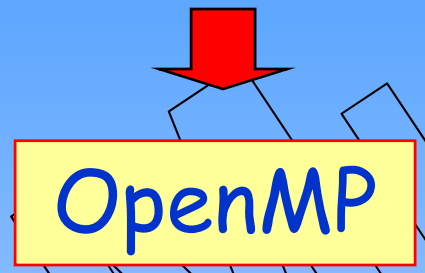
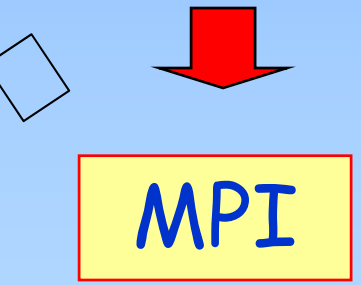
Alcuni strumenti per lo sviluppo di software su architetture MIMD

Calcolatori MIMD

- Architetture SM (Shared Memory)



- Architetture DM (Distributed Memory)



Message Passing Interface
MPI

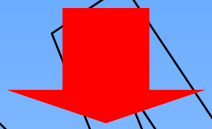
Le funzioni di MPI

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*

La comunicazione di un messaggio

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



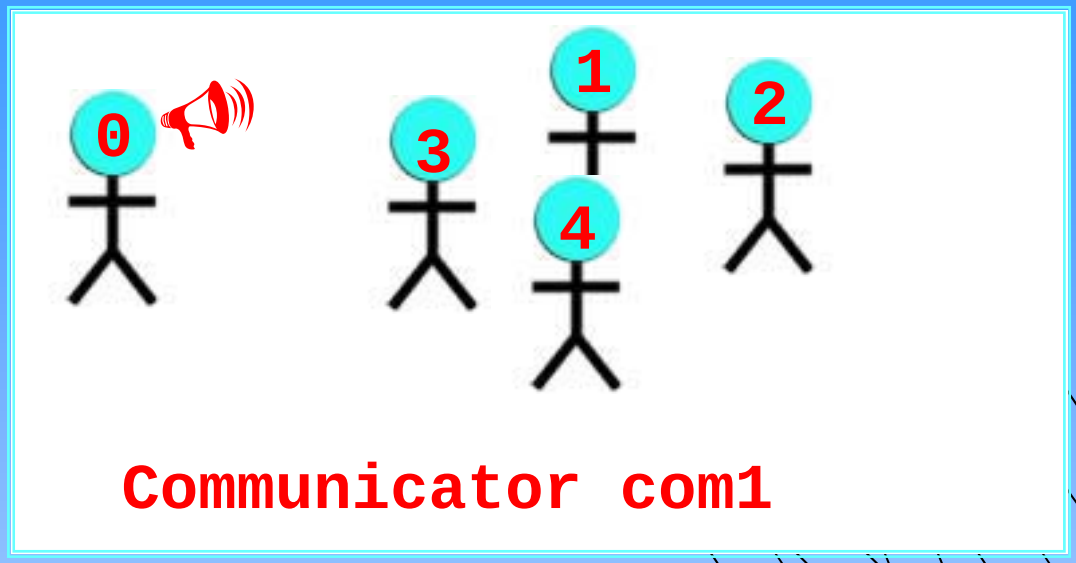
Per comunicazioni che coinvolgono solo due processori → Si considerano funzioni MPI per **comunicazioni uno a uno**

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

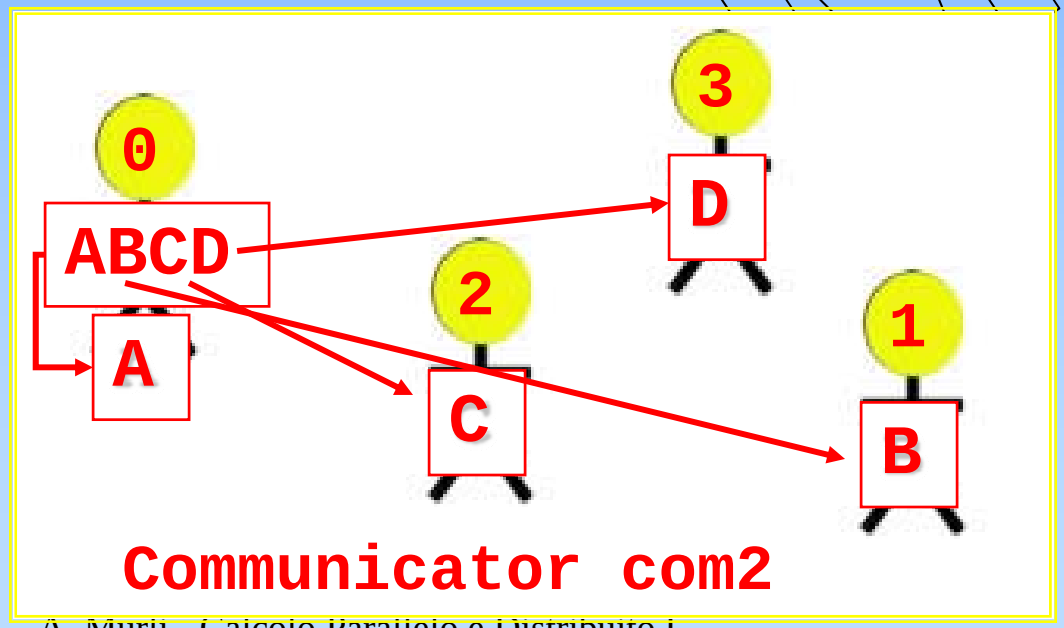
A. Murli

Comunicazione collettive di un messaggio

Comunicazioni collettive :



