

Alcuni strumenti software per  
lo sviluppo di software  
su architetture MIMD

## Calcolatori MIMD

• Architetture SM (**Shared Memory**)

OpenMP

• Architetture DM (**Distributed Memory**)

MPI

A. Murli

2

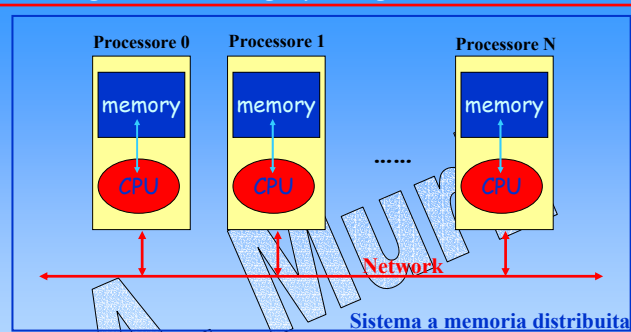
MPI :

**Message Passing Interface**  
**MPI**

A. Murli

3

## 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.

A. Murli

5

## 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**

A. Murli

6

## 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;
}
```

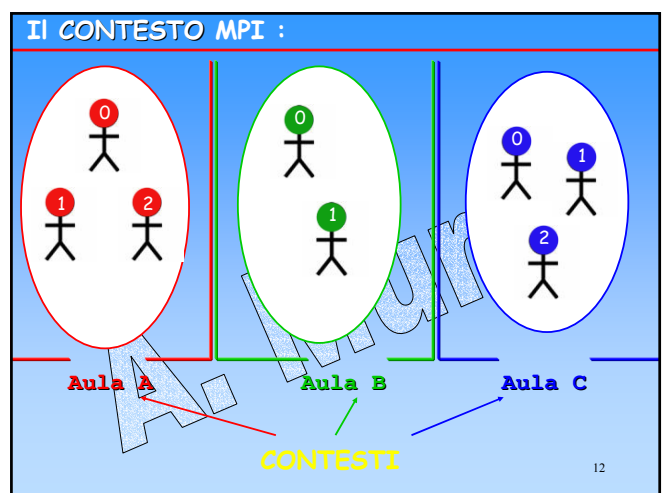
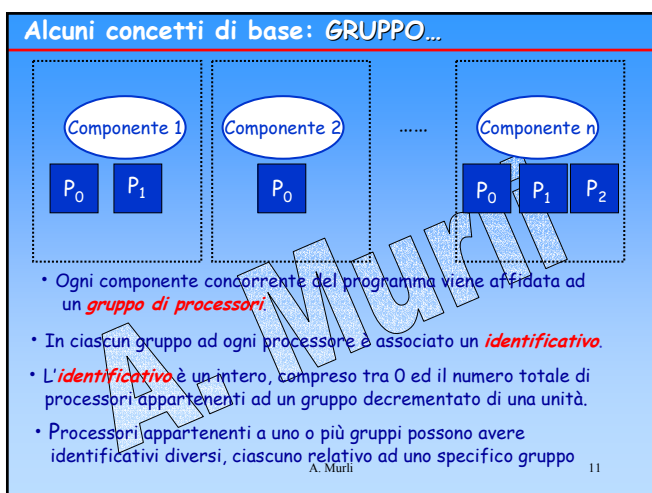
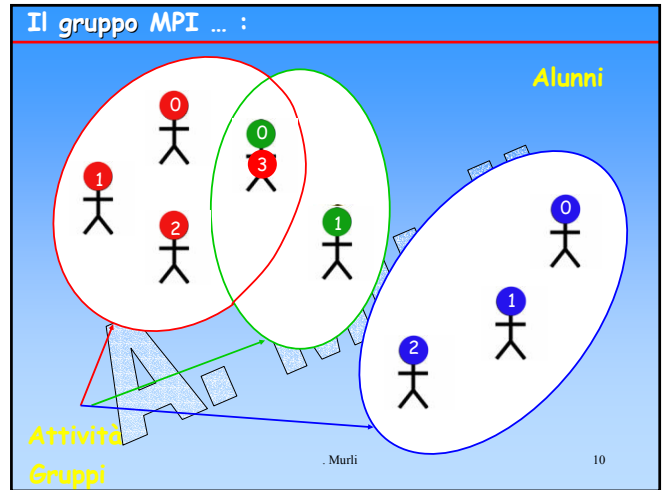
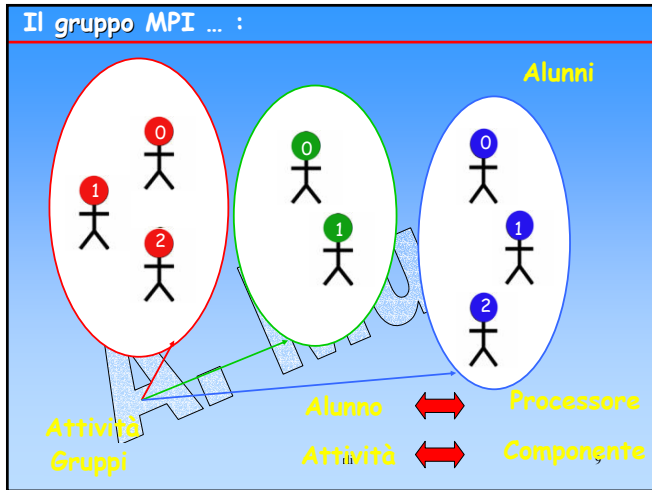
```
main()
{
    ...
    sum=0;
    for i=10 to 14do
        sum=sum+a[i];
    endfor
    ...
    return 0;
}
```

## In MPI i processori sono raggruppati in ....:

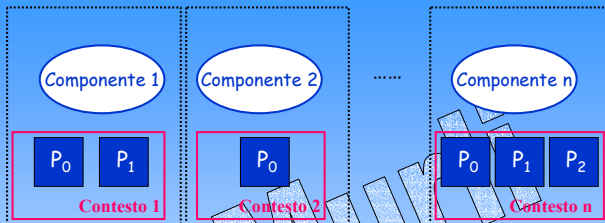


A. Murli

8



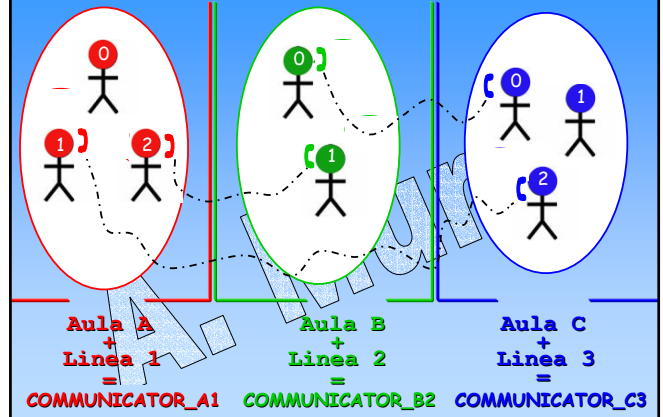
## Alcuni concetti di base: CONTESTO



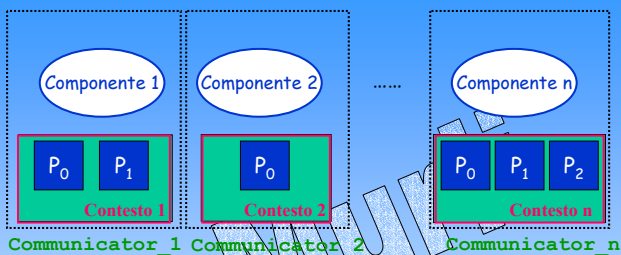
- A ciascun gruppo di processori viene attribuito un identificativo detto **contesto**.
- Il **contesto** definisce l'ambito in cui avvengono le comunicazioni tra processori di uno stesso gruppo.
- Se la spedizione di un messaggio avviene in un **contesto**, la ricezione deve avvenire nello **stesso**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

13

## Il Communicator MPI :



## Alcuni concetti di base: COMMUNICATOR...

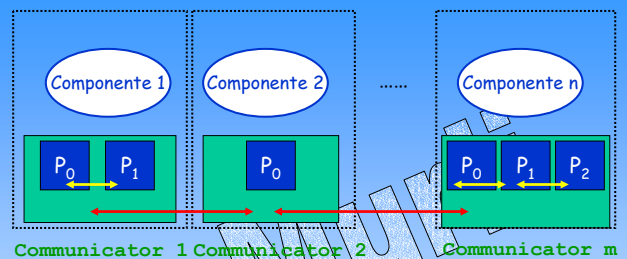


- Ad un gruppo di processori appartenenti ad uno stesso contesto viene assegnato un ulteriore identificativo: il **communicator**.
- Il **communicator** racchiude tutte le caratteristiche dell'ambiente di comunicazione: topologia, quali contesti coinvolge, ecc...

