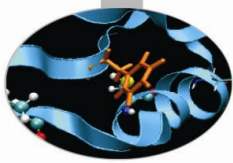
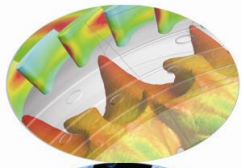
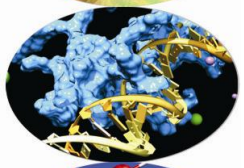
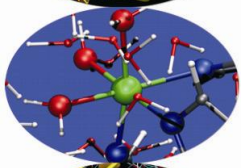
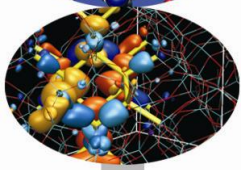
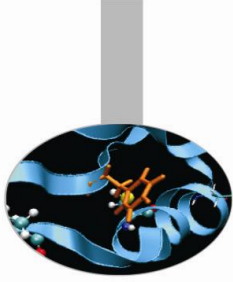


Concetti base di MPI

Introduzione alle tecniche di calcolo parallelo

MPI (Message Passing Interface)

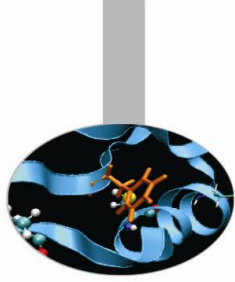


Origini MPI: 1992, "Workshop on Standards for Message Passing in a Distributed Memory Environment“

- **MPI-1.0**: giugno 1994;
- **MPI-1.1**: giugno 1995;
- **MPI-1.2** e **MPI-2**: giugno 1997
- **MPI-3.0** : novembre 2012

60 esperti da più di 40 organizzazioni (IBM T. J. Watson Research Center, Intel’s NX/2, Express, nCUBE’s Vertex, p4, PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL, …)

Hanno partecipato tra i più importanti costruttori di elaboratori paralleli, ricercatori da centri di ricerca universitari, governativi e dell’industria privata



Versioni MPI

Versioni di MPI più utilizzate e di pubblico dominio

MPICH : Argonne National Laboratory

Open MPI : implementazione "open source" di MPI-2

CHIMP/MPI : Università di Edinburgo

LAM : Ohio Supercomputer Center

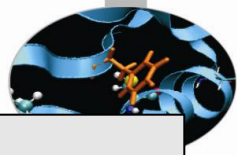
Con MPI bastano sei funzioni per realizzare un programma parallelo.

Se invece viene richiesta una maggiore complessità, si possono sfruttare le risorse complete di MPI, costituite da oltre cento funzioni.

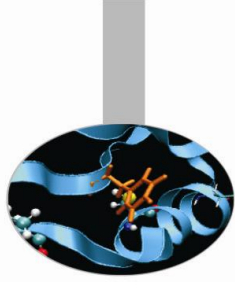
Hello world! (Fortran)

```
program greetings
    include 'mpif.h'
    integer my_rank
    integer p
    integer source
    integer dest
    integer tag
    character*100 message
    character*10 digit_string
    integer size
    integer status(MPI_STATUS_SIZE)
    integer ierr      call MPI_Init(ierr)

    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
    if (my_rank  .NE.  0) then
        write(digit_string,FMT="(I3)") my_rank
        message = 'Greetings from process ' // trim(digit_string) // ' !'
        dest = 0
        tag = 0
        call MPI_Send(message, len_trim(message), MPI_CHARACTER, dest, tag, MPI_COMM_WORLD, ierr)
    else
        do source = 1, p-1
            tag = 0
            call MPI_Recv(message, 100, MPI_CHARACTER, source, tag, MPI_COMM_WORLD, status, ierr)
            write(6,FMT="(A)") message
        enddo
     endif
    call MPI_Finalize(ierr)
end program greetings
```

Hello world! (C/C++)