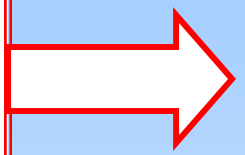
Nel connumicator com1 P_0 comunica con tutti gli altri processori



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

Spedizione collettiva, uno a molti:

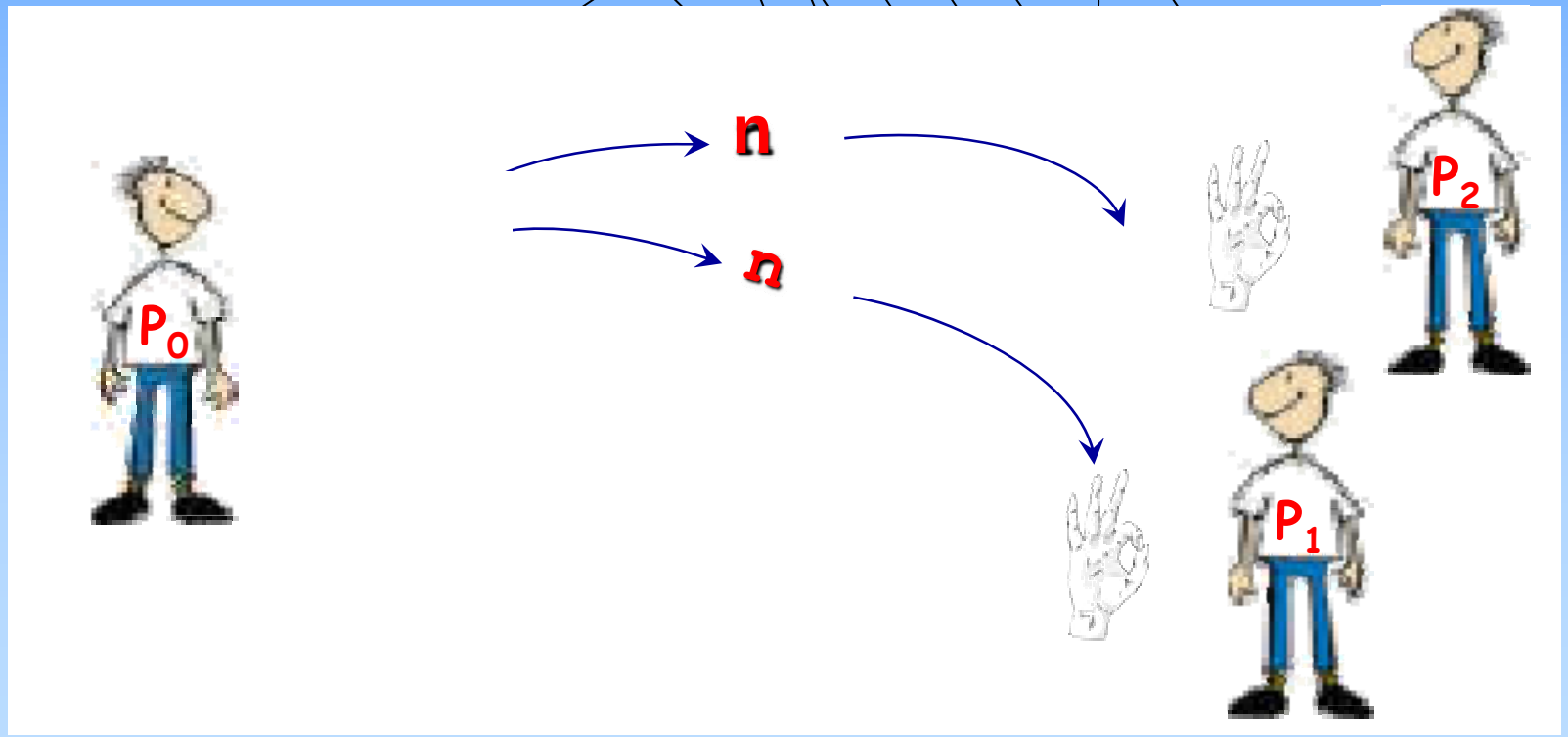
```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv[])
{   int menum, nproc;
    int n, tag, num;
    MPI_Status info;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    if(menum==0)
    {
      scanf(“%d”,&n);
    }
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);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



Nel programma … :

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

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



In generale … :

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

- Il processore con identificativo **root** spedisce a tutti i processori del comunicator **comm** lo stesso dato memorizzato in ***buffer**.
