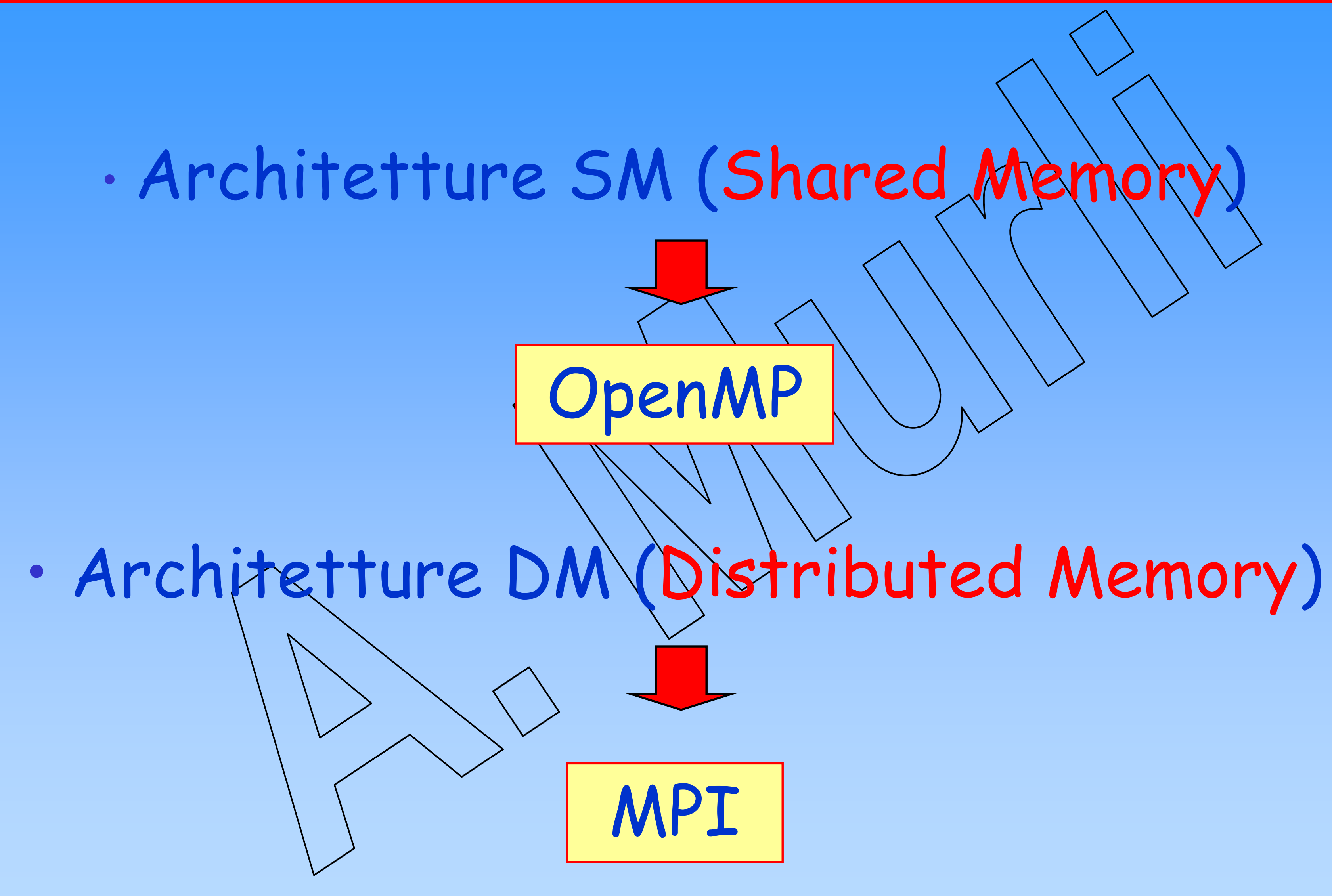
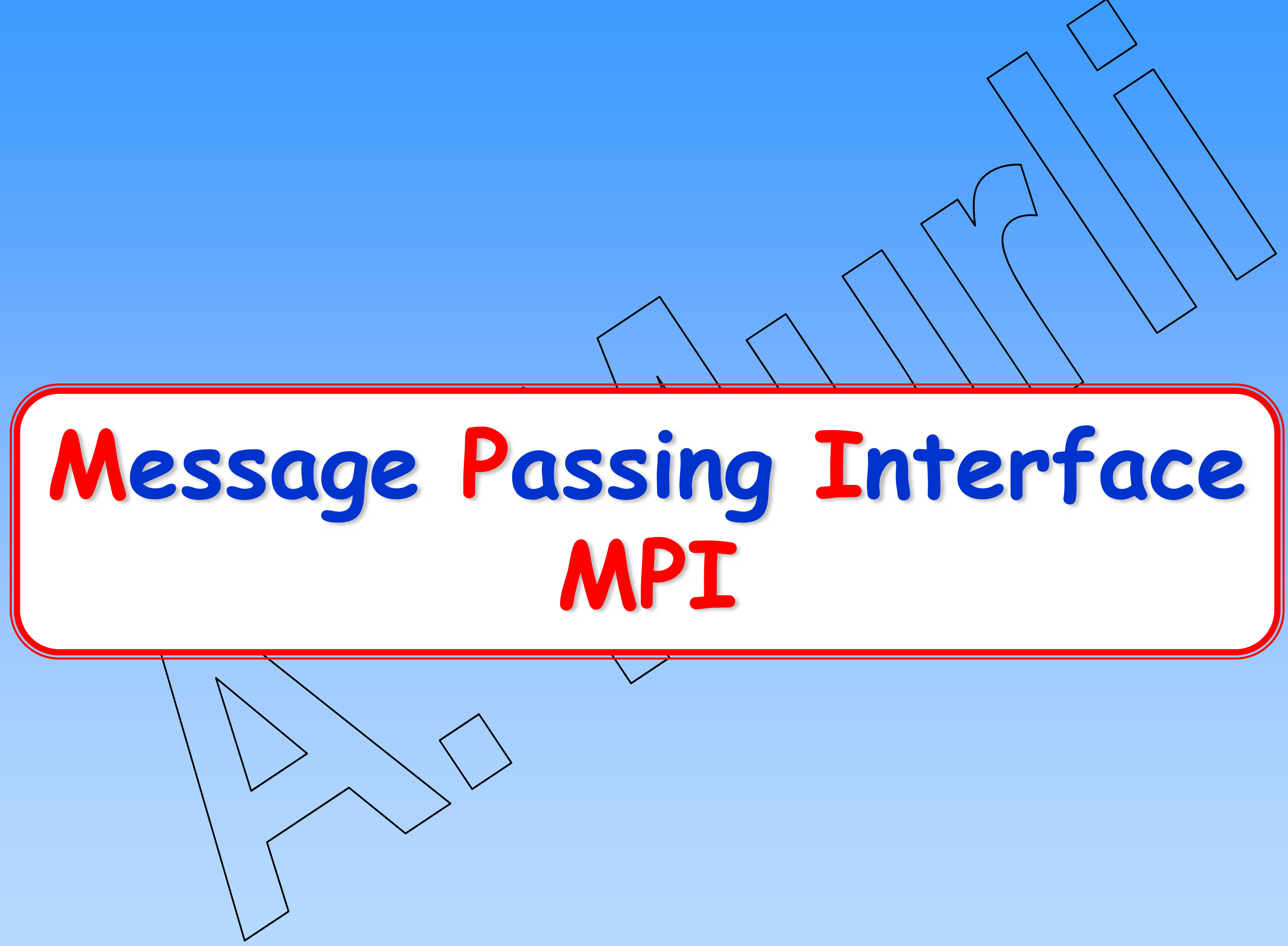
Alcuni strumenti per lo sviluppo di software su architetture MIMD

Calcolatori MIMD

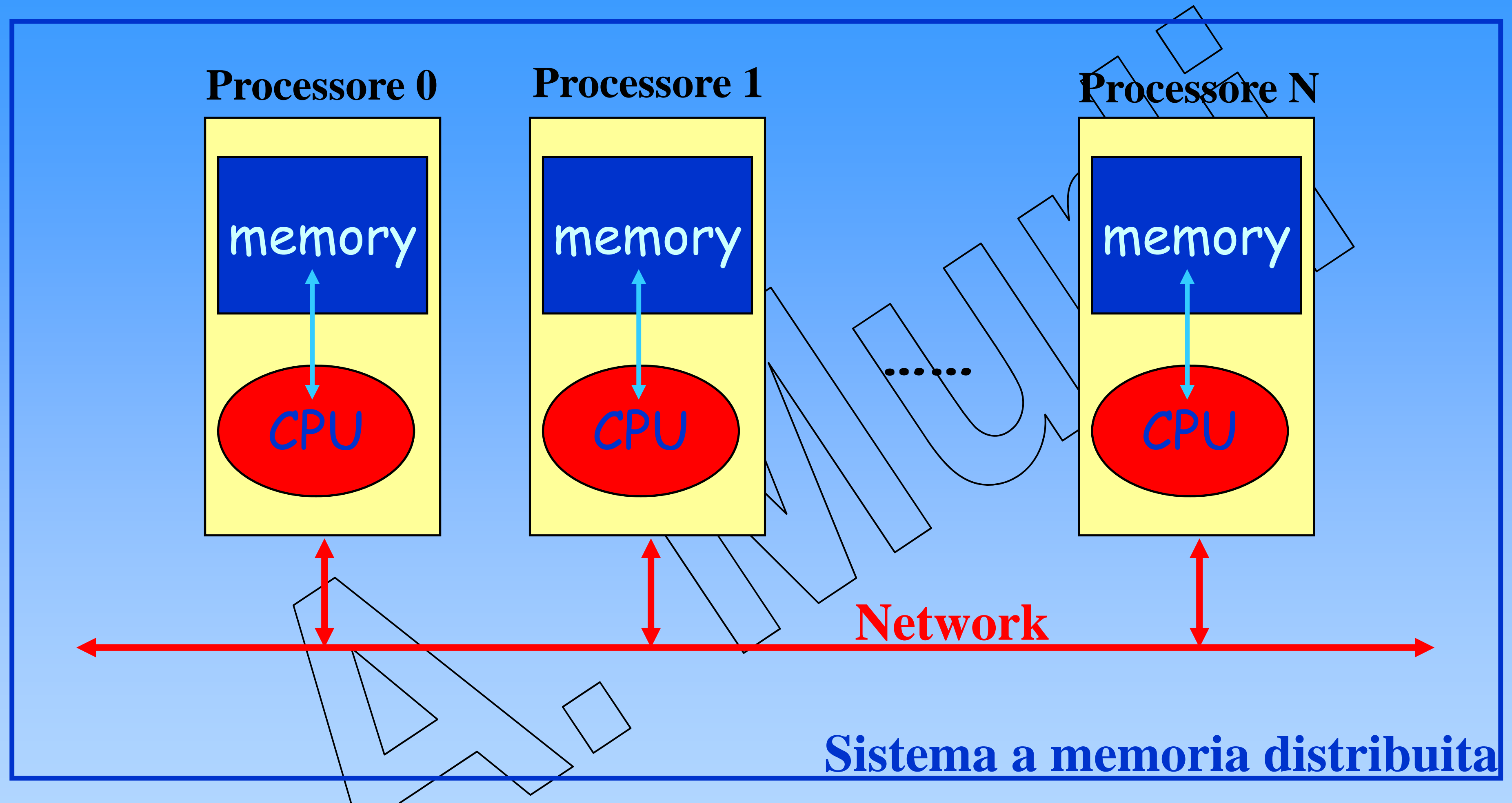


A. Murli



Message Passing Interface
MPI

Paradigma del *message passing*



Ogni processore ha una propria memoria locale alla quale accede direttamente.

Ogni processore può conoscere i dati in memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.

Definizione: *message passing*

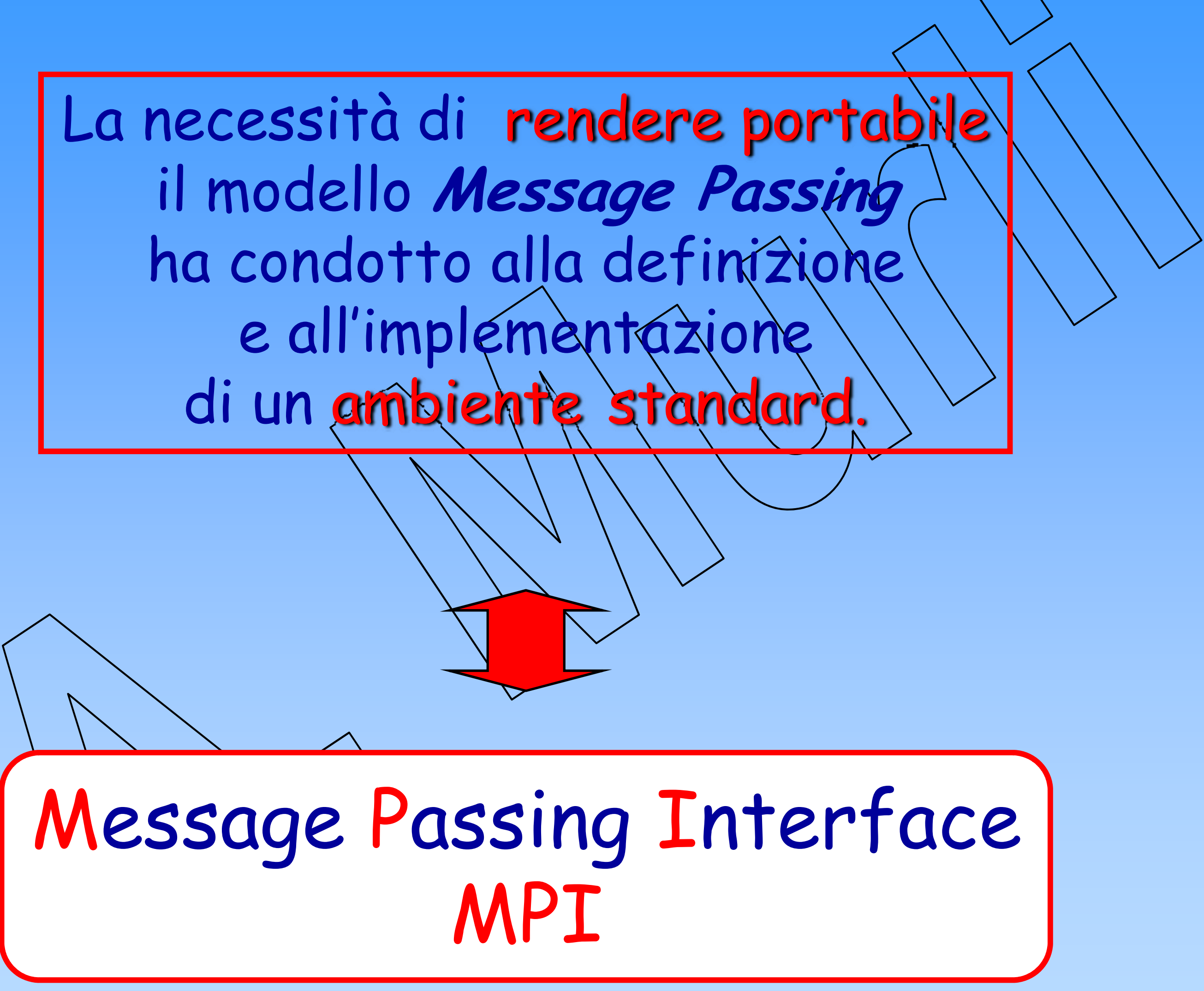
... ogni processore può conoscere i dati nella memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.

Message Passing :
modello per la progettazione
di software in
ambiente di Calcolo Parallelo.