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

int main( int argc, char *argv[])
{
    int my_rank, numprocs;
    char message[100];
    int dest, tag, source;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

    if (my_rank != 0)
        {
            sprintf(message,"Greetings from process %d !\0",my_rank);
            dest = 0;
            tag = 1000+my_rank;
            MPI_Send(message, sizeof(message),
                MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        } else {
            for (source = 1; source <= (numprocs-1); source++)
               {
                  tag=1000+source;
                  MPI_Recv(message, 100, MPI_CHAR,
                         source, tag, MPI_COMM_WORLD, &status);
                  printf("%s\n",message);
            }
        }

    MPI_Finalize();

    return 0;
}
```



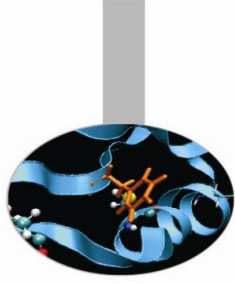
Hello world! (output)

Quando il programma viene eseguito su due processori si ha il seguente risultato:

```
Greetings from process 1!
```

Se viene eseguito su quattro processori:

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```



Costanti predefinite

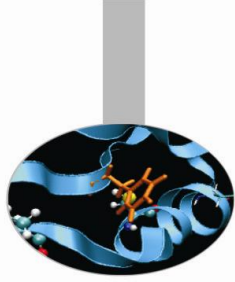
Per un corretto utilizzo di MPI è necessario inserire:

```
INCLUDE   'mpif.h'
```
fortran

```
#include   "mpi.h"
```
C/C++

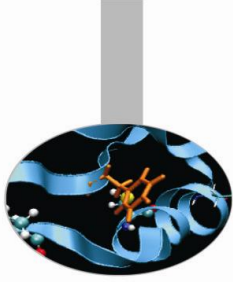
In tale file sono definite le costanti predefinite di MPI. Tra queste si tenga presente quella che rappresenta il valore dell'indice di errore restituito da ciascuna funzione MPI nel caso in cui l'operazione effettuata sia effettivamente riuscita. Ogni subroutine Fortran MPI richiede, come ultimo argomento, un INTEGER di INTENT(OUT) che è, appunto, il codice di errore. Ogni funzione C MPI ritorna un valore int che rappresenta il codice di errore. Se in uscita il codice vale MPI_SUCCESS l'operazione è terminata con successo.

MPI_SUCCESS



```
include "mpif.h"
    integer :: ierror
    ....
    call mpi_send (..., ierror)
    if (ierror .ne. mpi_success) then
        write (*,*)"L'operazione di SEND non e'riuscita"
        stop 777
    end if
```

Eventuali codici di errore diversi da MPI_SUCCESS dipendono dall'ambiente ovvero non sono standard.



Aprire e chiudere processi MPI

In generale, le chiamate alle subroutine MPI hanno la seguente sintassi:

```
call  MPI_name  ( parameter, ..., ierror )
```
fortran

```
rt = MPI_Name  (parameter, ...)
```
C/C++

Per inizializzare l'ambiente MPI è necessario all'inizio chiamare la funzione MPI_Init:

```
call  MPI_INIT ( ierror )
```
fortran

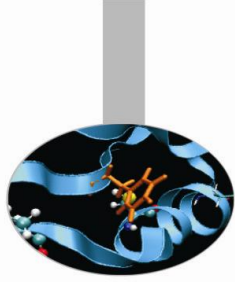
```
int MPI_Init(int *argc, char ***argv)
```
C/C++

Al termine dell'utilizzo della libreria MPI è necessario chiamare la funzione MPI_Finalize, che ha il compito di chiudere i processi MPI che per qualche motivo fossero ancora sospesi e rilasciare la memoria impegnata:

```
call MPI_FINALIZE  ( ierror )
```
fortran

```
rt = MPI_Finalize();
```
C/C++

Gruppi di processi MPI



Un **gruppo** è un insieme ordinato di processi.

Tutti i processi MPI, senza eccezione, **sono riuniti in gruppi**; un processo appartiene necessariamente ad almeno un gruppo di processi.

I processi sono ordinati in modo univoco e sequenziale:

possiedono un **numero identificativo** (**rango**) diverso dagli altri processi dello stesso gruppo.

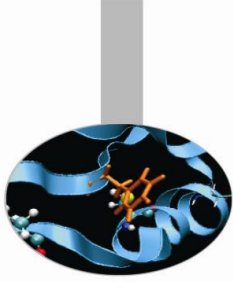
I numeri identificativi interni ad un gruppo sono interi non negativi consecutivi:

Il rango del processo è compreso tra 0 ed N-1 se N è l'estensione del gruppo (group size).

All'avvio di MPI viene automaticamente creato il gruppo che contiene tutti i processi: tale gruppo è associato al comunicatore globale MPI_COMM_WORLD.

Spesso, quando i processi sono pochi, non serve definire nuovi gruppi: per identificare un processo basta usare il rango associato al processo in questo gruppo "universale" definito per default.

Per definire altri gruppi di processi si assegnano i processi a sottoinsiemi disgiunti dell'insieme originario.



Chi sono io? E quanti siamo?

Per avere il numero di processi attivati:

```
call  MPI_COMM_SIZE  ( comm, size, ierror )
```
fortran

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
```
C/C++

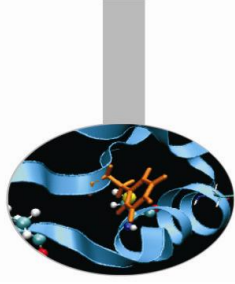
Per conoscere l'identificativo del singolo processo:

```
call  MPI_COMM_RANK ( comm, rank, ierror )
```
fortran

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```
C/C++

Dove `comm` = communicator (in genere: MPI_COMM_WORLD)
`size` = numero di processori attivati
`rank` = rango del processo (numero compreso tra 0 e K-1 dove K = size)
`ierror` = codice di errore

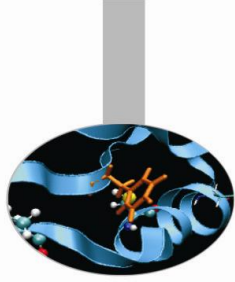
Domini di comunicazione (comunicatori)



Per permettere le comunicazioni all'interno di un nuovo gruppo viene generato un comunicatore. Il comunicatore creato automaticamente da MPI, associato al gruppo di tutti i processi, si chiama convenzionalmente MPI_COMM_WORLD.

Ogni volta che da un gruppo di processi si ritagliano uno o più sottogruppi, per ognuno di questi si deve generare un comunicatore nuovo.

La definizione di un gruppo di processi è locale, realizzata a livello di processo. Viceversa la generazione di un nuovo comunicatore è un'operazione globale, che coinvolge tutti i processi del gruppo.



Comunicazioni punto a punto

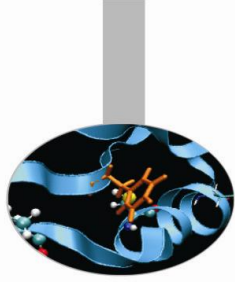
MPI mette a disposizione delle funzioni di comunicazione tra coppie di processi, o comunicazioni punto-punto.

A livello applicativo i messaggi sono identificati da un indice (rank) e da un'etichetta (tag), sempre riferiti allo specifico comunicatore. L'ambiente comunicativo è definito dal comunicatore. Indici ed etichette dei messaggi possono essere indefiniti (wildcarded), ma i comunicatori devono essere sempre specificati individualmente.

Al contrario le comunicazioni collettive, che coinvolgono tutti i processi nell'ambito del comunicatore, non possono far riferimento a messaggi con un'etichetta specifica.