- **Count, datatype, comm** devono essere uguali per ogni processore di **comm.**

In dettaglio…:

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

***buffer** indirizzo del dato da spedire

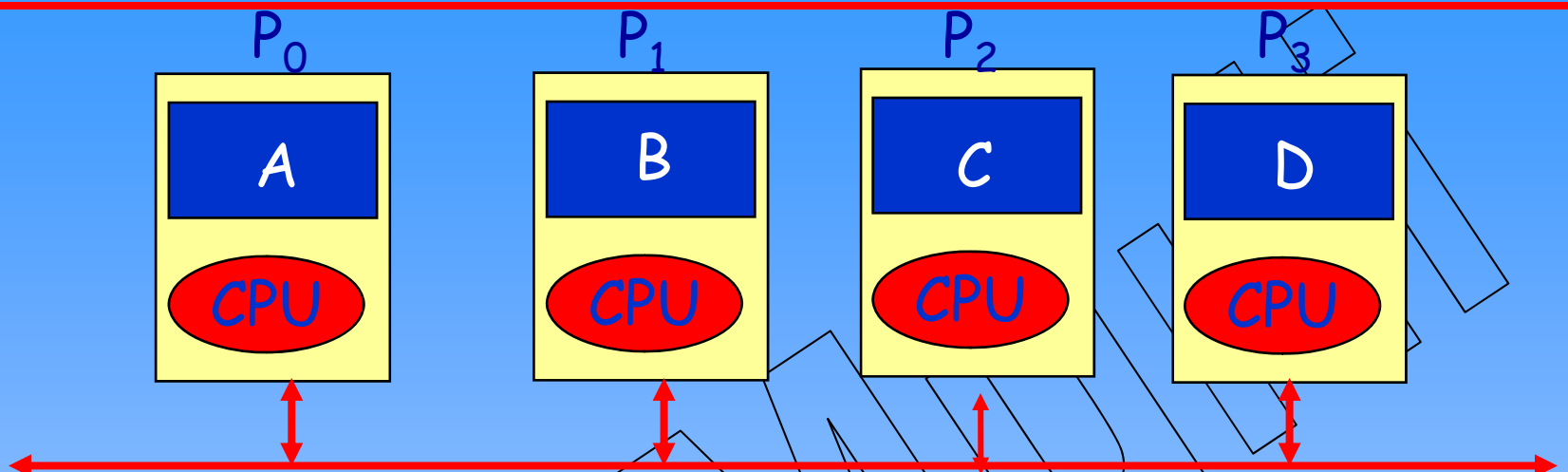
count numero dei dati da spedire

datatype tipo dei dati da spedire

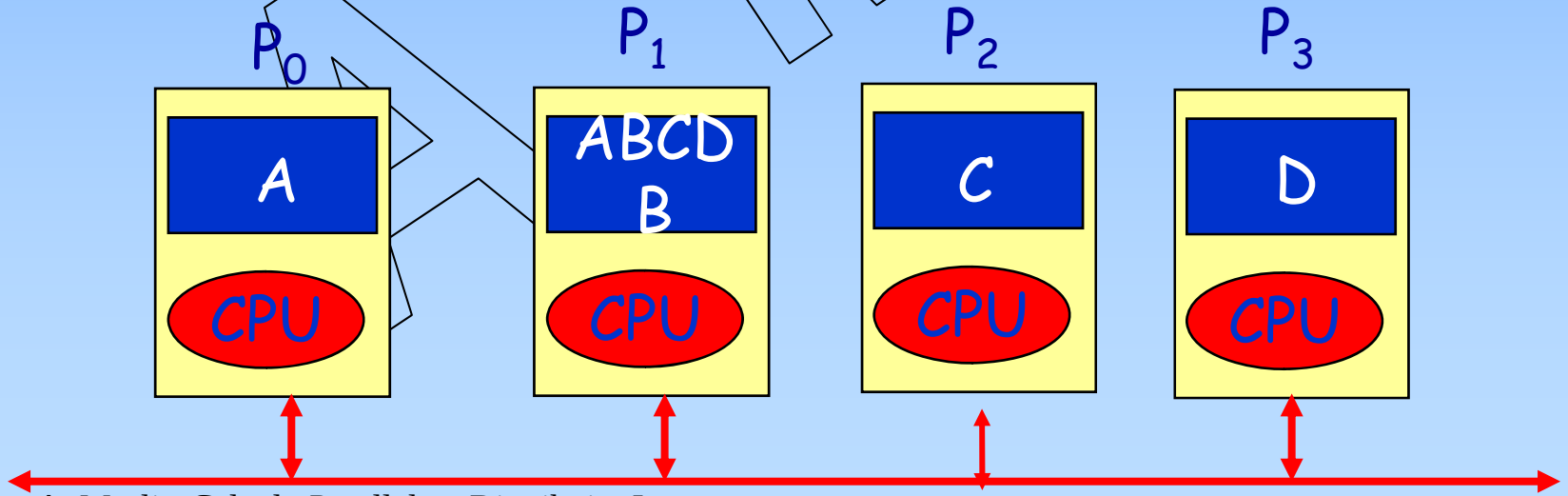
root identificativo del processore che spedisce a tutti

comm identificativo del communicator

Spedizione collettiva di tipo: *data gather*



Tutti i processori spediscono i propri dati ad un processore assegnato (es al processore 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 processore di **comm** spedisce il contenuto di ***send_buff** al processore con identificativo **root**
- Il processore con identificativo **root** *concatena* i dati ricevuti in **recv_buff**, memorizzando prima i dati ricevuti dal processore 0, poi i dati ricevuti dal processore 1, poi quelli ricevuti dal processore 2, etc…

Il gather in MPI: in dettaglio…