Murli

Lo standard

La necessità di **rendere portabile** il modello ***Message Passing*** ha condotto alla definizione e all'implementazione di un **ambiente standard.**



Message Passing Interface
MPI

Per cominciare…

```
…
int main()
{

 …
 sum=0;
 for i= 0 to 14 do
  sum=sum+a[i];
 endfor
…
return 0;
}
```

In ambiente **MPI** un programma è visto come un insieme di componenti (o processi) concorrenti

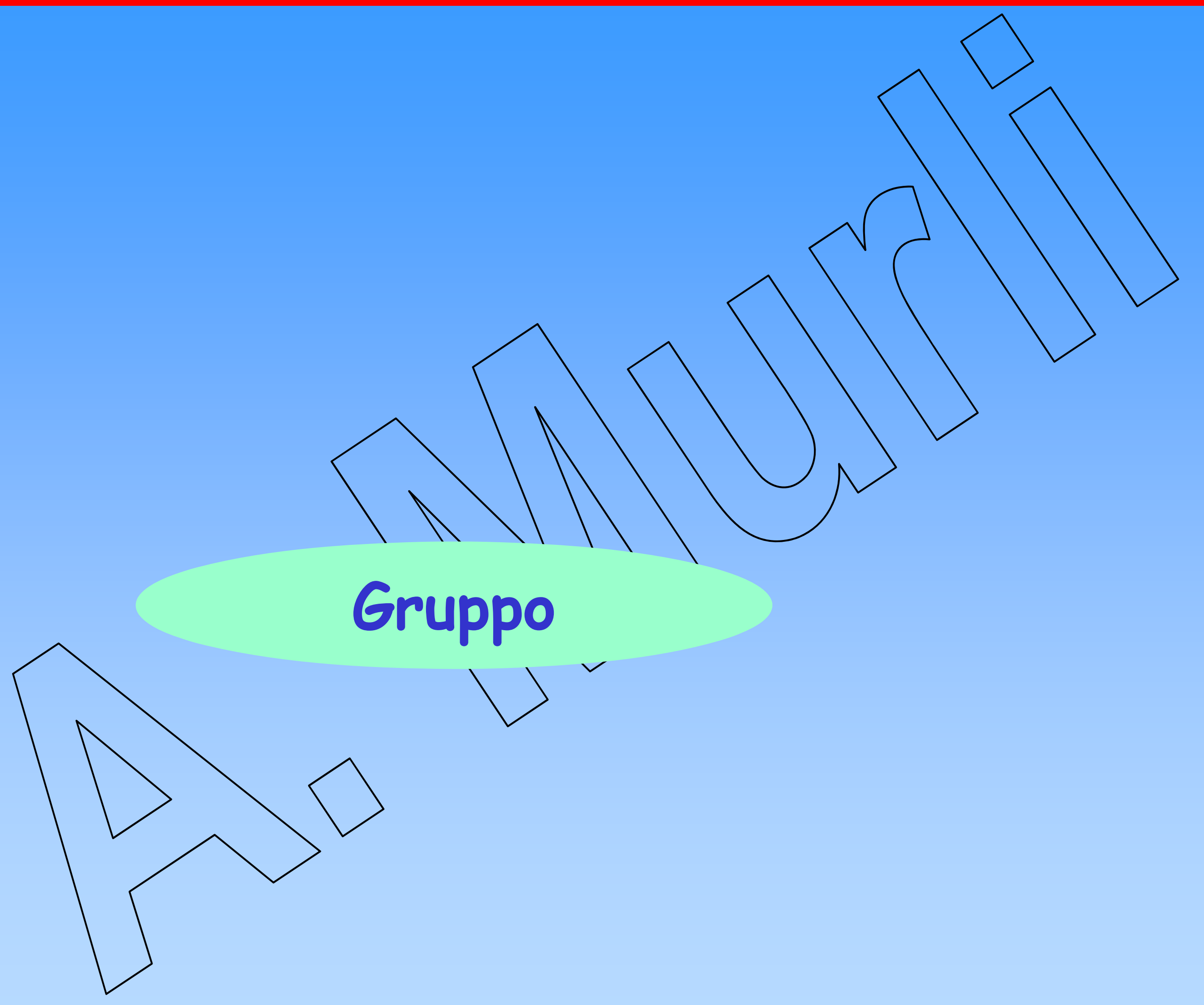
```
…
main()
{
  …
  sum=0;
  for i=0 to 4 do
   sum=sum+a[i];
  endfor
…
return 0;
}
```

```
…
main()
{
  …
  sum=0;
  for i=5 to 9 do
   sum=sum+a[i];
  endfor
…
return 0;
}
```

```
…
main()
{
  …
  sum=0;
  for i=10 to 14do
   sum=sum+a[i];
  endfor
…
return 0;
}
```

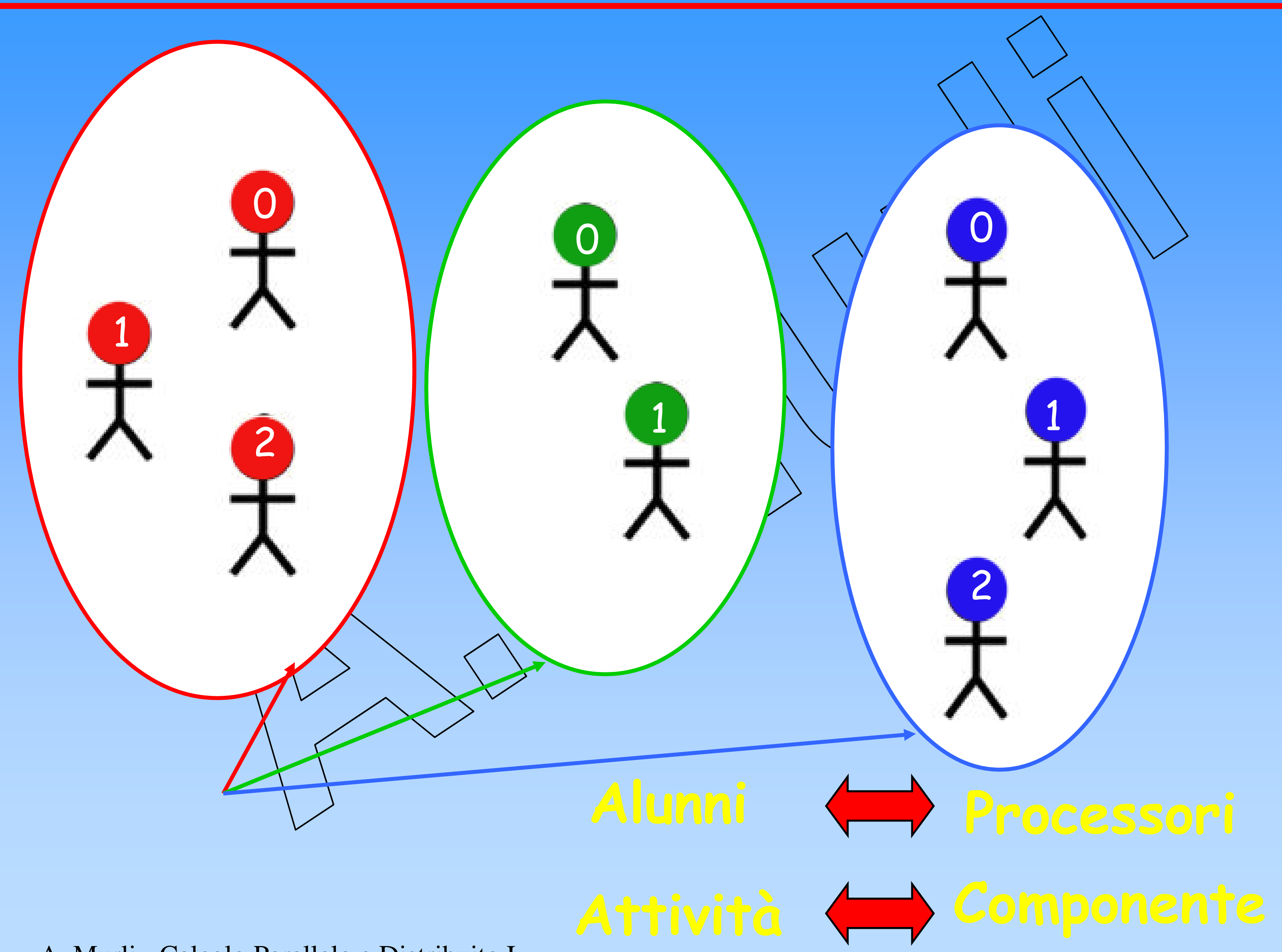
Murli

In MPI i processori sono raggruppati in ….:

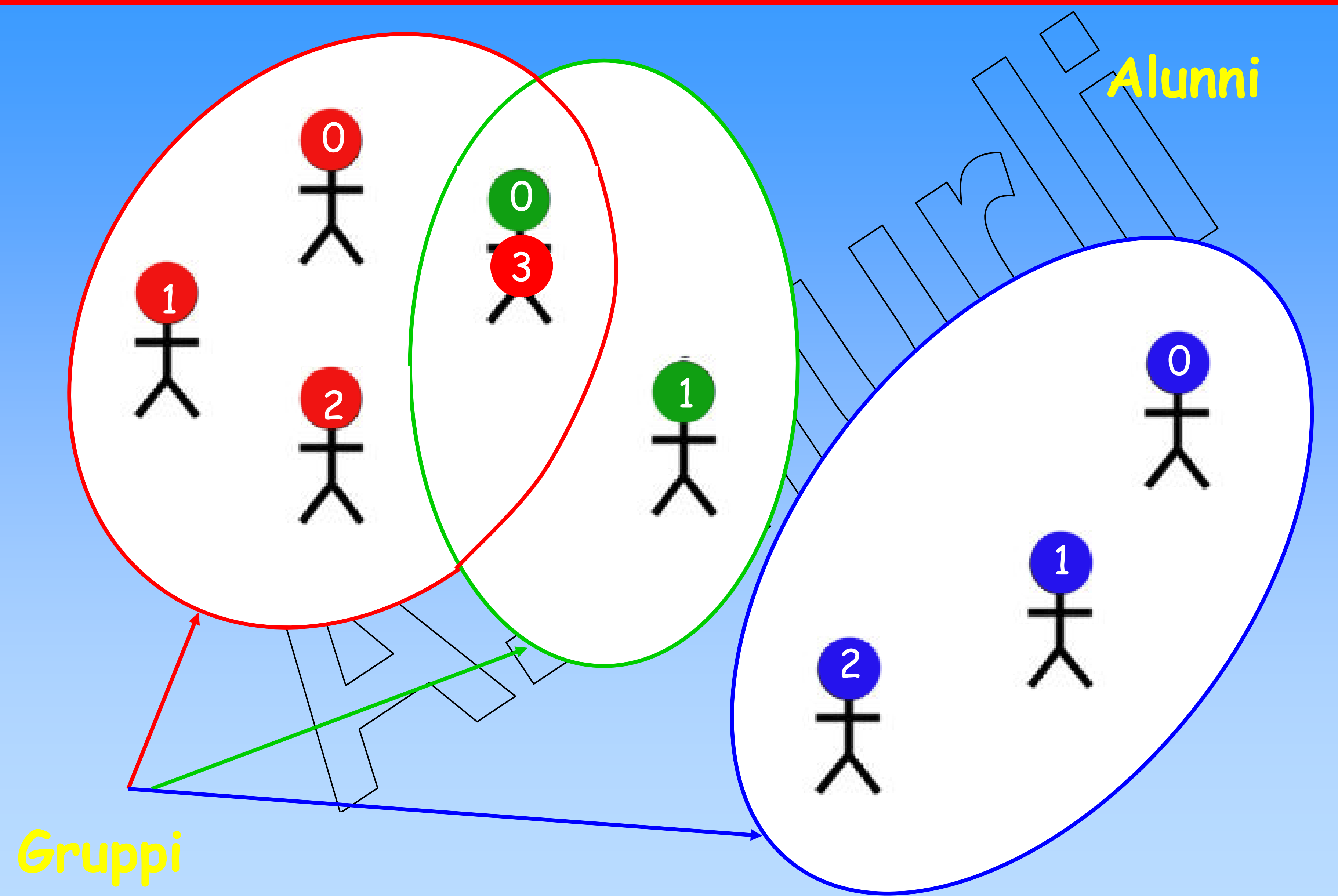


A. Murli

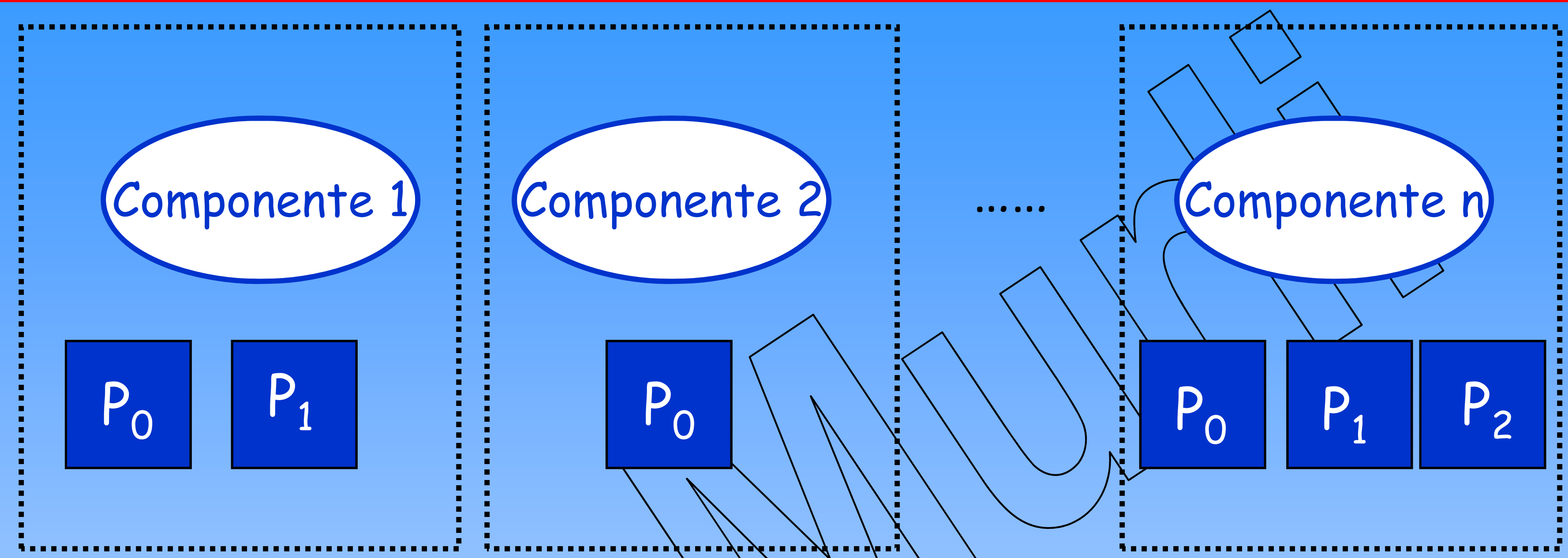
Gruppi e Componenti MPI … :



Gruppi e Componenti MPI … :