Una comunicazione si dice localmente completa quando il processo ha completato il suo ruolo nella comunicazione stessa. Una comunicazione si dice globalmente completa quando tutti i processi coinvolti hanno terminato il loro ruolo, ovvero se è localmente completa per tutti i processi coinvolti.

Tipi di messaggio



Esistono due categorie di messaggi: **bloccanti** e **non bloccanti**.

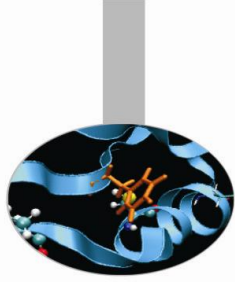
Le subroutine che gestiscono **messaggi bloccanti** non terminano fino a quando i dati passati in argomento ritornano ad essere modificabili senza pericolo di alterare il contenuto del messaggio spedito o ricevuto.

Queste subroutine sono **molto affidabili** (MPI_Send, MPI_Recv) ma possono **rallentare di molto il calcolo** perché il processo resta bloccato fino a che il messaggio non viene trasferito.

Le subroutine che gestiscono messaggi **non bloccanti restituiscono il controllo al programma chiamante nel più breve tempo necessario** per attivare la procedura di scambio messaggi **ma non verificano che l'intero messaggio sia stato veramente e totalmente spedito o ricevuto**.

I vettori di dati passati in argomento non possono essere subito usati o modificati ma occorre prima chiamare opportune subroutine e verificare che l'operazione di spedizione sia stata VERAMENTE terminata (usando le subroutine MPI_Wait oppure MPI_Test). Le subroutine non bloccanti, distinte per la "I" iniziale (MPI_Isend, MPI_Irecv) sono perciò più veloci ma di uso pericoloso.

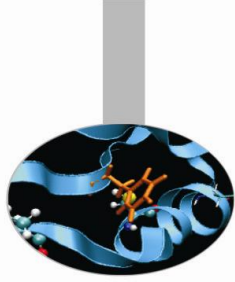
Modi di spedizione



In MPI esistono quattro modi di spedizione

- **Bufferizzato** - Il messaggio viene sempre trasferito su un'area di memoria fornita dal processo mittente. La bufferizzazione può riguardare sia i messaggi bloccanti (MPI_Bsend) che quelli non bloccanti (MPI_Ibsend). Questi ultimi tuttavia presentano il rischio di superare la memoria disponibile al programma perché il mittente resta libero di avviare un numero anche elevatissimo di messaggi senza accorgersi che il sistema, non riuscendo a smistarli in tempo, sta andando in crisi.

- **Sincrono** - L'operazione di spedizione è completata solo se la corrispondente operazione di ricezione è stata avviata e il ricevente ha quindi fornito la memoria in cui mettere i dati. Questa modalità non è rischiosa dal punto di vista della allocazione di memoria perché il sistema usa sempre lo spazio fornito da mittente e ricevente. Il difetto è la lentezza dovuta al tempo necessario perché il ricevente e il mittente siano entrambi in sincronia. Le chiamate sono MPI_Ssend e MPI_Issend.

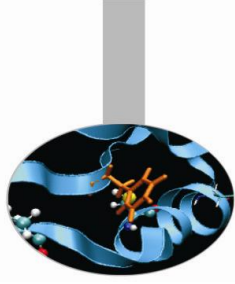


Modi di spedizione

- **Standard** - L'operazione viene gestita liberamente dal sistema che può decidere autonomamente di usare o non usare un buffer per effettuarla. Se il sistema decide di bufferizzare il messaggio spedito, deve anche autonomamente fornire la memoria necessaria (che non viene fornita dal mittente come nella spedizione bufferizzata) e questo ancora una volta può comportare il rischio di eccedere nella memoria allocata. Chiamate: MPI_Send e MPI_Isend.