A. Murli

15

## ...Alcuni concetti di base: COMMUNICATOR...

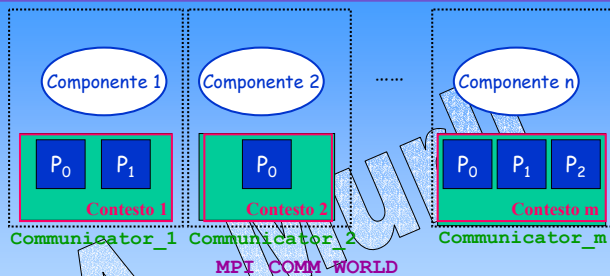


- Esistono due tipi principali di **communicator**.
- L'**intra-communicator** in cui le comunicazioni avvengono all'interno di un gruppo di processori.
- L'**inter-communicator** in cui le comunicazioni avvengono tra gruppi di processori.

A. Murli

16

## ...Alcuni concetti di base: COMMUNICATOR



• Tutti i processori fanno parte per default di un unico communicator detto **MPI\_COMM\_WORLD**.

17

## 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**

A. Murli

18

## MPI :

Le funzioni dell'ambiente.

A. Murli

19

## 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**.

20

Nel programma ... :

 `#include "mpi.h"`

`mpi.h` : Header File

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


A. Murli

21

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`.<sup>22</sup>

Nel programma ... :

 `MPI_Init(&argc, &argv);`

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

A. Murli

23

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 (comunicator).

A. Murli

24

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

26

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**.

A. Murli



Nel programma ... :

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &menum);**

- Questa routine permette al processore chiamante, appartenente al communicator **MPI\_COMM\_WORLD**, di memorizzare il proprio identificativo nella variabile **menum**.

A. Murli

28

### 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

29

### 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**.

A. Murli

30