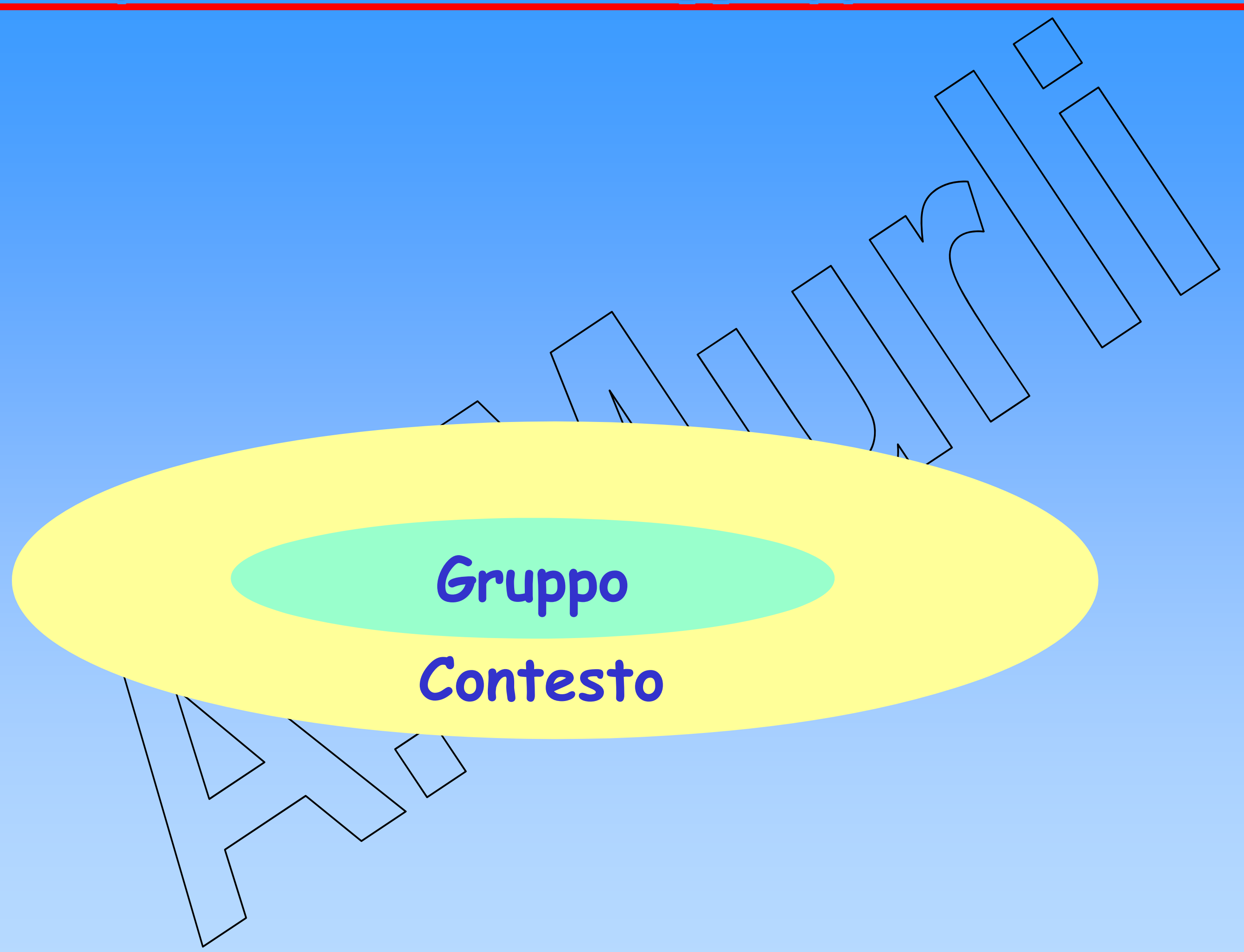
Alcuni concetti di base: GRUPPO…



- Ogni componente concorrente del programma viene affidata ad un ***gruppo di processori***.
- In ciascun gruppo ad ogni processore è associato un ***identificativo***.
- L'***identificativo*** è un intero, compreso tra 0 ed il numero totale di processori appartenenti ad un gruppo decrementato di una unità.
- Processori appartenenti a uno o più gruppi possono avere identificativi diversi, ciascuno relativo ad uno specifico gruppo

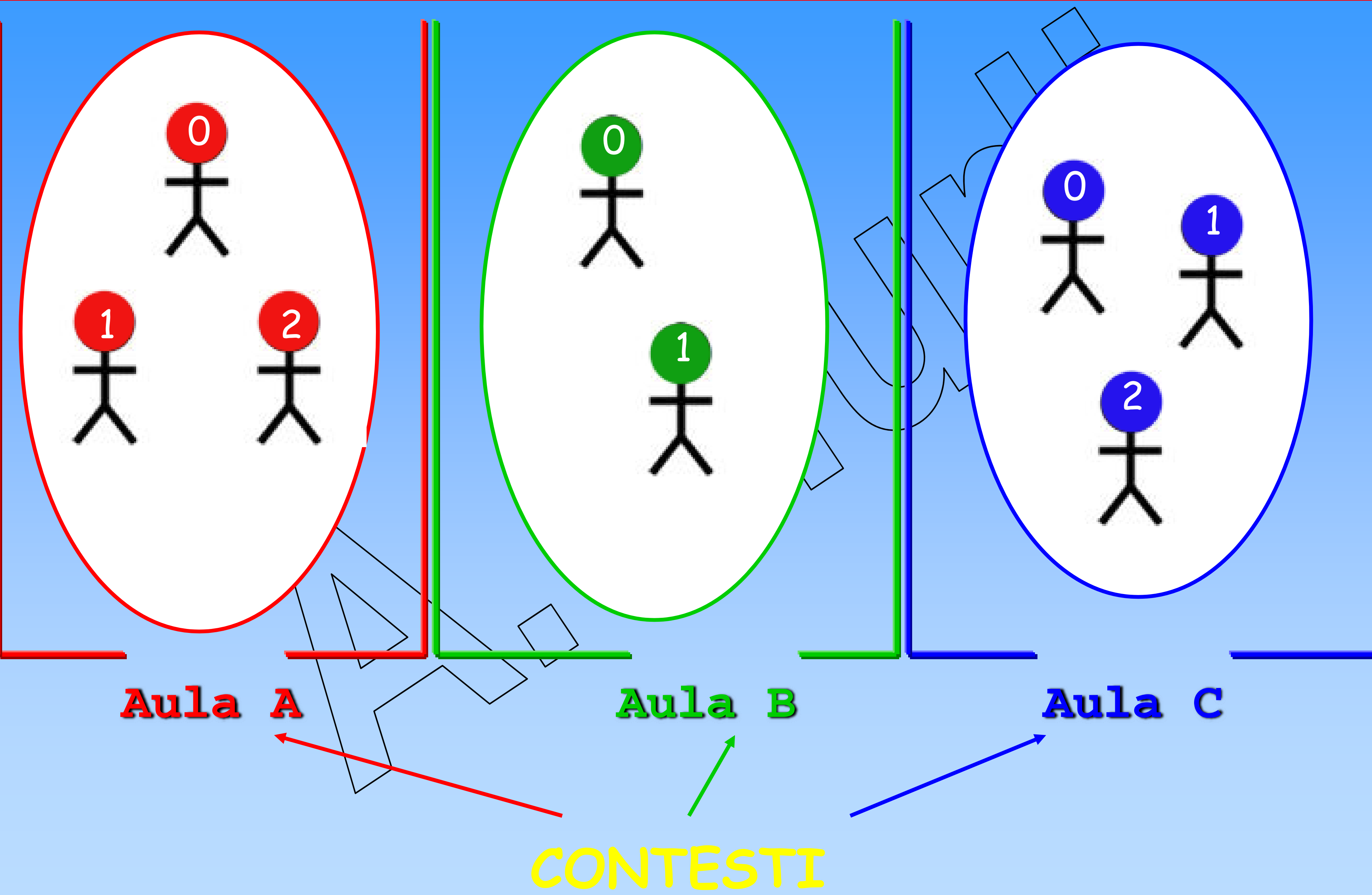
A. Murli

In MPI i processori sono raggruppati in ….:

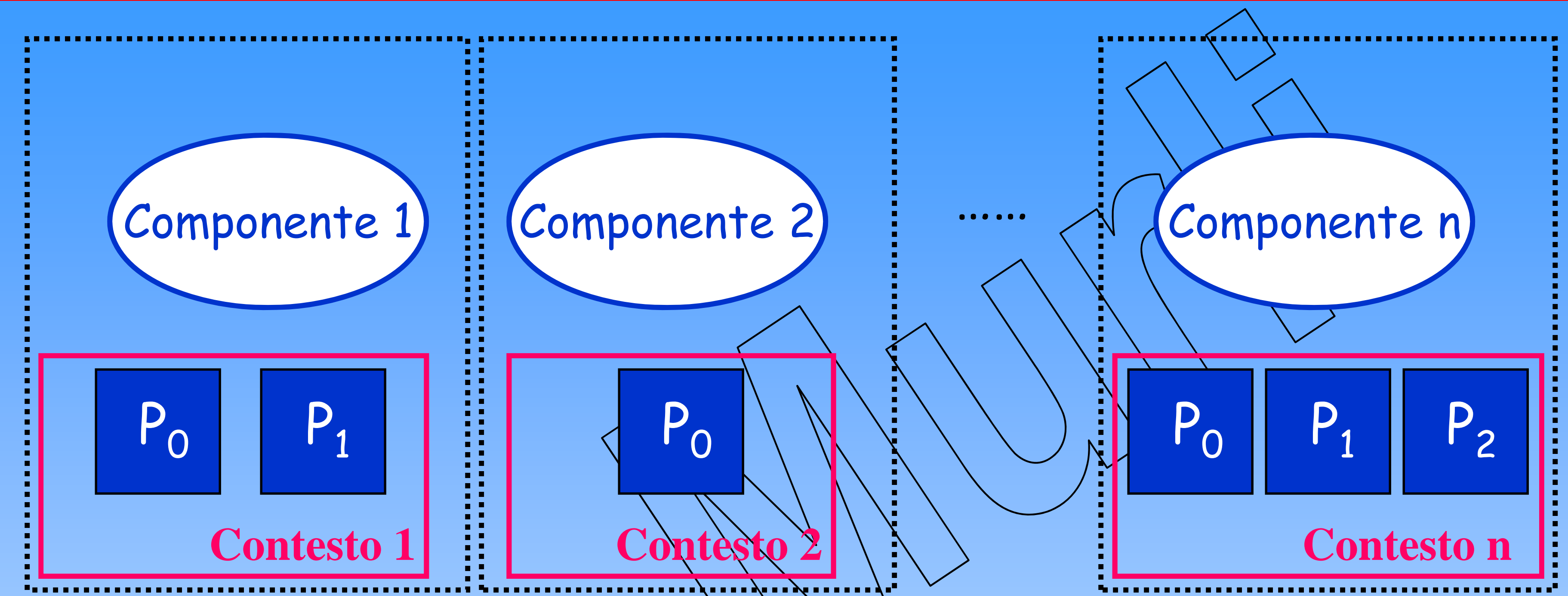


A. Murli

Il CONTESTO MPI :



Alcuni concetti di base: CONTESTO



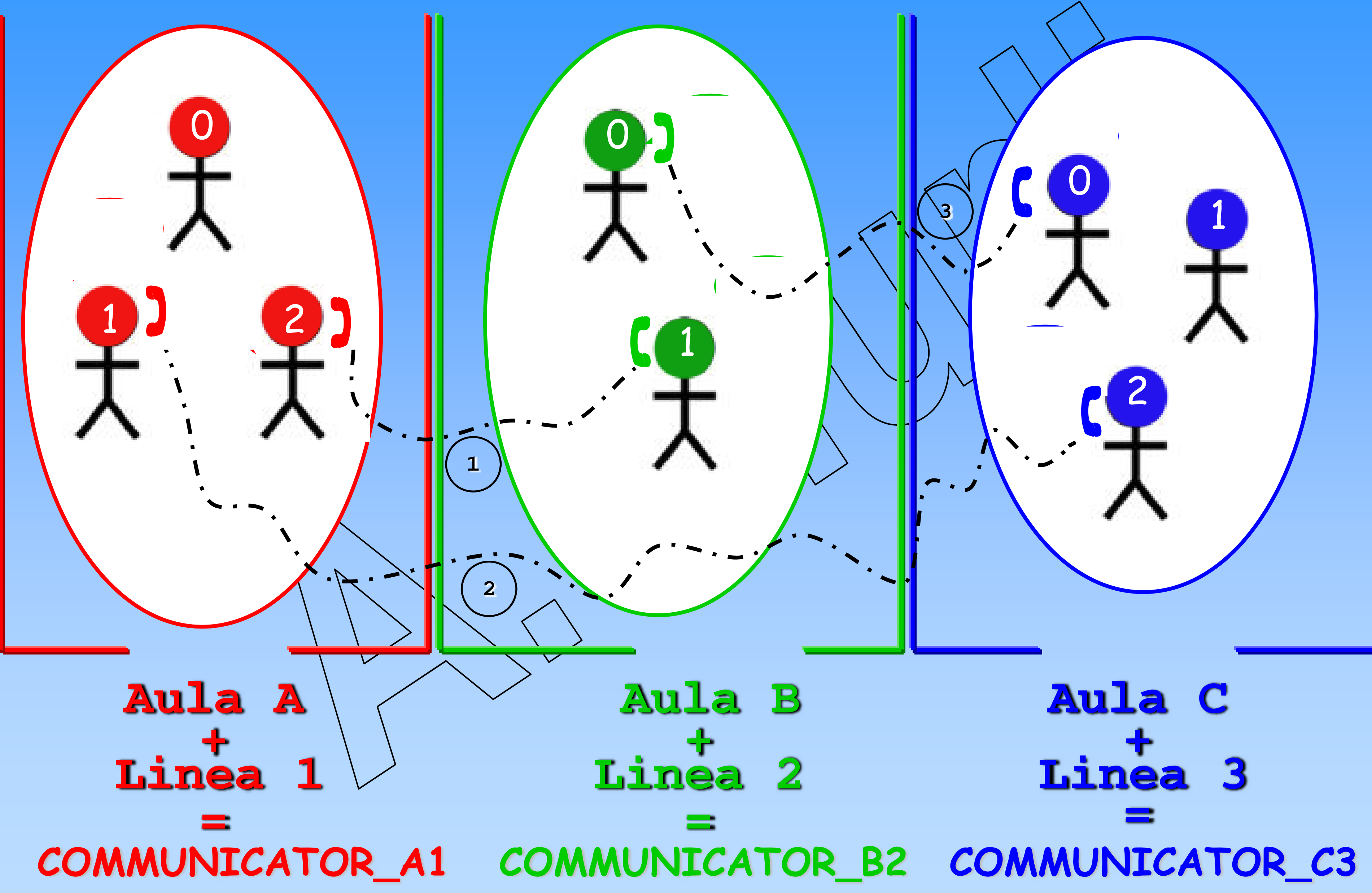
- A ciascun gruppo di processori viene attribuito un identificativo detto ***contesto***.
- Il ***contesto*** definisce l'ambito in cui avvengono le comunicazioni tra processori di uno stesso gruppo.
- Se la spedizione di un messaggio avviene in un ***contesto***, la ricezione deve avvenire nello **stesso** contesto.

A. Murli

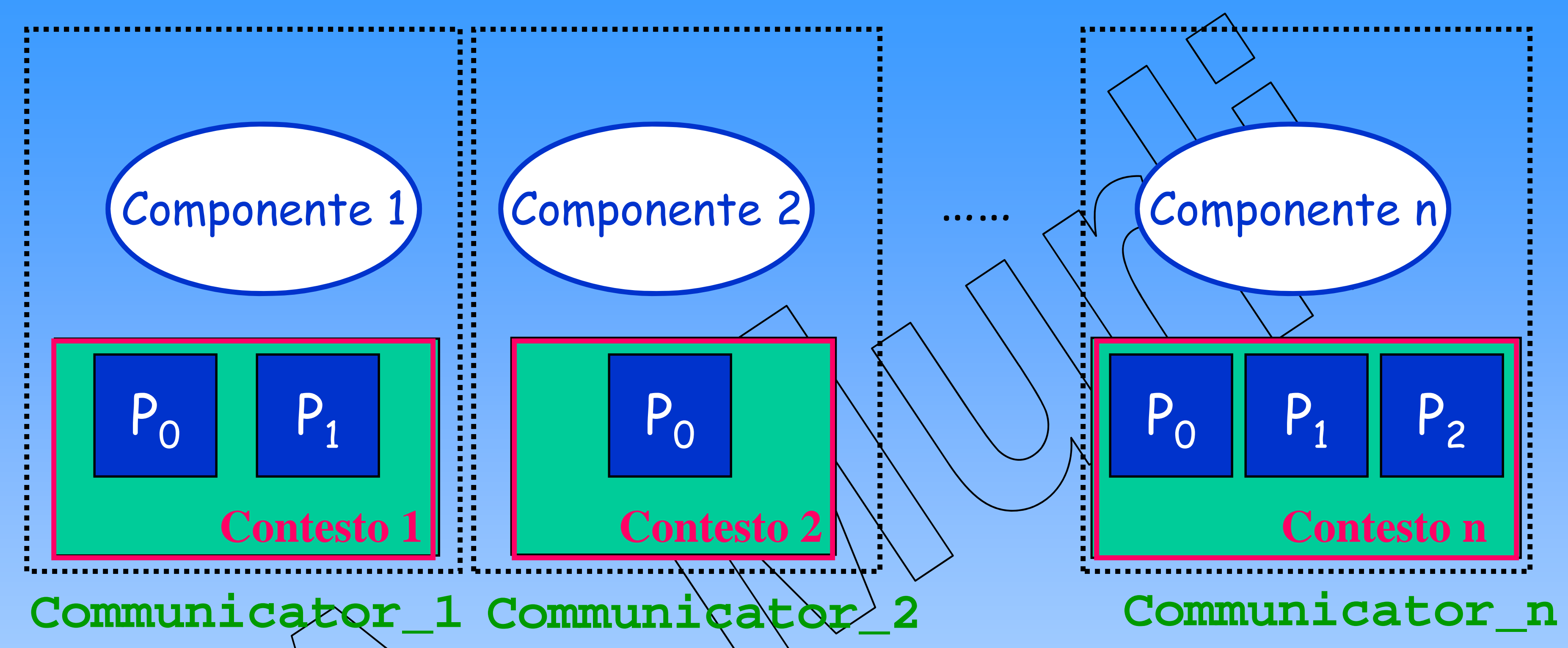
In MPI i processori sono raggruppati in ….:

A. Murli

Il Communicator MPI :



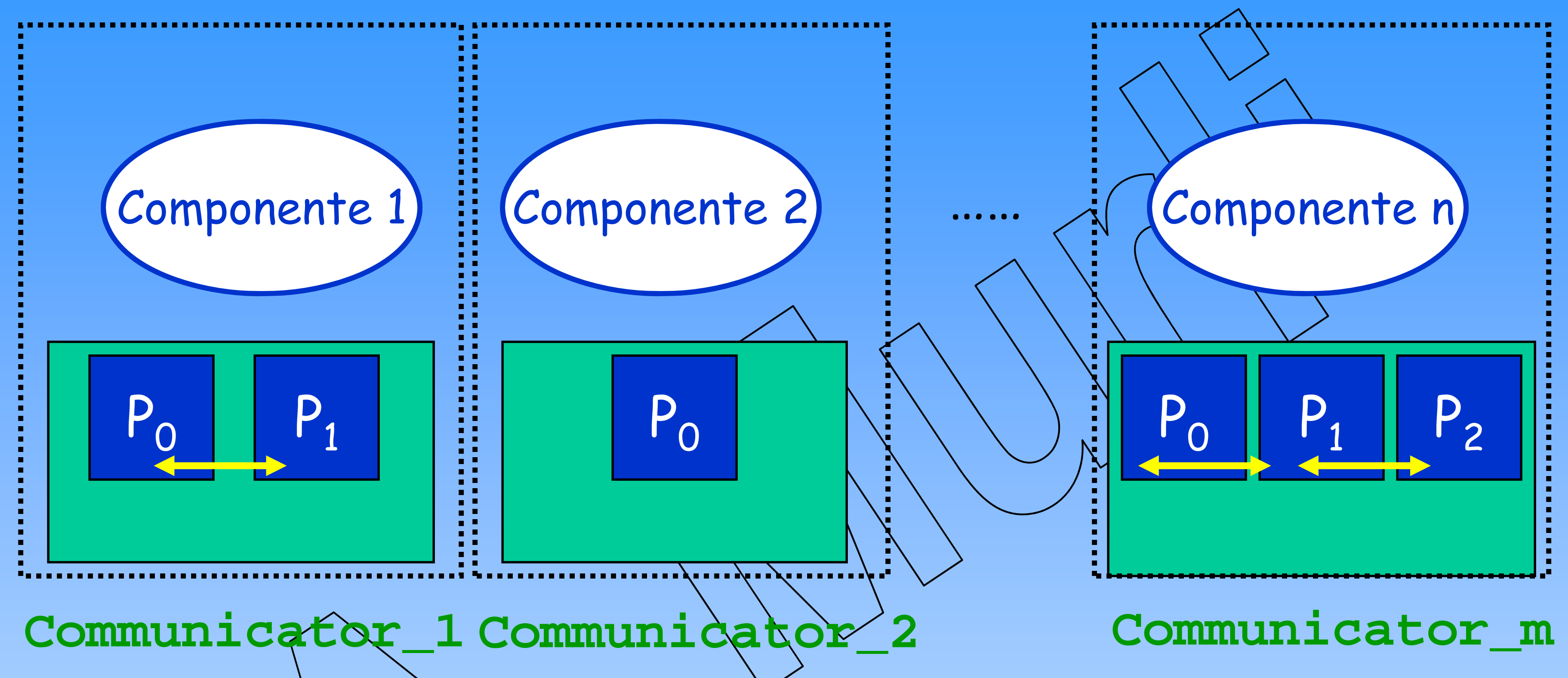
Alcuni concetti di base: *COMMUNICATOR…*



- Ad un gruppo di processori appartenenti ad uno stesso contesto viene assegnato un ulteriore identificativo: il ***communicator***.

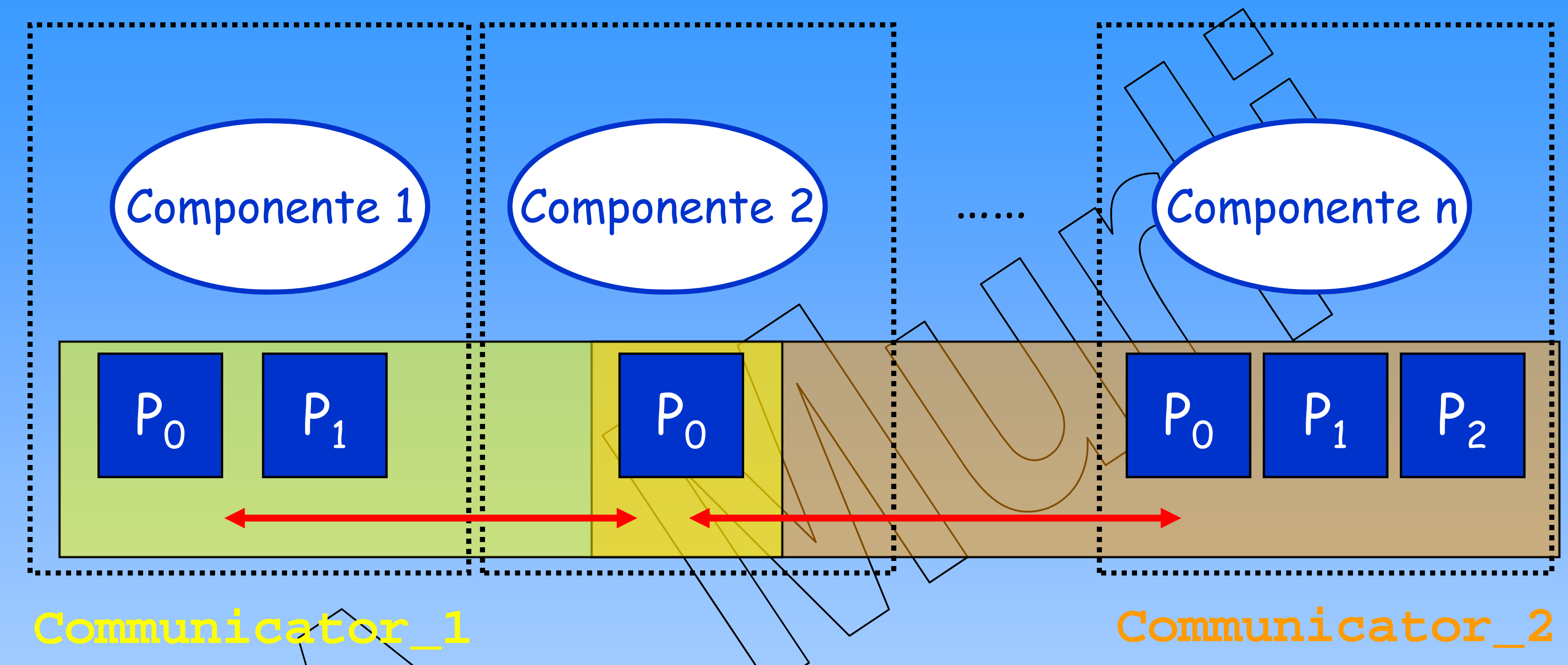
- Il ***communicator*** racchiude tutte le caratteristiche dell'ambiente di comunicazione: topologia, quali contesti coinvolge, ecc…

…Alcuni concetti di base: COMMUNICATOR…



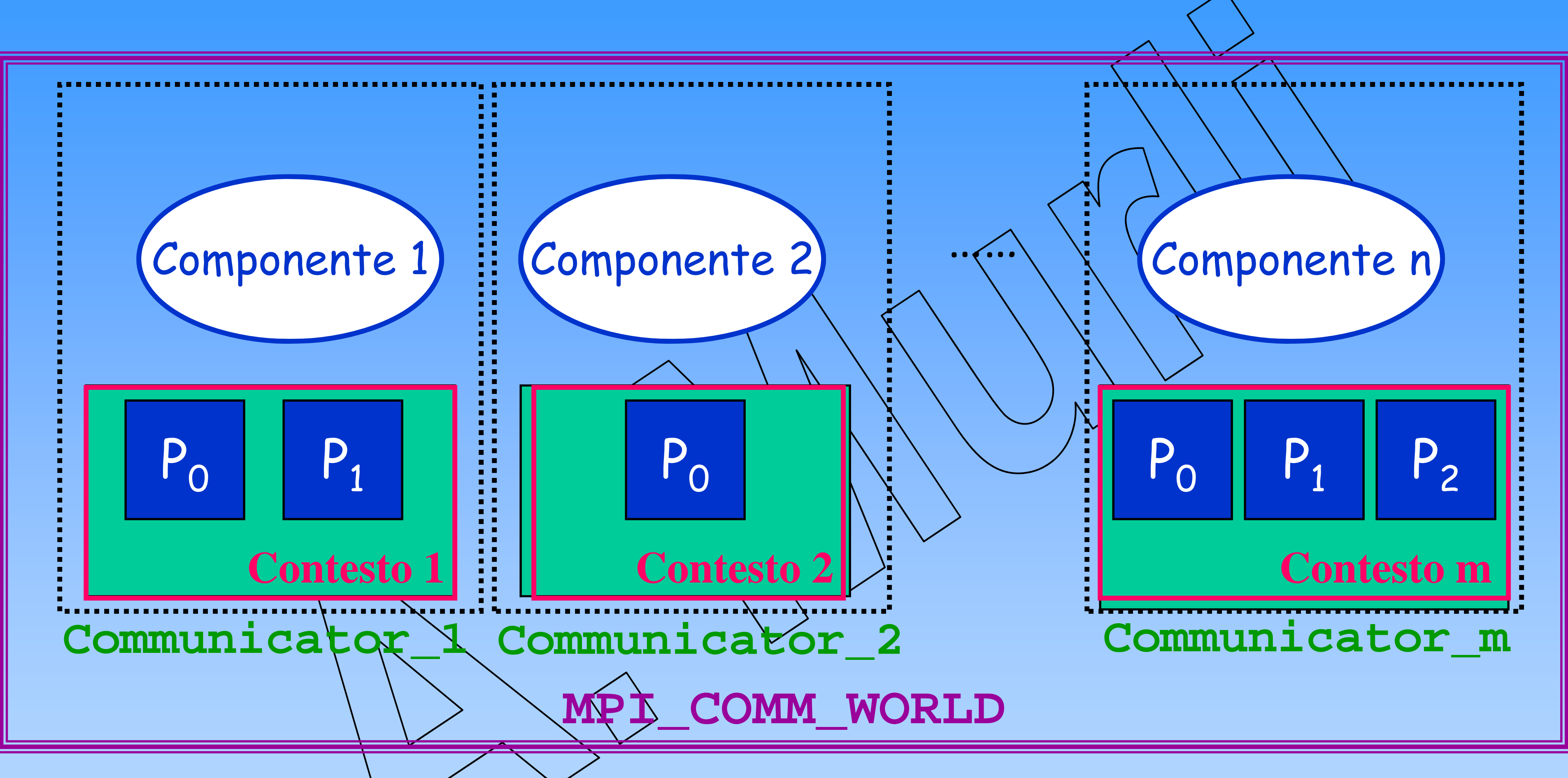
- Esistono due tipi principali di *communicator*.
- L'***intra-communicator*** in cui le comunicazioni avvengono all'interno di un gruppo di processori.

…Alcuni concetti di base: COMMUNICATOR…



- Esistono due tipi principali di *communicator*.
- L'***intra-communicator*** in cui le comunicazioni avvengono all'interno di un gruppo di processori.
- L'***inter-communicator*** in cui le comunicazioni avvengono tra gruppi di processori.

…Alcuni concetti di base: COMMUNICATOR



- Tutti i processori fanno parte per default di un unico communicator detto **MPI_COMM_WORLD**.

Le funzioni di MPI

Murli

MPI è una libreria che comprende:

- Funzioni per definire l'***ambiente***
- Funzioni per ***comunicazioni uno a uno***
- Funzioni per ***comunicazioni collettive***
- Funzioni per ***operazioni collettive***

Le funzioni dell'ambiente.

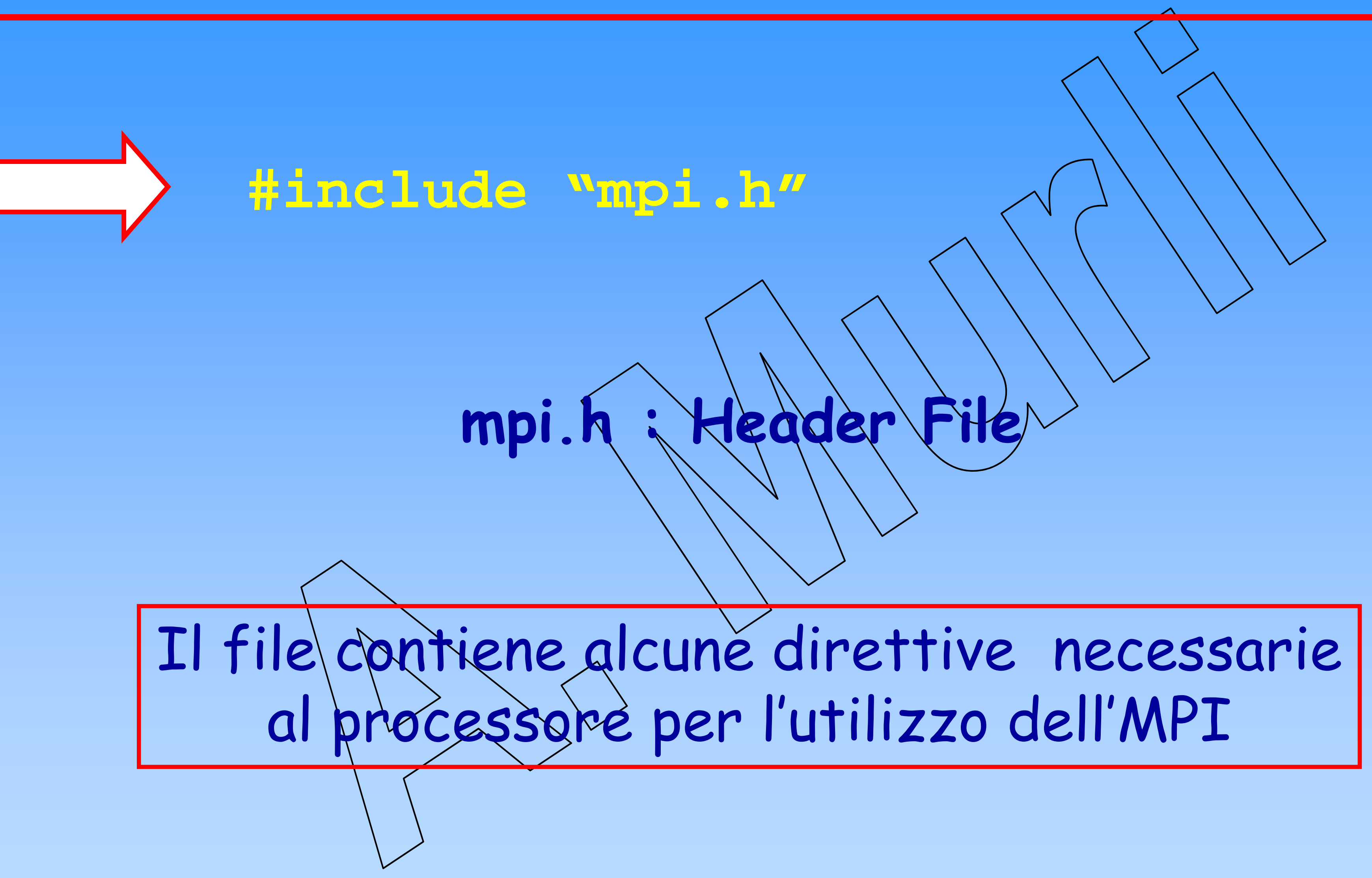
A. Murli

Un semplice programma :

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```

Tutti i processori di **MPI_COMM_WORLD** stampano a video il proprio identificativo **menum** ed il numero di processori **nproc**.