```
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);
```

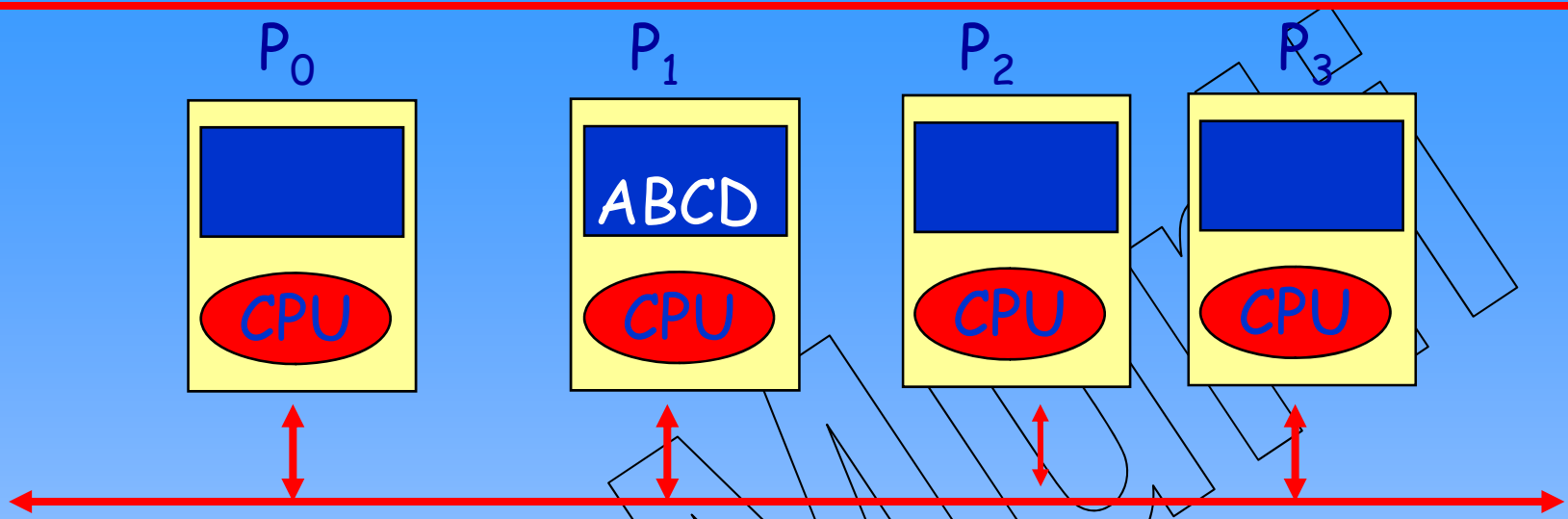
*send_buff	indirizzo del dato da spedire
send_count	numero dei dati da spedire
sendtype	tipo dei dati da spedire
*recv_buff	indirizzo del dato in cui root riceve
recv_count	numero dei dati che root riceve
recv_type	tipo dei dati che root riceve
root	identificativo del processore che riceve da tutti
comm	identificativo del communicator

Il gather in MPI

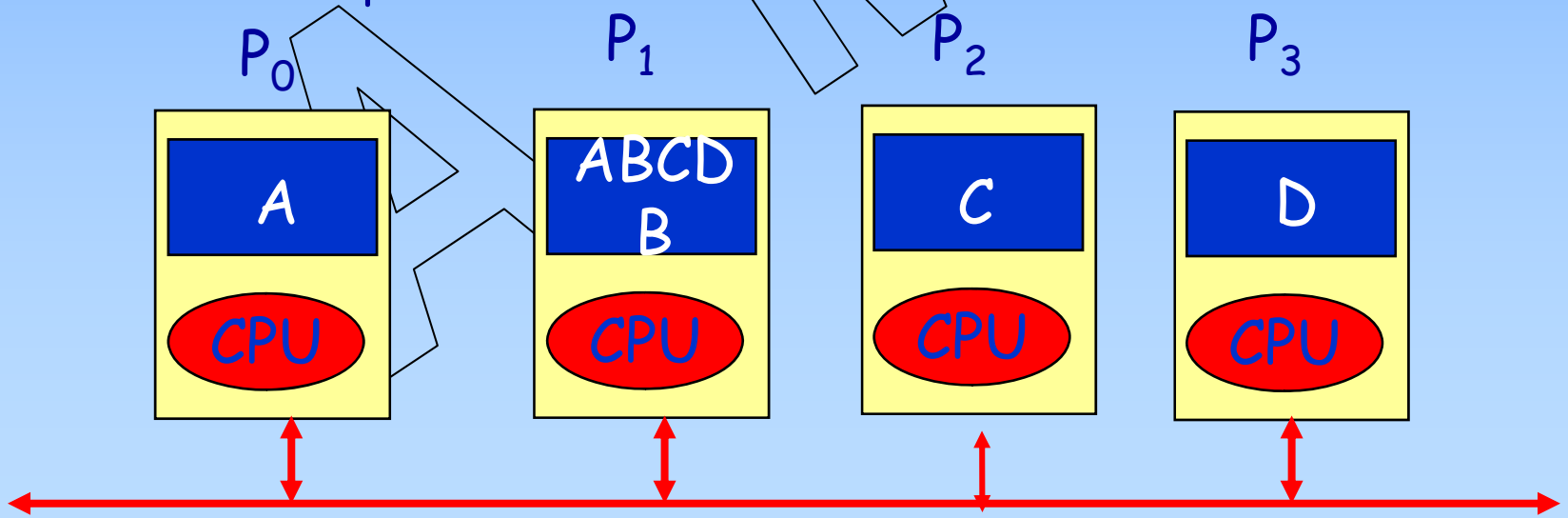
```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype datatype,
           void *recv_buff,int recv_count,
           MPI_Datatype recv_type,
           int root, MPI_Comm comm);
```

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

Spedizione collettiva di tipo: *data scatter*



Un solo processore distribuisce i propri dati agli altri processori, se stesso compreso.



Lo *scatter* in MPI