### Nel programma ... :

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);**

- 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

31

### 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

32



MPI :

## Comunicazione di un messaggio.

A. Murli

33

Il Communicator MPI :



A. Murli

34

## 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)

Indirizzo in memoria del dato

Tipo del dato

Dimensione del dato

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

A. Murli

35

## Esempio 1: linguaggio C

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

### MPI datatype

- MPI\_INT
- MPI\_FLOAT
- MPI\_DOUBLE
- MPI\_CHAR
- ...

### C datatype

- int
- float
- double
- char
- ...

A. Murli

36

## Esempio 2: linguaggio Fortran

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

### MPI datatype

- MPI\_INT
- MPI\_FLOAT
- MPI\_DOUBLE
- MPI\_CHAR
- MPI\_LOGICAL

### Fortran datatype

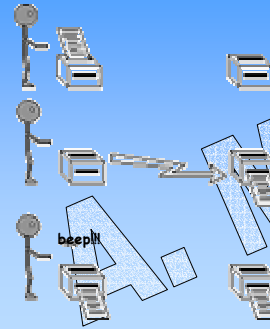
- integer
- real
- double precision
- character
- logical
- ...

A. Murli

37

## Tipi di comunicazioni ( Esempio 1 ) :

### Trasmissione di un fax



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

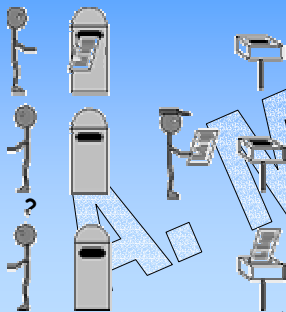


L'operazione di ricezione del messaggio **è stata completata**.

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.



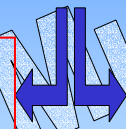
Il mittente **non sa** se l'operazione di ricezione del messaggio è stata completata.

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

40

## Comunicazioni bloccanti in MPI

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

**MPI\_Send**

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

**MPI\_Recv**

A. Murli

41

## 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;
}
```