- **Pronto** (ovvero Ready) - L'operazione di spedizione viene attuata senza nessun controllo perché si è certi che la spedizione avviene SEMPRE dopo che il ricevente si è messo in stato di attesa. Tuttavia, se questo non avviene, i risultati sono completamente erronei e imprevedibili. Per questo motivo, questa modalità è utile solo quando la sincronizzazione tra processi sia assicurata. Si tratta perciò della modalità più veloce ma anche della più pericolosa. Chiamate: MPI_Rsend e MPI_Irsend.

Modi di spedizione e ricezione (riepilogo)

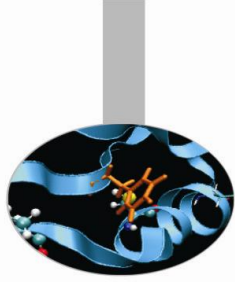


Le modalità di ricezione utilizzabili non dipendono dalla modalità di spedizione e possono essere solo di due tipi: bloccante (RECV) o non bloccante (IRECV).

Riepilogando

SEND	**Blocking**	**Nonblocking**
Standard	mpi_send	mpi_isend
Ready	mpi_rsend	mpi_irsend
Synchronous	mpi_ssend	mpi_issend
Buffered	mpi_bsend	mpi_ibsend

RECEIVE	**Blocking**	**Nonblocking**
Standard	mpi_recv	mpi_irecv



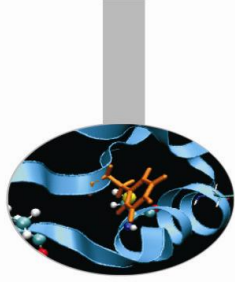
Precedenza tra messaggi

Se due distinti processi spediscono uno stesso tipo di messaggio il destinatario potrà **riceve i messaggi in un ordine temporale diverso da quello di spedizione**.

Bisogna quindi adottare provvedimenti per **imporre il rispetto dell'ordine**

Sono tuttavia sempre rispettate le seguenti due regole:

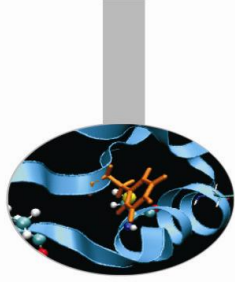
- I **messaggi identici inviati** da uno **stesso mittente** ad uno **stesso destinatario non possono sorpassarsi tra loro**. Il ricevente potrà accedere ad un messaggio solo dopo aver ricevuto tutti i messaggi speditigli prima dallo stesso mittente.
- In caso di messaggi inviati con **spedizione non bloccante** l'ordine di recapito è quello della effettuazione della chiamata che ha iniziato la spedizione. **Le subroutine di spedizione non bloccante restituiscono il controllo prima che realmente la spedizione sia completa** per cui una spedizione di un lungo messaggio potrebbe in realtà essere completata molto dopo l'invio di un breve messaggio iniziato successivamente: anche in questo caso il sorpasso è vietato.



Tipi di dato per spedizione e ricezione messaggi

Ogni messaggio MPI contiene dati formalmente omogenei tra loro.
Si spedisce sempre un certo numero di "oggetti" tutti dello stesso tipo.
I tipi ammissibili sono quelli del sistema MPI e sono pre-definiti o derivati.

Dati primitivi in Fortran	Dati primitivi in C
MPI_INTEGER	MPI_CHAR
MPI_REAL	MPI_SHORT
MPI_DOUBLE_PRECISION	MPI_INT
MPI_COMPLEX	MPI_LONG
MPI_DOUBLE_COMPLEX	MPI_UNSIGNED_CHAR
MPI_LOGICAL	MPI_UNSIGNED_SHORT
MPI_CHARACTER	MPI_UNSIGNED
MPI_BYTE	MPI_UNSIGNED_LONG
MPI_PACKED	MPI_FLOAT
	MPI_DOUBLE
	MPI_LONG_DOUBLE
	MPI_BYTE
	MPI_PACKED



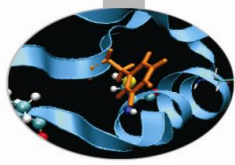
Tipi di dato per spedizione e ricezione messaggi

Il tipo MPI_BYTE non ha corrispondenti in Fortran e C, ma coincide sostanzialmente con il tipo carattere (MPI_CHARACTER/char), anche se viene distinto da questo perché in certi ambienti i bit del carattere sono ordinati in modo diverso (little endian <-> big endian), mentre nel tipo MPI_BYTE viene strettamente mantenuto l'ordine dei bit.