```
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 processore 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 processore con identificativo 0, il secondo al processore con identificativo 1, il terzo al processore con identificativo 2, etc.

Lo *scatter* in MPI: in dettaglio

```
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 processore che spedisce a tutti

comm identificativo del communicator

Le operazioni collettive.

Caratteristiche delle operazioni collettive

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

Inoltre…

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

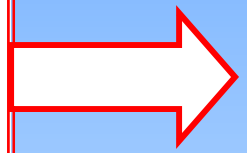
Scopo delle operazioni collettive

Le operazioni collettive permettono:

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

Sincronizzazione dei processori … :

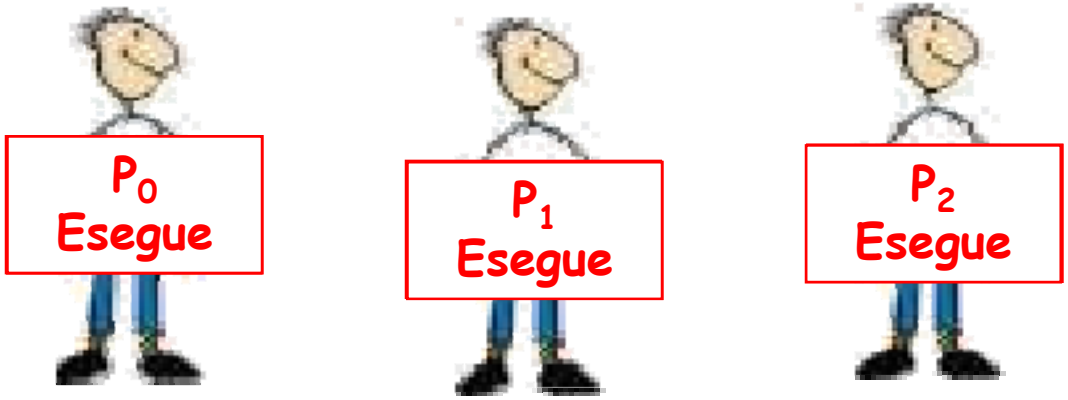
```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv[])
{   …
    MPI_Init(&argc,&argv);
    …
    MPI_Barrier(MPI_COMM_WORLD);
    …
  MPI_Finalize();
}
```



Nel programma … :

MPI_Barrier(MPI_COMM_WORLD);

- Ogni processore 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 processori del communicator **comm**.

- Ogni processore si *ferma* fin quando tutti i processori di **comm** non eseguono **MPI_Barrier**.

Operazioni di *riduzione*

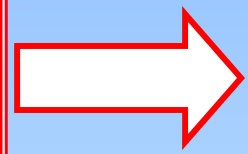
MPI possiede una classe
di operazioni collettive chiamate
operazioni di riduzione.

In ciascuna operazione di riduzione
tutti i processori di un
communicator contribuiscono al
risultato di un'operazione.

Un semplice programma :

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv[])
{   int menum, nproc;
    int sum, sumtot;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    sum=nproc;
    sumtot=0;
    sum+=menum;
    MPI_Reduce(&sum,&sumtot,1,MPI_INT,MPI_SUM,0,
               MPI_COMM_WORLD);
    printf(“Sono %d sum=%d sumtot=%d\n”,menum,sum,sumtot);

    MPI_Finalize();
    return 0;
}
```