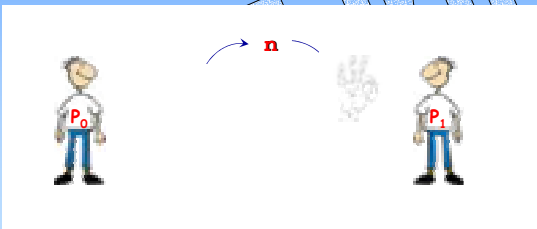
A. Murli

42

## 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.



43

## 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

44

## 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  
**tag** identificativo del messaggio  
**comm** identificativo del communicator

A. Murli

45

## 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;
}
```

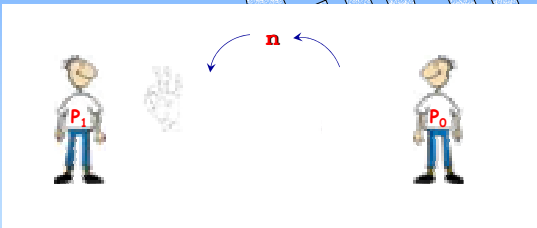
A. Murli

46

## 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.



47

## 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

48

## 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

**tag** identificativo del messaggio

**comm** identificativo del communicator

A. Murli

49

## 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;
}
```

A. Murli

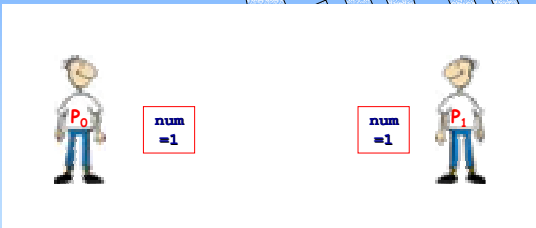
50

## 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**.



51

## 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 due campi:

- identificativo del processore da cui ricevere
- identificativo del messaggio

- 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

52

## 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**

A. Murli

53

## 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;
}
```

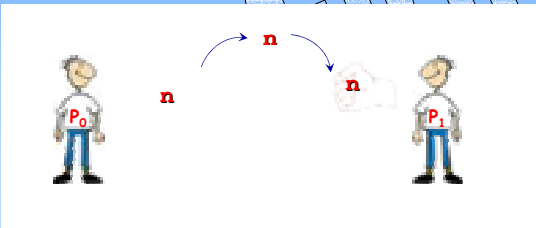
A. Murli

54

## 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** individua 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.



55

## 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

56

## 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;
}
```

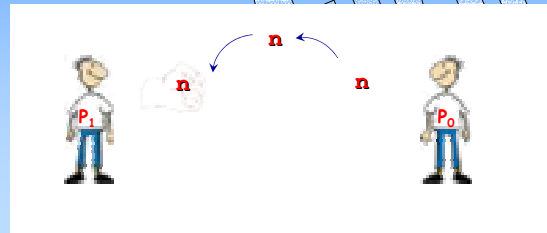
A. Murli

57

## 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.



58

## 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

59

## 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**.

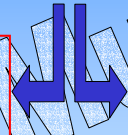
A. Murli

60

## 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.

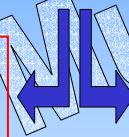
A. Murli

61

## 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.

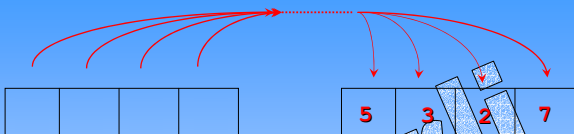


Nel caso della  
**ricezione** quando il  
messaggio è stato  
interamente  
ricevuto.

