo processo distribuisce i propri dati agli altri processi, se stesso compreso.



Lo *scatter* in MPI 1/2

```
MPI_Scatter(void *send_buff, int send_count,
            MPI_Datatype send_type,
            void *recv_buff,int recv_count,
             MPI_Datatype recv_type,
            int root, MPI_Comm comm);
```

- Il processo con identificativo `root` *distribuisce* i dati contenuti in `send_buff`.

- Il contenuto di `send_buff` viene suddiviso in `nproc` segmenti ciascuno di lunghezza `send_count`

- Il primo segmento viene affidato al processo con identificativo 0, il secondo al processo con identificativo 1, il terzo al processo con identificativo 2, etc.

Lo *scatter* in MPI: in dettaglio 2/2

```
MPI_Scatter(void *send_buff, int send_count,
            MPI_Datatype send_type,
            void *recv_buff,int recv_count,
            MPI_Datatype recv_type,
            int root, MPI_Comm comm);
```

***send_buff** indirizzo del dato da spedire

send_count numero dei dati da spedire

send_type tipo dei dati da spedire

***recv_buff** indirizzo del dato in cui ricevere

recv_count numero dei dati da ricevere

recv_type tipo dei dati da ricevere

root identificativo del processo che spedisce a tutti

comm identificativo del communicator

FINE LEZIONE