Il tipo di dati MPI_PACKED non corrisponde a nessun tipo Fortran o C perché serve per le operazioni di impacchettamento / spacchettamento.

Notare che il parametro specificato per la spedizione deve corrispondere a quello utilizzato per la ricezione: in caso contrario si possono generare errori nell'esecuzione del programma, proprio a causa di un'errata interpretazione dei dati trasmessi tra i processi.

Sintassi spedizione



La sintassi utilizzata per spedire un messaggio è la seguente:

```
type :: buf(count)
integer :: count, datatype, dest, tag, comm, ierror
call  MPI_spedizione (buf, count, datatype, dest, tag, &
       & comm, ierror )
```

fortran

```
int MPI_Spedizione ( void *buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm);
```

C/C++

dove:

`buf` = insieme di dati che si vogliono inviare

`count` = quantità di elementi che si vogliono inviare

`datatype` = tipo di dati che si invia

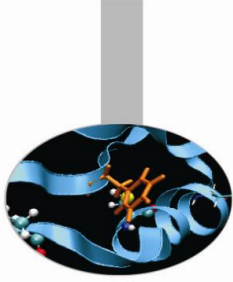
`dest` = identificativo (“rank”) del processo destinatario

`tag` = identificativo del messaggio inviato

`comm` = “communicator”, ne fanno parte il processo che spedisce e quello che riceve

`ierror` = codice di errore

Da notare che alla funzione di spedizione va sempre passato il riferimento al primo elemento del vettore che si intende inviare.



Sintassi ricezione

Per quanto riguarda la ricezione, la sintassi è molto simile:

```
integer :: source, status(*)
call  MPI_ricezione  ( buf, count, datatype, source, &
       & tag, comm, status, ierror )
```
fortran

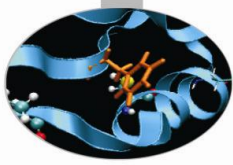
```
int   MPI_ricezione  ( void *buf, int count,   MPI_Datatype
datatype,, int source, int tag,   MPI_Comm comm, MPI_Status
*status );
```
C/C++

Dove:

`source` = identificativo (“rank”) del processo che ha spedito

`status` = vettore di informazioni utili sullo stato del messaggio

Osservazioni (ricezione e spedizione)



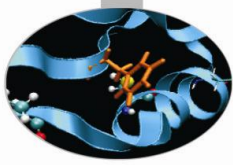
- Quando l'operazione è bloccante, il ritorno da MPI_ricezione si ha solo quando tutto il vettore buf è stato riempito di dati.

- Se si vuole ricevere un messaggio con un identificativo qualsiasi, tag deve valere MPI_ANY_TAG mentre se si vuole ricevere da un mittente qualsiasi, source deve valere MPI_ANY_SOURCE.

- Il vettore status, in uscita, contiene informazioni utili sul messaggio ricevuto. La dimensione di status è MPI_STATUS_SIZE e le informazioni più utili sono:
  - `status (MPI_SOURCE)` = "rank" del mittente che sarebbe ignoto quando si vuole ricevere un messaggio da chiunque ossia da MPI_ANY_SOURCE.
  - `status (MPI_TAG)` = valore di "tag" del messaggio, che sarebbe ignoto quando si vuole ricevere qualsiasi messaggio senza badare a "tag" ossia quando "tag" vale MPI_ANY_TAG.