A. Murli

62

## 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**

A. Murli

63

## Domanda

Utilizzando una comunicazione non bloccante  
**come si fa a sapere**  
se l'operazione di spedizione o di ricezione  
**è stata terminata**

?

A. Murli

64



## Risposta

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

A. Murli

65

## 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 (menu==0)
    {
        scanf("%d", &n);
        tag=20;
        MPI_Isend(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &rqst);
    }
    if (menu!=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);
    }
    ...
}
```

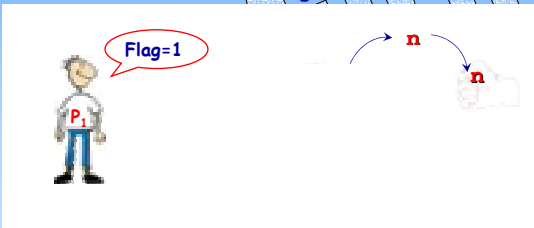
66

## 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  $flag=0$ .

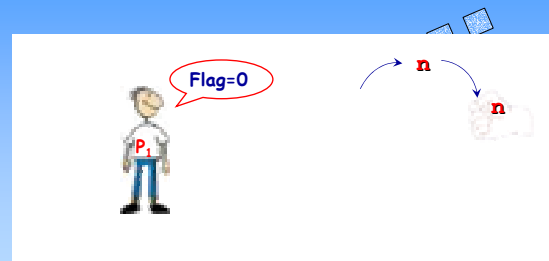
Caso  $flag=1$



67

## Nel programma ... :

Caso  $flag=0$



A. Murli

68

### 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

69

### Controllo sullo stato di un'operazione non bloccante...

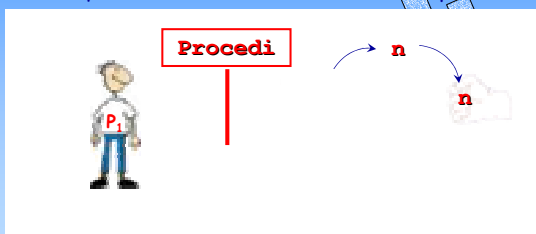
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int flag, nloc;
    ...
    nloc=1;
    if (menum==0)
    {
        scanf("%d", &n);
        tag=20;
        MPI_Isend(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &rqst);
    }
    if (menum!=0)
    {
        tag=20;
        MPI_Irecv(&n,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

70

### Nel programma ... :

**MPI\_Wait(&rqst, &info);**

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



A. Murli

71

### 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

72

## 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)
{
    ...
}
...
```

73

## 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_WORLD,&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);
}
...
```

A. Murli

74

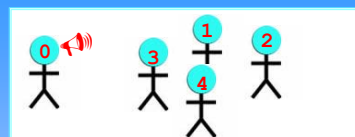
## MPI :

### Le operazioni collettive.

A. Murli

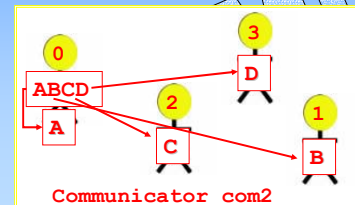
75

## Esempi di operazioni collettive :



Communicator com1

Nel communicator **com1** P<sub>0</sub> comunica con tutti gli altri processori



Communicator com2

P<sub>0</sub> distribuisce a tutti i processori di **com2** un elemento del proprio vettore

76

## 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**.

A. Murli

77

## 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.

A. Murli

78

## 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();
}
```