Nel programma … :



```
#include "mpi.h"
```

mpi.h : Header File

Il file contiene alcune direttive necessarie al processore per l'utilizzo dell'MPI

A. Murli

Nel programma ... :

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

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```

Tutti i processori di **MPI_COMM_WORLD** stampano a video il proprio identificativo **menum** ed il numero di processori **nproc**.

Nel programma … :

MPI_Init(&argc,&argv);

- Inizializza l'ambiente di esecuzione **MPI**
- Inizializza il communicator **MPI_COMMON_WORLD**
- I due dati di input: **argc** ed **argv** sono gli argomenti del `main`

In generale … :

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

Input: argc, **argv;

- Questa routine inizializza l'ambiente di esecuzione di MPI. Deve essere chiamata una sola volta, prima di ogni altra routine MPI.
- Definisce l'insieme dei processori attivati (communicator).

A. Murli

Nel programma … :

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

int main(int argc, char *argv[])
{
     int menum, nproc;
     MPI_Init(&argc,&argv);

     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     MPI_Comm_size(MPI_COMM_WORLD,&nproc);

     printf("Sono %d di %d\n",menum,nproc);

     MPI_Finalize();
     return 0;
}
```

Tutti i processori di **MPI_COMM_WORLD** stamperanno sul video il proprio identificativo **menum** ed il numero di processori **nproc**.

Nel programma … :

MPI_Finalize();

- Questa routine determina la **fine** del programma MPI.
- Dopo questa routine **non** è possibile richiamare nessun'altra routine di MPI.

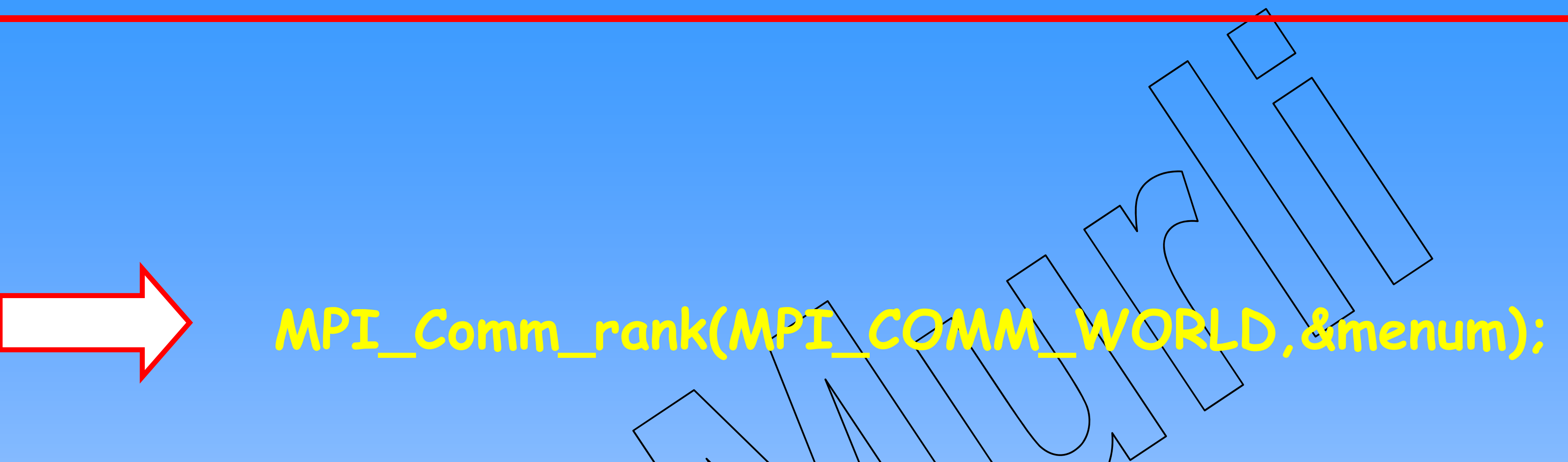
A. Murli

Nel programma … :

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

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```

Tutti i processori di **MPI_COMM_WORLD** stamperanno sul video il proprio identificativo **menum** ed il numero di processori **nproc**.

Nel programma ... :



- Questa routine permette al processore chiamante, appartenente al communicator MPI_COMM_WORLD, di memorizzare il proprio identificativo nella variabile menum.

A. Murli

In generale … :