Nel programma … :

```
MPI_Reduce(&sum,&sumtot,1,MPI_INT,MPI_SUM,
           0, MPI_COMM_WORLD);
```

- Il processore P_0 otterrà la somma dei elementi **sum**, di tipo **MPI_INT** e di dimensione **1**, distribuiti tra tutti i processori del communicator **MPI_COMM_WORLD**. Il risultato lo pone nella propria variabile **sumtot**.

Operazioni di Reduce … :

Le operazioni globali di Reduce sono implementate in maniera efficiente in quanto:

1) Ottimizzano le comunicazioni tra i processori del communicator coinvolto. Le comunicazioni vengono eseguite seguendo uno schema ad albero.

2) Sfruttano la proprietà associativa e/o la proprietà commutativa.

In Generale :

```
MPI_Reduce(void *operand, void *result,
           int count, MPI_Datatype datatype,
           MPI_Op op, int root,
           MPI_Comm comm);
```

- Tutti i processori di **comm** combinano i propri dati memorizzati in ***operand** utilizzando l'operazione **op**.
- Il risultato viene memorizzato in ***result** di proprietà del processore con identificativo **root**
- **Count, datatype, comm** devono essere uguali per ogni processore di **comm.**

In dettaglio… :

```
MPI_Reduce(void *operand, void *result,
           int count, MPI_Datatype datatype,
           MPI_Op op, int root,
           MPI_Comm comm);
```

***operand** indirizzo dei dati su cui effettuare l'operazione.

***result** indirizzo del dato contenente il risultato.

count numero dei dati su cui effettuare l'operazione.

datatype tipo degli elementi da spedire.

op operazione effettuata.