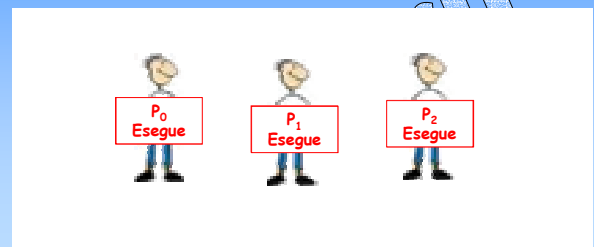
A. Murli

79

## 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.



A. Murli

80

## In generale ... :

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

- Questa routine fornisce un meccanismo sincronizzante per **tutti** i processori del comunicatore **comm**.
- Ogni processore si ferma fin quando tutti i processori di **comm** non eseguono **MPI\_Barrier**.

A. Murli

81

## La comunicazione collettiva 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

82

## 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;
    }
}
```

A. Murli

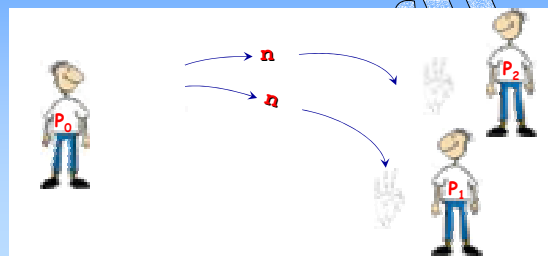
83

## Nel programma ... :

2/4

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

- Il processore **P<sub>0</sub>** spedisce **n**, di tipo **MPI\_INT** e di dimensione **1**, a tutti i processori dell'ambiente **MPI\_COMM\_WORLD**.



### In generale ... :

3/4

```
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 comunicatore **comm** lo stesso dato memorizzato in **\*buffer**.
- **Count**, **datatype**, **comm** devono essere uguali per ogni processore di **comm**.

A. Murli

85

### In dettaglio...:

4/4

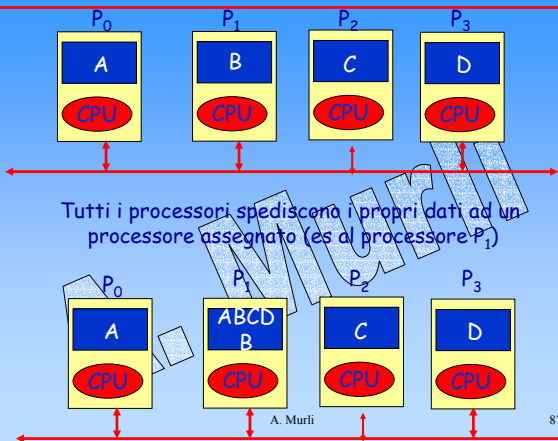
```
MPI_Bcast(void *buffer, int count,
MPI_Datatype datatype,
int root, MPI_Comm comm);
```

- \*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 comunicatore

A. Murli

86

### Operazione collettiva di tipo: data gather



A. Murli

87

### Il gather in MPI

1/3

```
MPI_Gather(void *send_buff, int send_count,
MPI_Datatype datatype,
void *recv_buff, int recv_count,
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...

A. Murli

88

## Il gather in MPI

2/3

```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype datatype,
           void *recv_buff, int recv_count,
           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.