```
MPI_Comm_rank(MPI_Comm comm, int *menum);
```

Input: comm ;
Output: menum.

- Fornisce ad ogni processore del communicator **comm** l'identificativo **menum**.

A. Murli

Nel programma … :

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

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```

Tutti i processori di **MPI_COMM_WORLD** stamperanno sul video il proprio identificativo **menum** ed il numero di processori **nproc**.

Nel programma … :

- Questa routine permette al processore chiamante di memorizzare nella variabile **nproc** il numero totale dei processori concorrenti appartenenti al communicator **MPI_COMM_WORLD**.

A. Murli

In generale … :

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

Input: comm ;

Output: nproc.

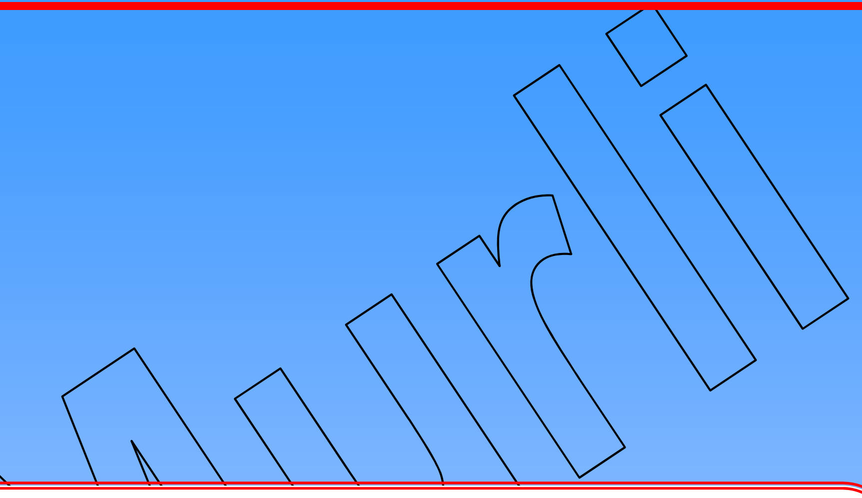
- Ad ogni processore del communicator **comm** , restituisce in **nproc**, il numero totale di processori che costituiscono **comm**.
- Permette di conoscere quanti processori concorrenti possono essere utilizzati per una determinata operazione.

A. Murli

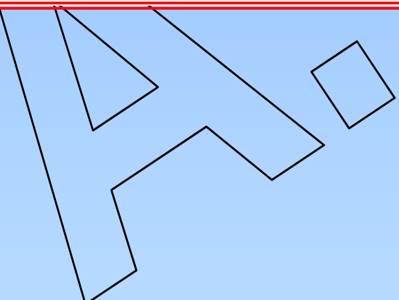
La comunicazione di un messaggio

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

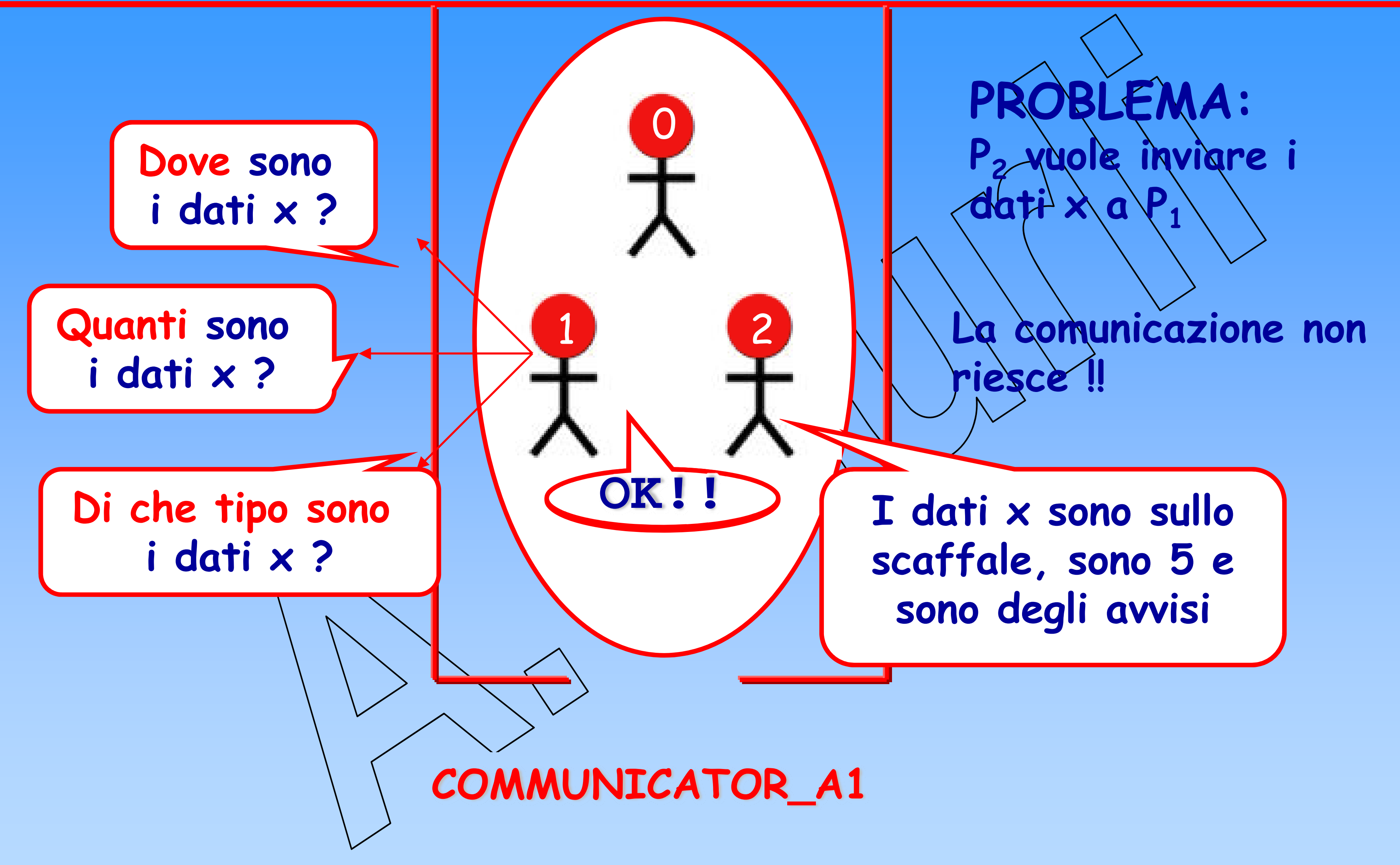
Per comunicazioni che coinvolgono solo due processori	→	Si considerano funzioni MPI per **comunicazioni uno a uno**
Per comunicazioni che coinvolgono più processori	→	Si considerano funzioni MPI per **comunicazioni collettive**



Comunicazione di un messaggio uno a uno

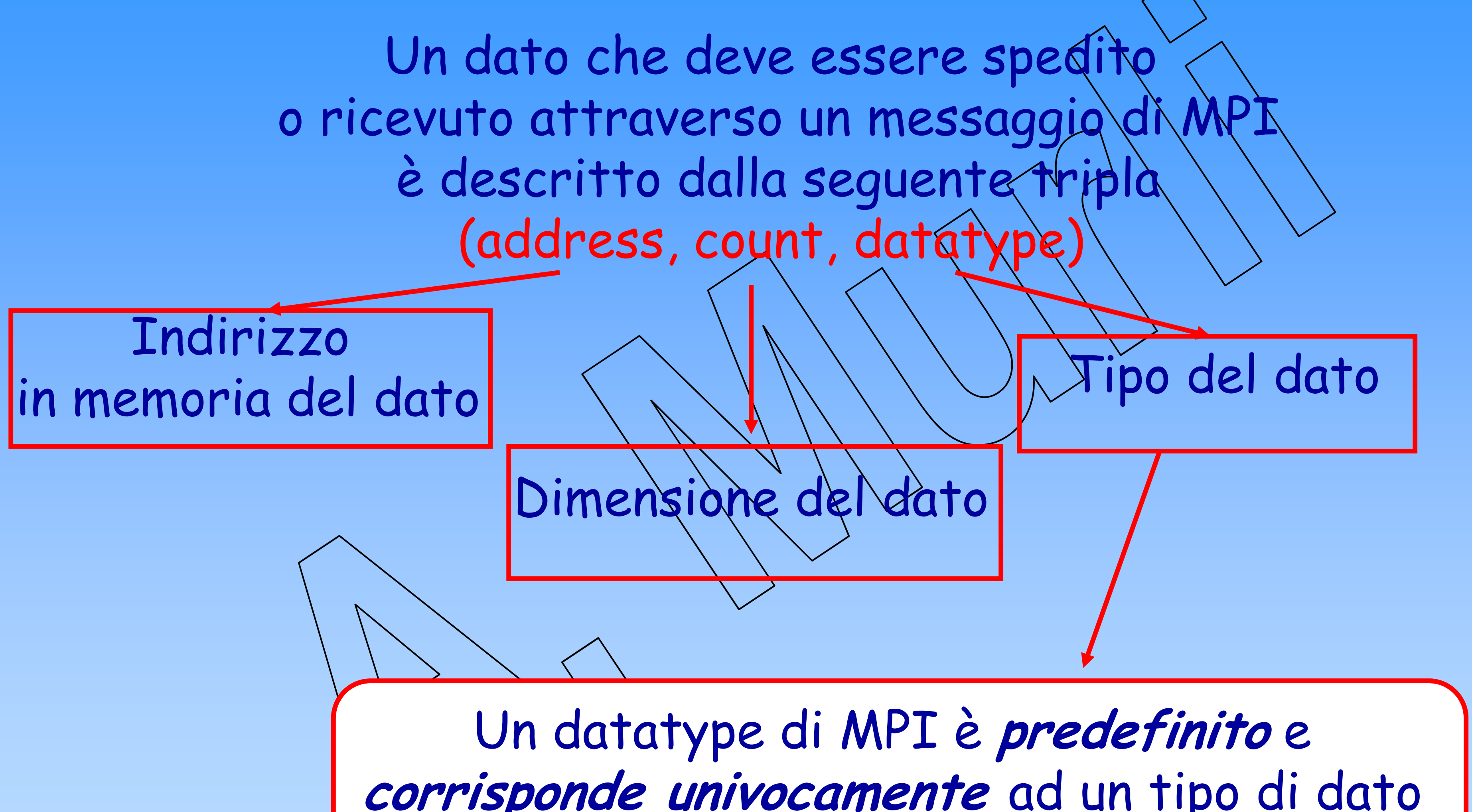


Il Communicator MPI :



Comunicazione Riuscita !

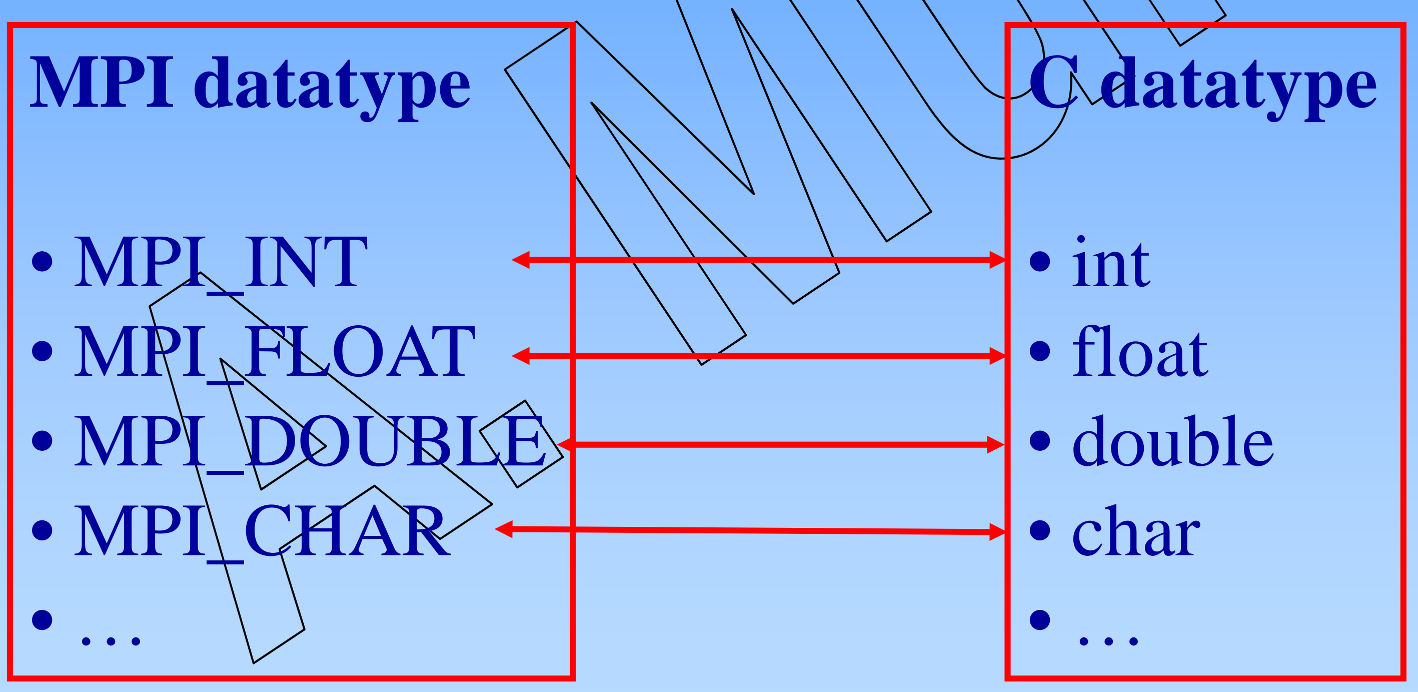
Caratteristiche di un messaggio



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

Esempio 1: linguaggio C

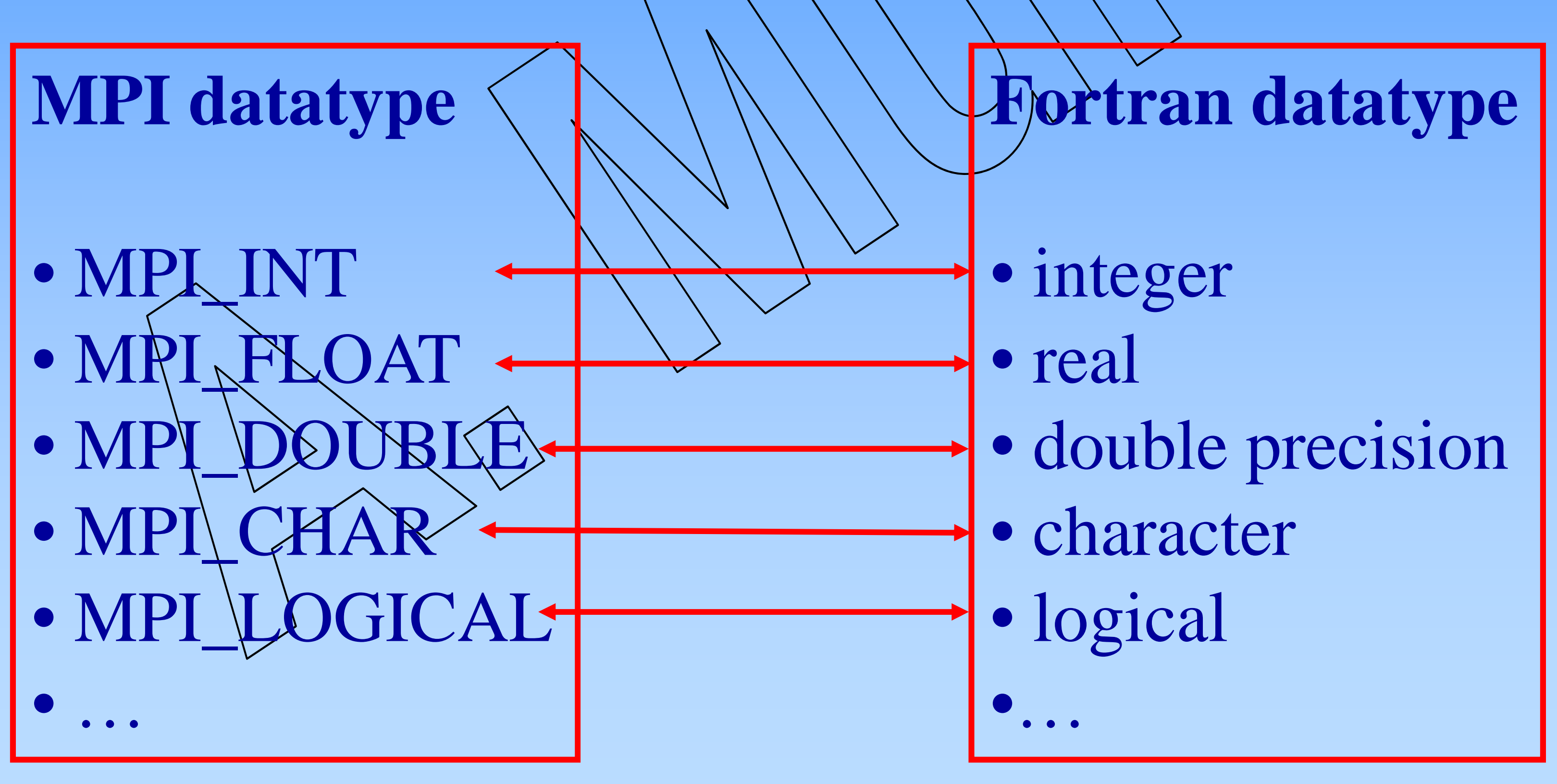
Ogni tipo di dato di MPI corrisponde **univocamente** ad un tipo di dato del linguaggio C.



AMurli

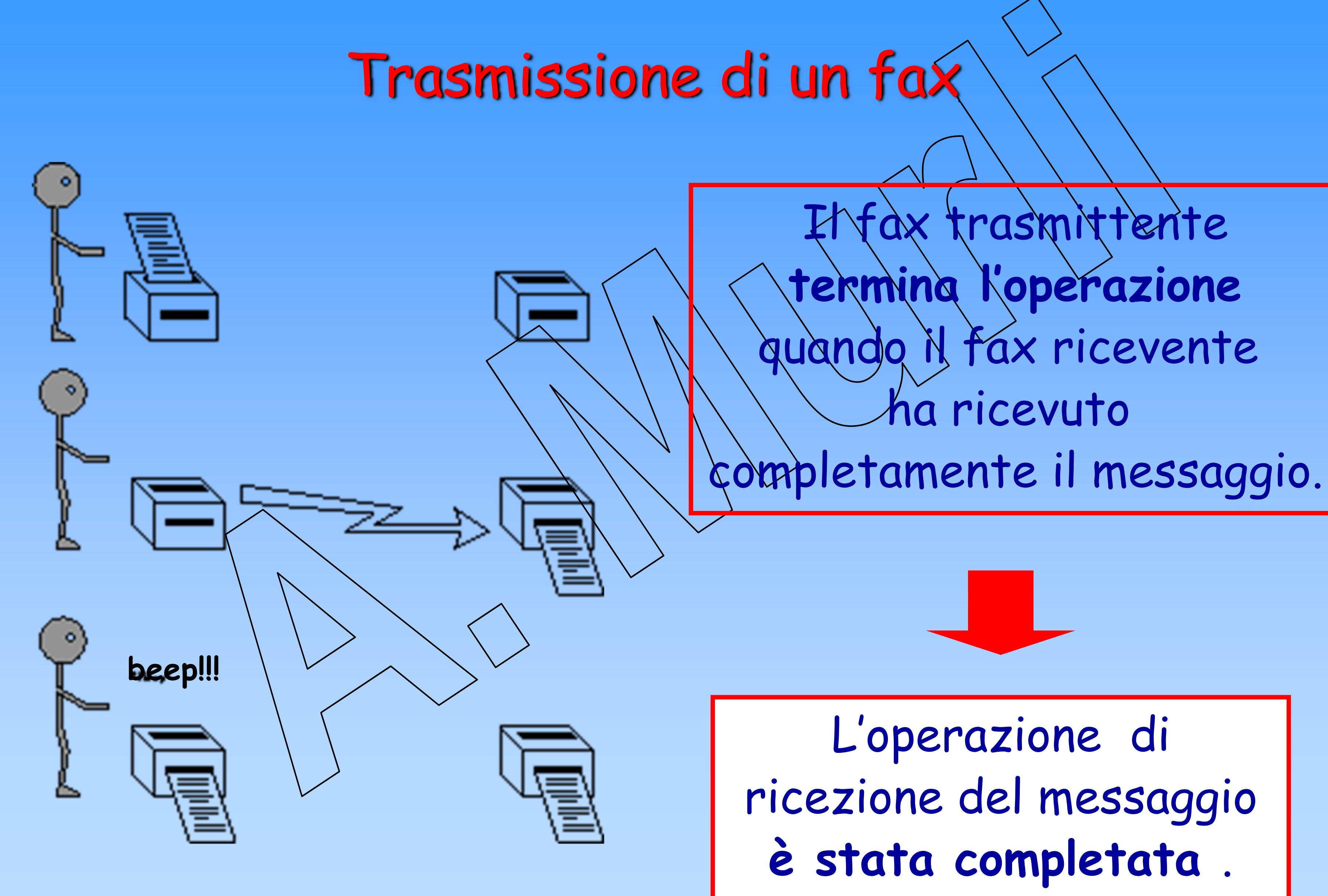
Esempio 2: linguaggio Fortran

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



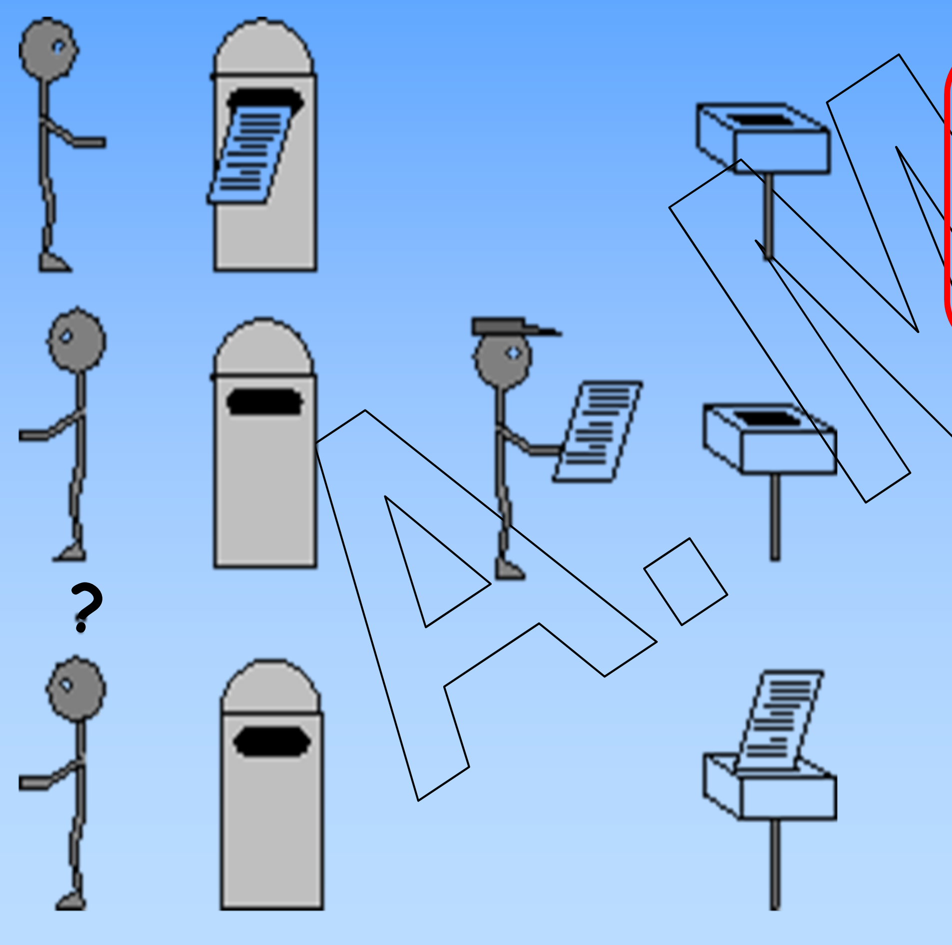
Il fax trasmittente **termina l'operazione** quando il fax ricevente ha ricevuto completamente il messaggio.

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

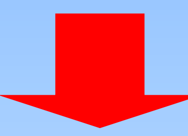
A. Murli

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.

A. Murli

Tipi di comunicazioni:

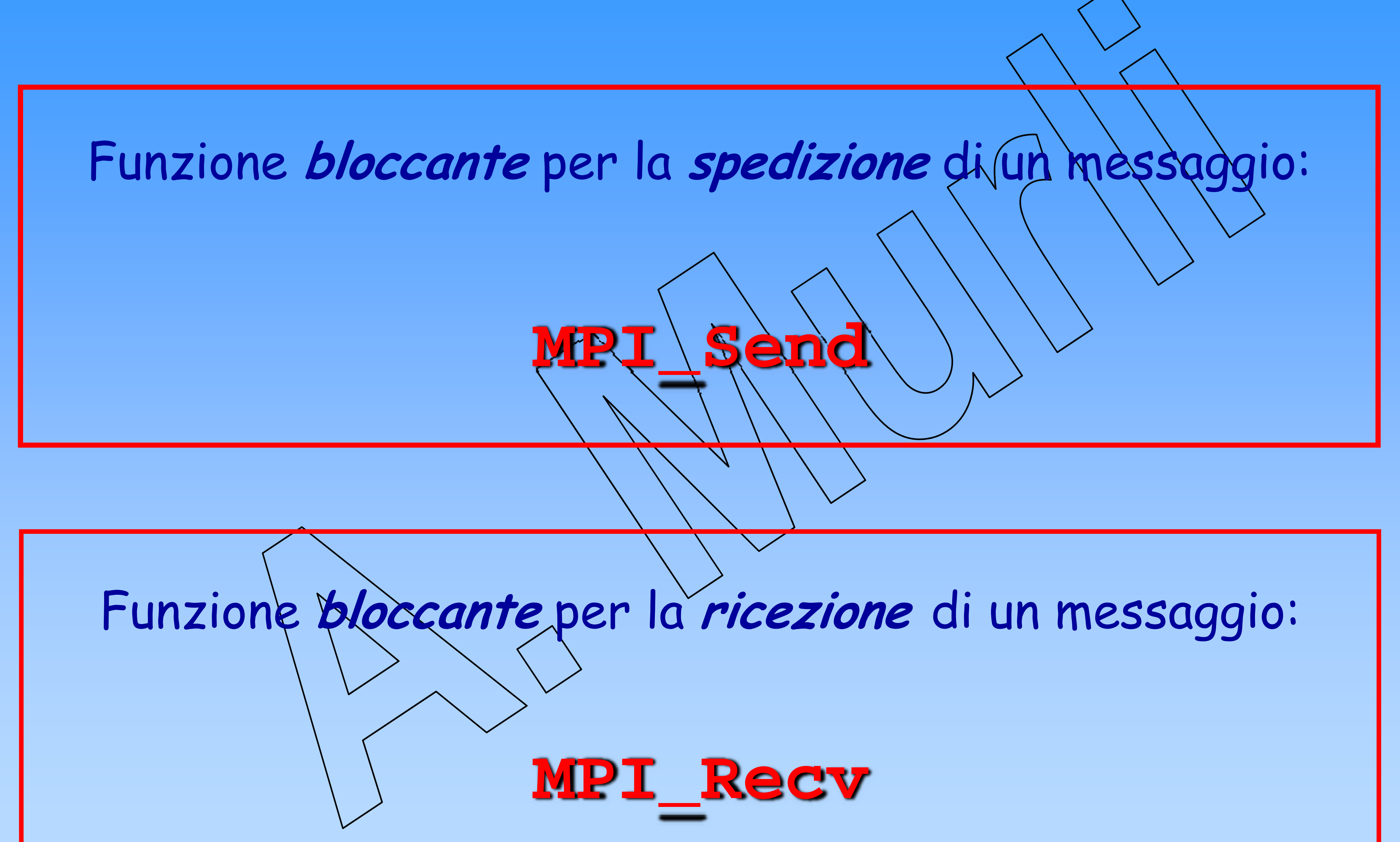
La spedizione o la ricezione
di un messaggio
da parte di un processore può essere
bloccante o **non bloccante**

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

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

A. Murli

Comunicazioni bloccanti in MPI



A. Murli

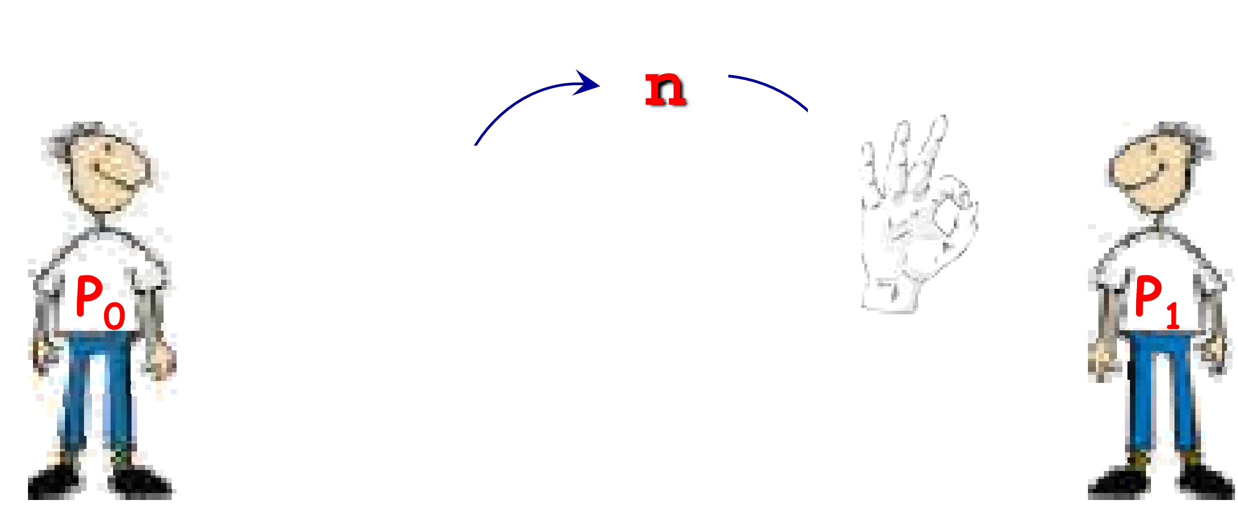
Un semplice programma con 2 processori:

```
#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 processore chiamante P_0 spedisce il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, al processore P_1 ; i due processori 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 processore che esegue questa routine spedisce i primi **count** elementi di **buffer**, di tipo **datatype**, al processore con identificativo **dest**.

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

A. Murli

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 processore destinatario
`comm`	identificativo del communicator
`tag`	identificativo del messaggio

A. Murli

Un semplice programma con 2 processori:

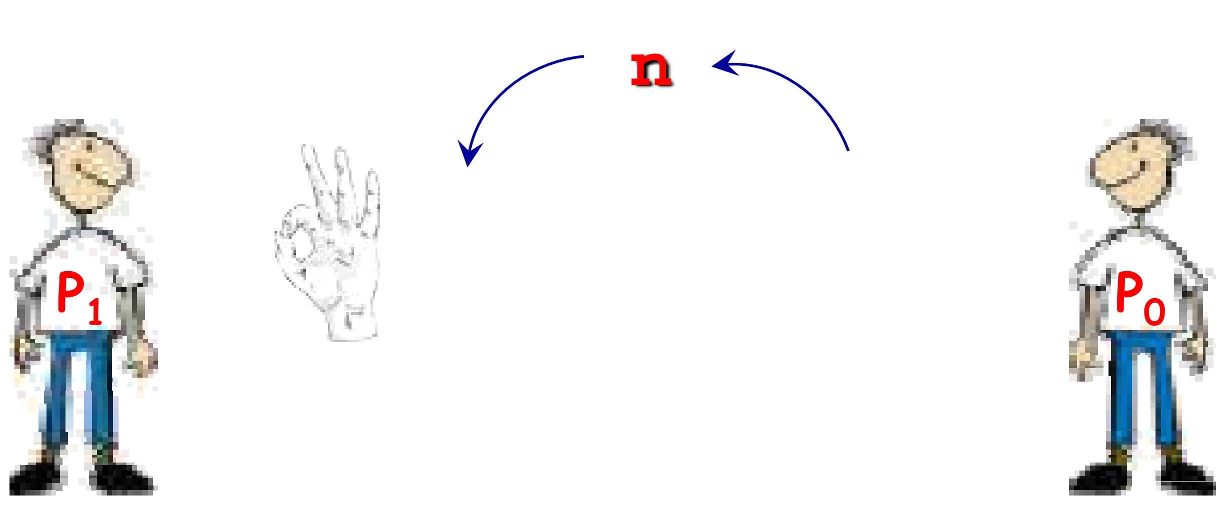
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
     int n, tag, num;
     MPI_Status info;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&menum);
     if(menum==0)
     { scanf("%d",&n);
       tag=10;
       MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     }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 processore chiamante P_1 riceve il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, dal processore P_0 ; i due processori appartengono entrambi al communicator **MPI_COMM_WORLD**. Il parametro **tag** individua univocamente tale spedizione. Il parametro **info**, di tipo **MPI_Status**, contiene informazioni sulla ricezione del messaggio.

Murli



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 processore che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processore 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.

A. Murli

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 processore da cui ricevere

comm identificativo del communicator

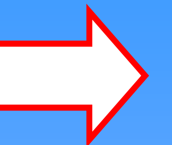
tag identificativo del messaggio

A. Murli

Un semplice programma con 2 processori:

```
#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 processore chiamante di conoscere il numero, num, di elementi ricevuti, di tipo MPI_INT, nella spedizione individuata da info.

num =1

num =1

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

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

A. Murli

Comunicazioni non bloccanti in MPI: modalità *Immediate*

A. Murli

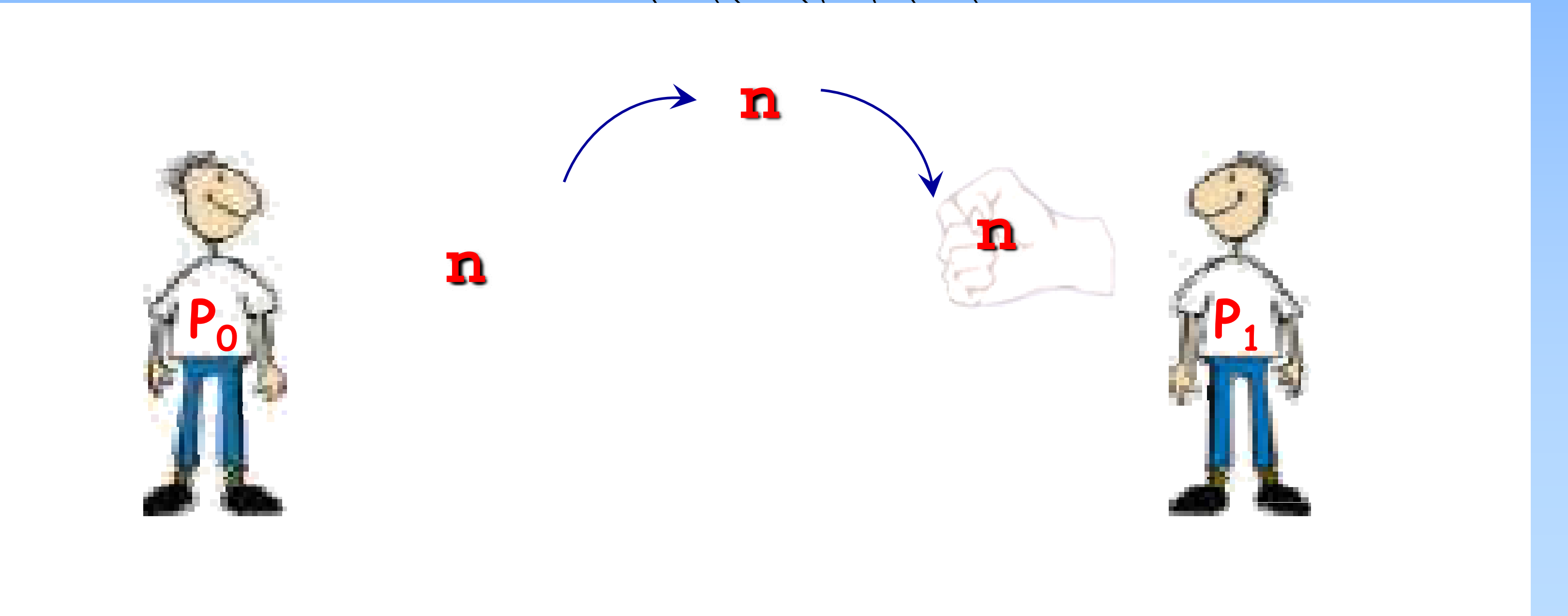
Un semplice programma con 2 processori:

```
#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 processore chiamante P_0 spedisce il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, al processore P_1 ; i due processori appartengono entrambi al communicator **MPI_COMM_WORLD**. Il parametro **tag** idividua univocamente tale spedizione. Il parametro **rqst** contiene le informazioni dell'intera spedizione. Il processore 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 processore che esegue questa routine spedisce i primi **count** elementi di **buffer**, del tipo **datatype**, al processore 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

A. Murli

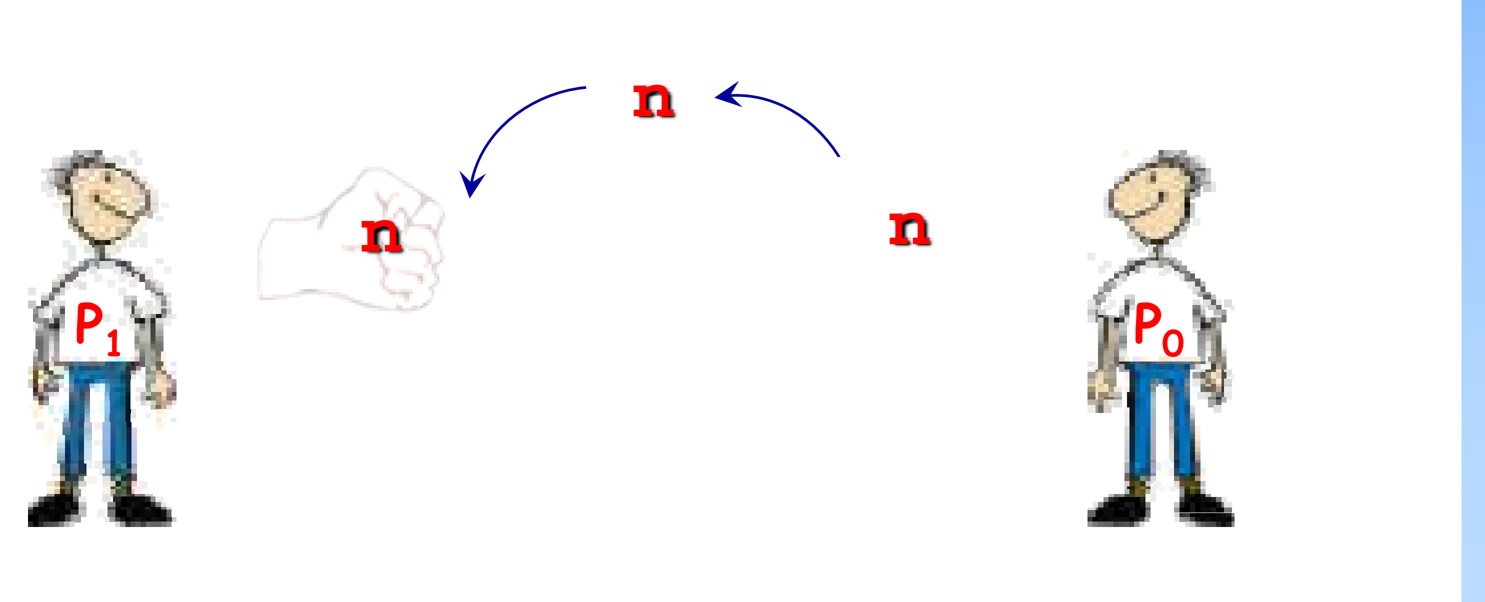
Un semplice programma con 2 processori:

```
#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 processore chiamante P_1 riceve il parametro **n**, di tipo **MPI_INT** e di dimensione **1**, dal processore P_0 ; i due processori appartengono entrambi al communicator **MPI_COMM_WORLD**. Il parametro **tag** individua univocamente tale ricezione. Il parametro **rqst** contiene le informazioni dell'intera spedizione. Il processore 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 processore che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processore 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.

A. Murli

In particolare: request

A. Murli

Osservazione

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

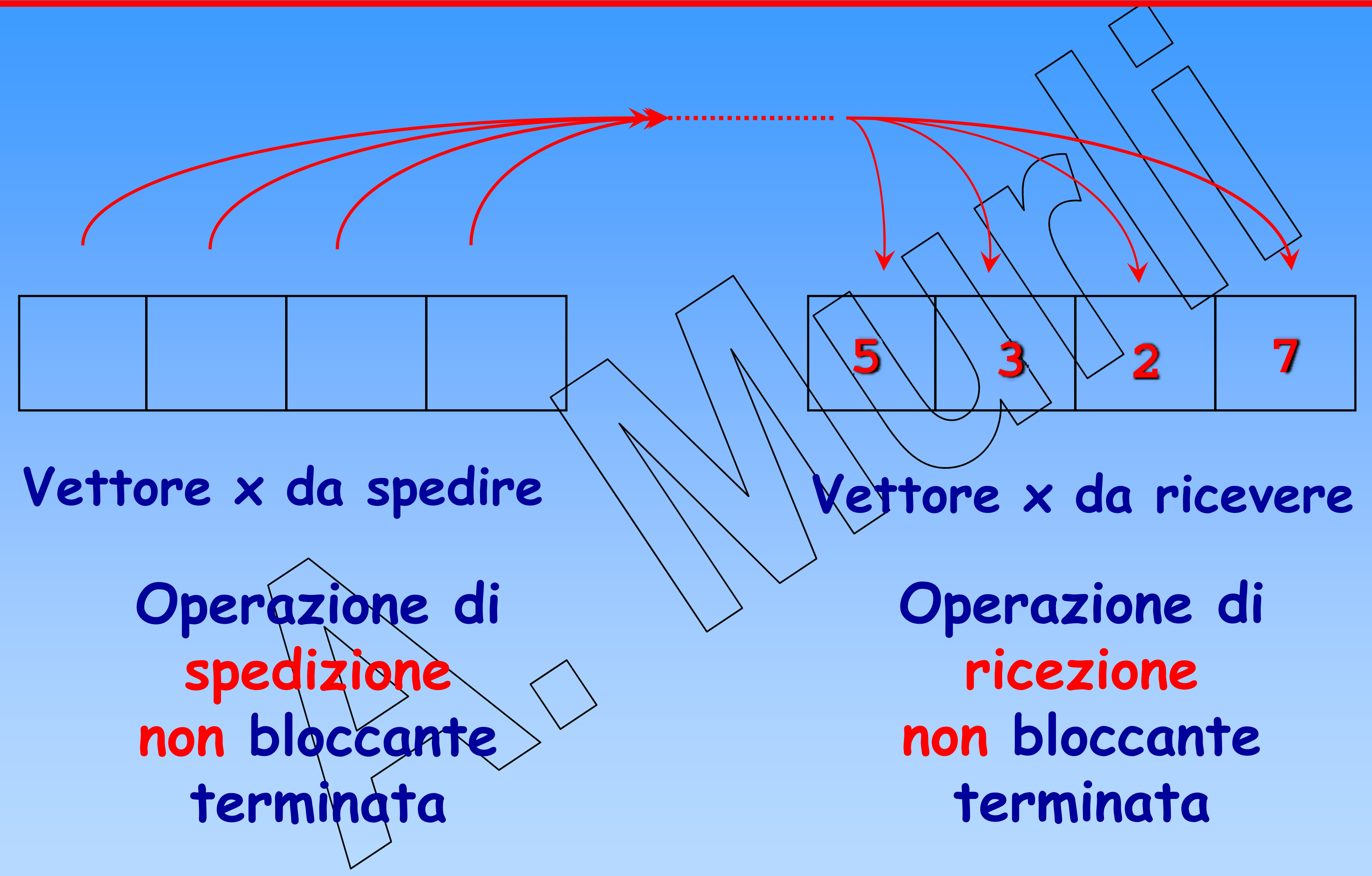
status
contiene informazioni sulla ***ricezione***
del messaggio.

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

A. Murli

Osservazione: termine di un'operazione non bloccante

Osservazione: termine di un'operazione non bloccante



Operazione di **spedizione** **non** bloccante terminata

Operazione di **ricezione** **non** bloccante terminata

A. Murli

Domanda

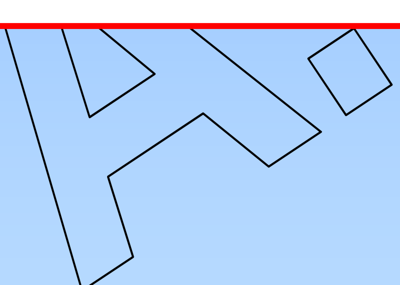
Utilizzando una comunicazione non bloccante

come si fa a sapere

se l'operazione di spedizione o di ricezione

è stata terminata

?



Risposta

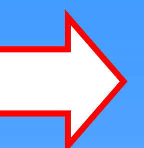
MPI mette a disposizione delle funzioni per ***controllare tutta la fase*** **di trasmissione di un messaggio mediante comunicazione non bloccante.**

Murli

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);
    }
    …
}
```