root identificativo del processore che conterrà il risultato

comm identificativo del communicator

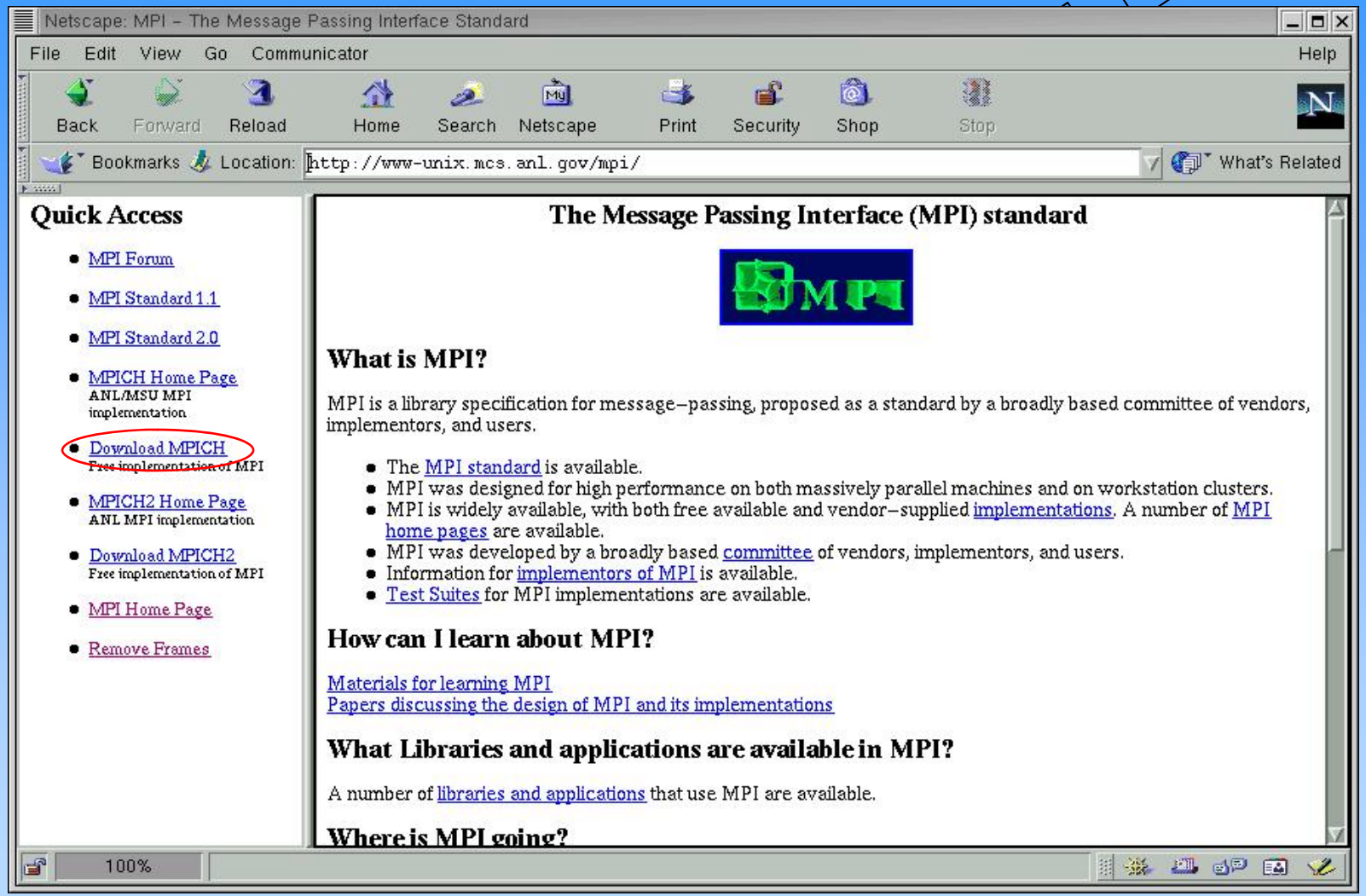
Operazione collettiva: *Reduce*

L'argomento **op**, che descrive l'operazione da eseguire sugli operandi **operand**, distribuiti fra i processori può essere scelto fra i seguenti valori predefiniti:

MPI_MAX	Massimo di un vettore distribuito
MPI_MIN	Minimo di un vettore distribuito
MPI_SUM	Somma componente per componente fra vettori distribuiti
MPI_PROD	Prodotto componente per componente fra vettori distribuiti