A. Murli

89

## Il gather in MPI: in dettaglio...

3/3

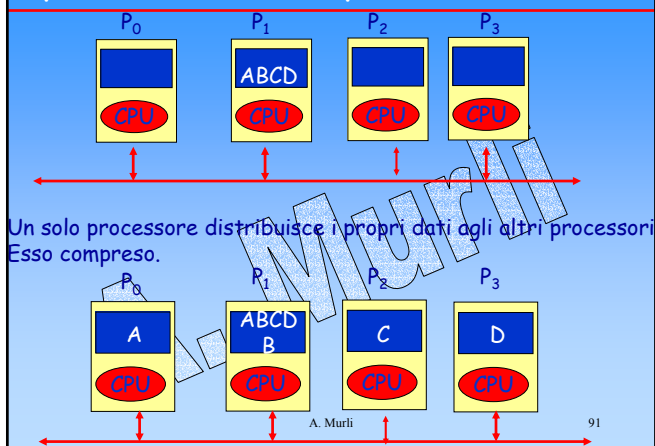
```
MPI_Gather(void *send_buff, int send_count,
           MPI_Datatype sendtype,
           void *recv_buff, int recv_count,
           MPI_Datatype recvtype,
           int root, MPI_Comm comm);
```

- \*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
- recvtype** tipo dei dati che **root** riceve
- root** identificativo del processore che riceve da tutti
- comm** identificativo del communicator

A. Murli

90

## Operazione collettiva di tipo: data scatter



91

## Lo scatter in MPI

1/2

```
MPI_Scatter(void *send_buff, int send_count,
            MPI_Datatype sendtype,
            void *recv_buff, int recv_count,
            MPI_Datatype recvtype,
            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.

A. Murli

92

## Lo scatter in MPI: in dettaglio

2/2

```
MPI_Scatter(void *send_buff, int send_count,
            MPI_Datatype datatype,
            void *recv_buff, int recv_count,
            MPI_Datatype recvtype,
            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 ricevere  
**recv\_count** numero dei dati da ricevere  
**recvtype** tipo dei dati da ricevere  
**root** identificativo del processore che spedisce a tutti  
**comm** identificativo del communicator

93

## 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.

A. Murli

94

## Un semplice programma :

1/6

```
#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, nproc);

    MPI_Finalize();
    return 0;
}
```

A. Murli

95

## Nel programma ... :

2/6

→ **MPI\_Reduce(&sum, &sumtot, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

- Il processore  $P_0$  effettua la somma dei **sum** elementi, 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**.

A. Murli

96



## Operazioni di Reduce ... :

3/6

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.

A. Murli

97

## In Generale :

4/6

```
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**.

A. Murli

98

## In dettaglio... :

5/6

```
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

A. Murli

99

## Operazione collettiva: *Reduce*

6/6

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

.....

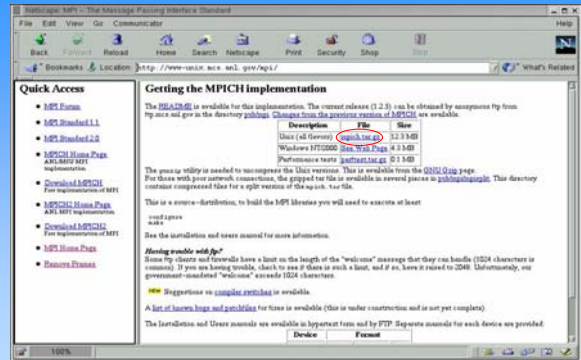
A. Murli

100



[www-unix.mcs.anl.gov/mpi/](http://www-unix.mcs.anl.gov/mpi/)

101

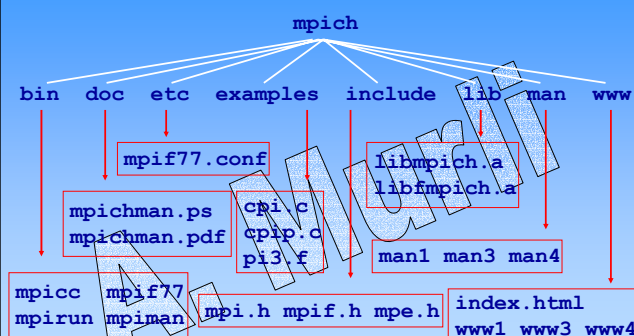


[www-unix.mcs.anl.gov/mpi/](http://www-unix.mcs.anl.gov/mpi/)

A. Muri

102

## Esplorazione directory MPI



Aldo Neri, Dipartimento di Informatica, Università di Pisa, Italy