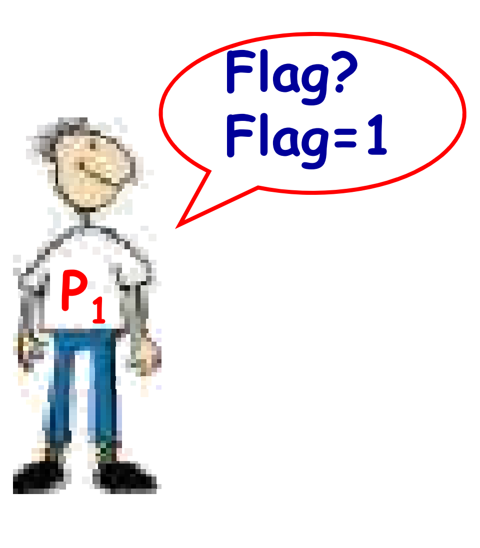
Nel programma … :

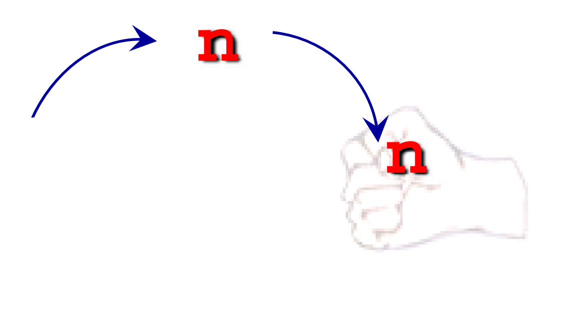


MPI_Test(&rqst,&flag,&info);

- Il processore 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.

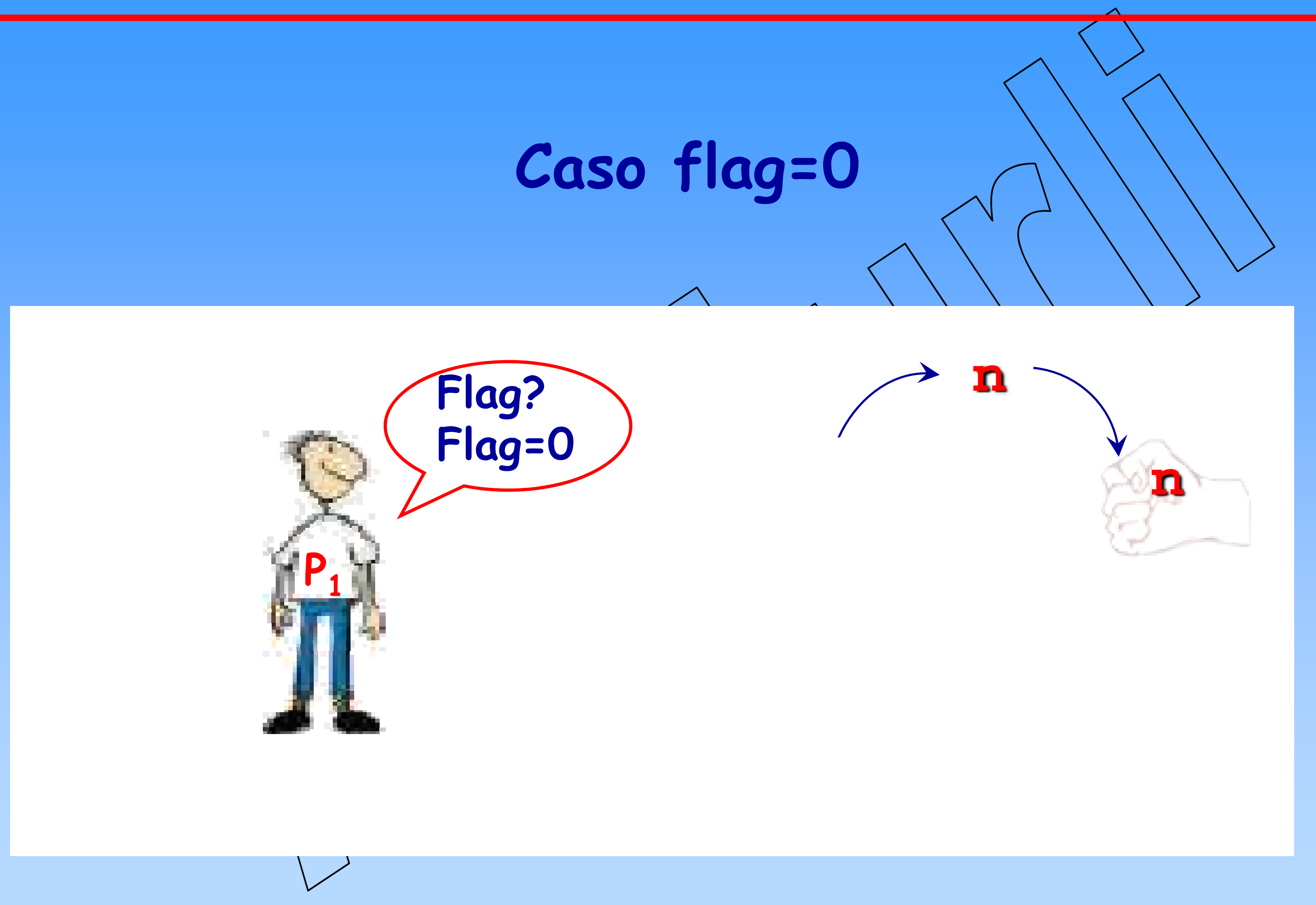
Caso flag=1





Murli

Nel programma … :



Controllo sullo stato di un'operazione non bloccante…

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

- Il processore 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;

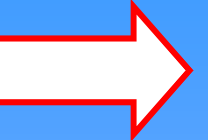
  flag = 0, l'operazione identificata da **request** NON è terminata;

A. Murli

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 processore chiamante P_1 procede nelle istruzioni solo quando la ricezione di n è stata completata.

Procedi

n

P_1

n

In generale… :

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

- Il processore 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.

A. Murli

Vantaggi delle operazioni non bloccanti

Le comunicazioni di tipo non bloccante hanno DUE vantaggi:

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

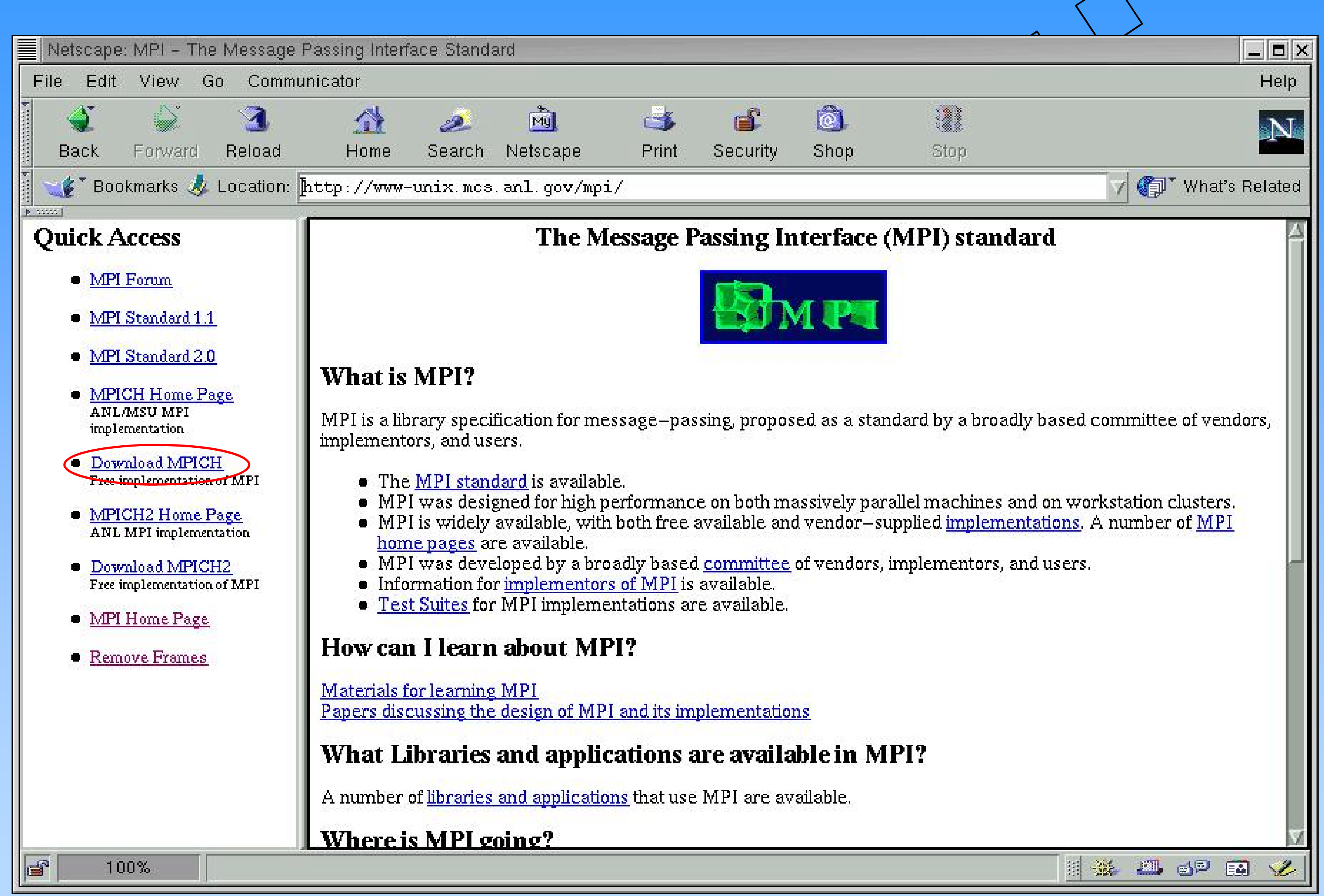
```
…
if(menum==0)
 { scanf("%d",&n);
   tag=20;
   MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
 }
 /* P0 può procedere nelle operazioni senza dover
    attendere il risultato della spedizione di n
    al processore P1 */
 if(menum!=0)
 {
  …
 }
 …
```

Vantaggi delle operazioni non bloccanti

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

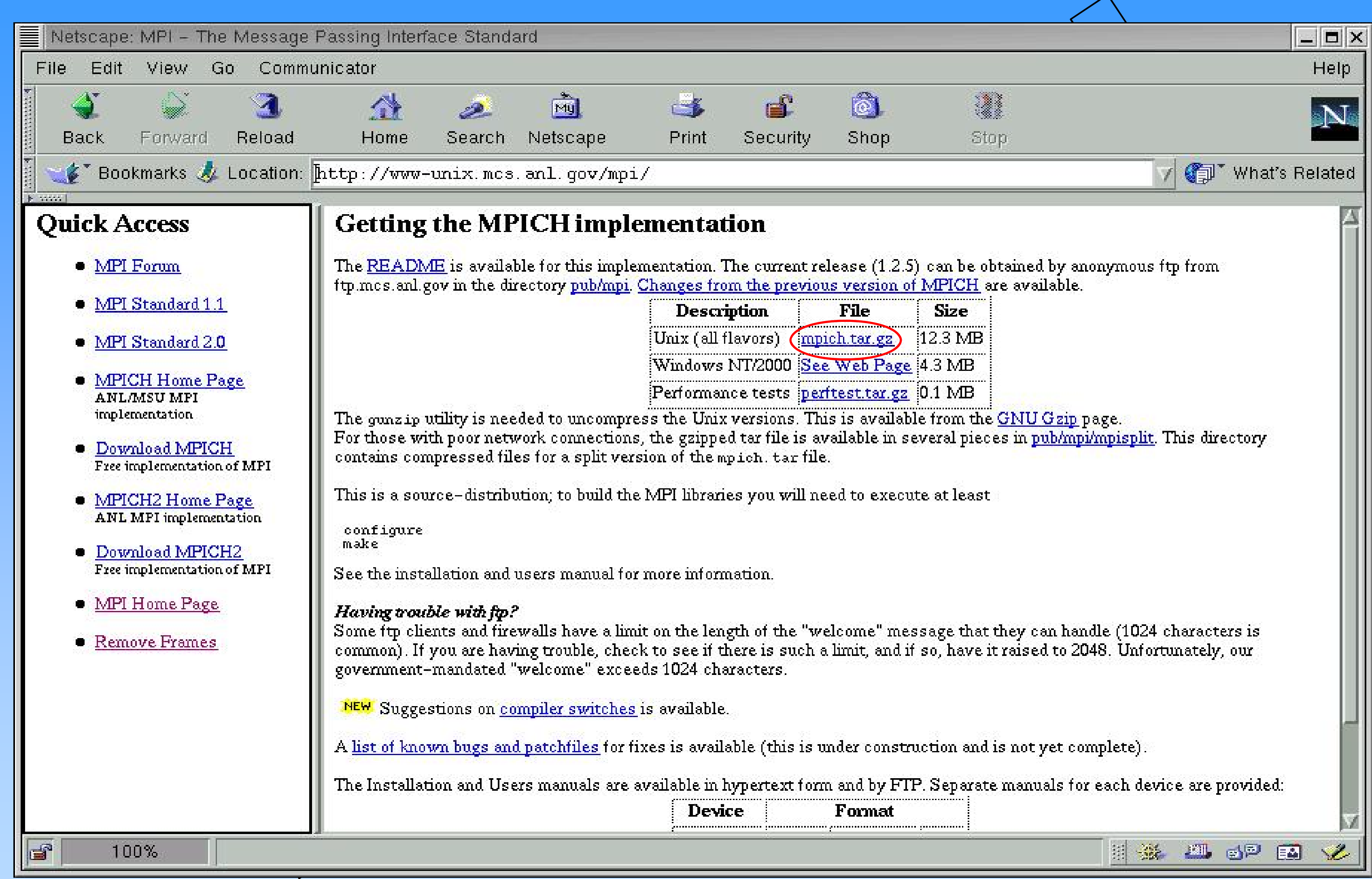
```
…
if(menum==0)
{ …
 /* P0 spedisce due elementi a P1 */
  MPI_Isend(&a,1,MPI_INT,1,0,MPI_COMM_WPRLD,&rqst1);
  MPI_Isend(&b,1,MPI_INT,1,0,MPI_COMM_WORLD,&rqst2);
} elseif(menum==1) {
/* P1 riceve due elementi da P0
  secondo l’ordine di spedizione*/
  MPI_Irecv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&rqst1);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

Dove recuperare MPI



www-unix.mcs.anl.gov/mpi/

Dove recuperare MPI



www-unix.mcs.anl.gov/mpi/

Esplorazione directory MPI