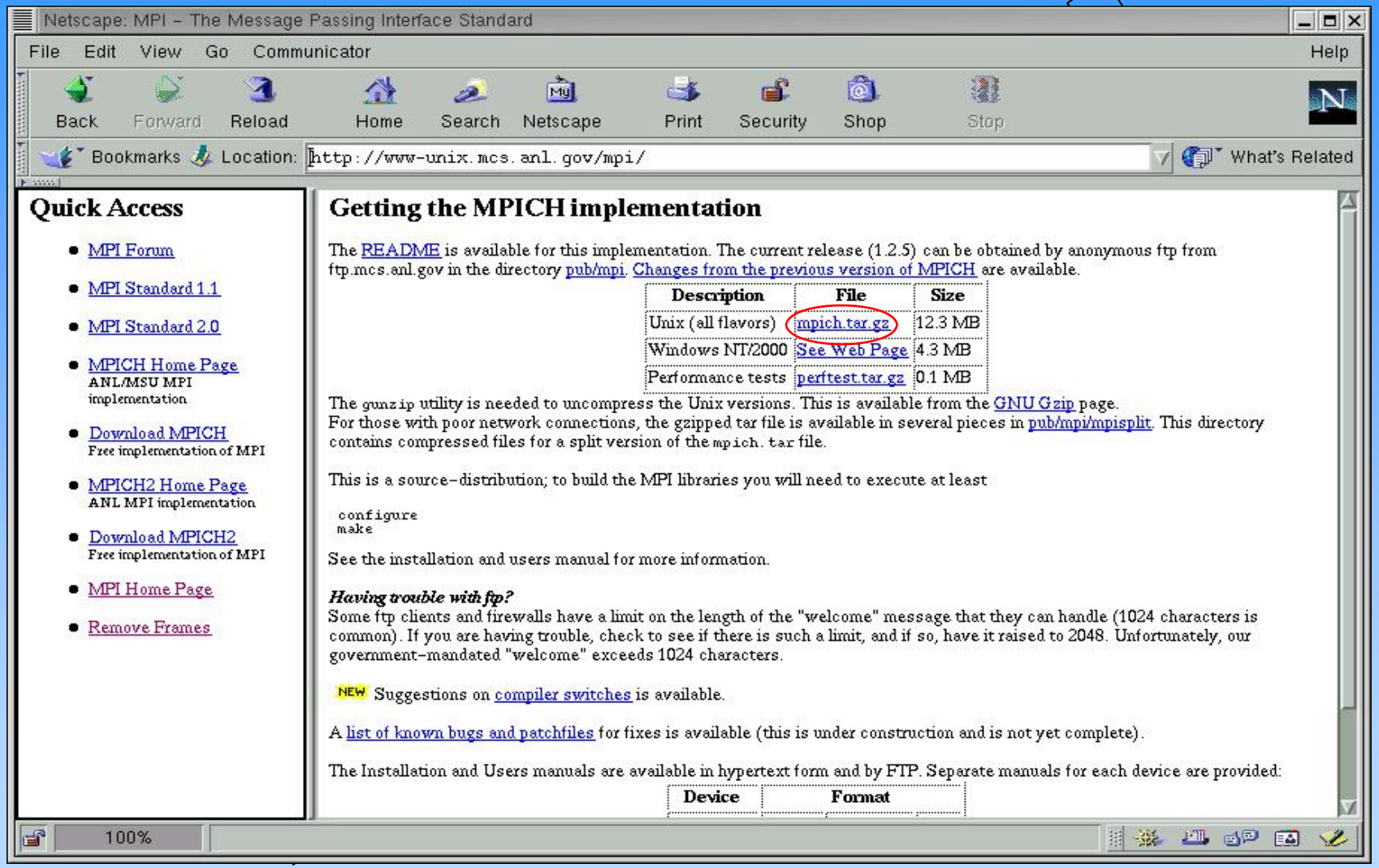
……

Dove recuperare MPI



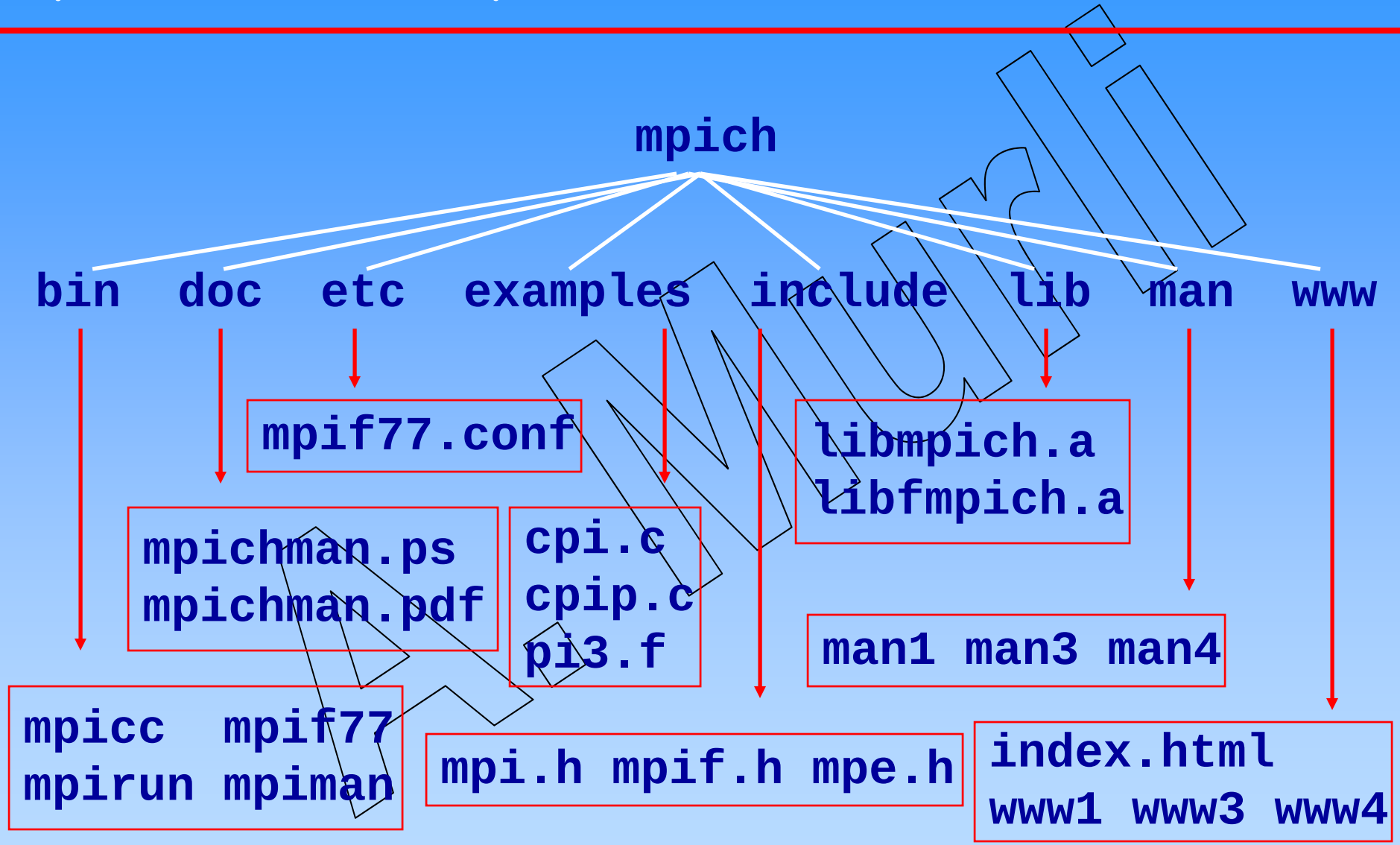
www-unix.mcs.anl.gov/mpi/

Dove recuperare MPI



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

Esplorazione directory MPI



Alcuni programmi per l'utilizzo di MPI