A differenza della MPI_Recv, la funzione non bloccante MPI_Irecv non ritorna uno STATUS, ma un MPI_Request *request, che può essere utilizzato con MPI_Wait e MPI_Test per controllare il completamento della ricezione, oppure attenderne il completamento.

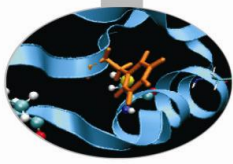
Esempio (ricezione e spedizione)



In questo esempio si è voluto riempire il vettore di integer chiamato box con 2000 elementi in tutto, di cui 1500 ricevuti dal mittente che nel gruppo globale (mpi_comm_world) ha rango 2 ed i restanti 500 ricevuti da qualunque mittente, senza controllare il tag del messaggio.

```
integer, dimension (2000) :: box
integer :: codice_errore, intestazione=5432, mittente=2
integer, dimension (mpi_status_size) :: status
....
call mpi_recv (box(1), 1500, mpi_integer, mittente, &
& intestazione, mpi_comm_world, status, codice_errore)
....
call mpi_recv (box(1501), 500, mpi_integer, mpi_any_source, &
& mpi_any_tag, mpi_comm_world, status, codice_errore)
```

Esempio (ricezione e spedizione)

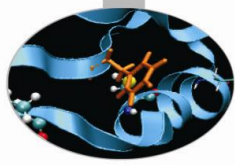


Altro esempio:

spedizione e ricezione di un pacchetto di dati dal processore 0 al processore 1.

```
real :: vector(100)
        integer :: status(MPI_STATUS_SIZE)
        integer :: my_rank, ierr, tag, count, dest, source
        . . .
      if (my_rank == 0) then
          tag = 47
          count = 50
          dest = 1
          call MPI_SEND (vector(51), count, MPI_REAL, dest, tag, &
                      & MPI_COMM_WORLD, ierr)
      else
          tag = 47
          count = 50
          source = 0
          call MPI_RECV (vector(51), count, MPI_REAL, source, tag, &
                      & MPI_COMM_WORLD, status, ierr)
      endif
```

Ricezione e spedizione in una soubroutine



E' possibile combinare in un'unica operazione le funzioni di spedizione e di ricezione: per fare questo si utilizza la seguente subroutine:

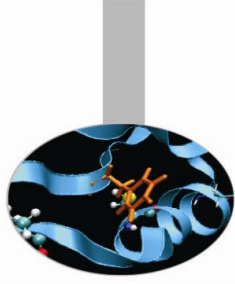
```
type :: SENDBUF, RECVBUF
integer :: SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, &
& RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR

call MPI_SENDRECV ( SENDBUF, SENDCOUNT, SENDTYPE, &
& DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, &
& SOURCE, RECVTAG, COMM, STATUS, IERROR )
```

fortran

```
int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```

C/C++

Ricezione e spedizione in una soubroutine



dove:

SENDBUF = dati da spedire

SENDCOUNT = numero di elementi da spedire

SENDTYPE = tipo dei dati da spedire

DEST = “rank” del processo di destinazione

SENDTAG = etichetta del messaggio da spedire

RECVBUF = dati da ricevere

RECVCOUNT = numero di elementi da ricevere

RECVTYPE = tipo dei dati da ricevere

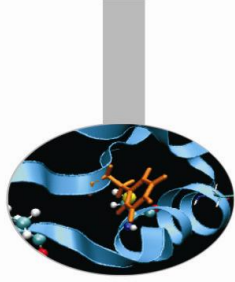
SOURCE = “rank” del processo che spedisce

RECVTAG = etichetta del messaggio da ricevere

COMM = comunicatore

STATUS = vettore di informazioni sullo stato del messaggio

IERROR = codice di errore



mpi_sendrecv_replace

Una funzione analoga, la mpi_sendrecv_replace, permette di utilizzare un unico buffer per la spedizione/ricezione.

La seguente funzione permette di scoprire il numero *count* di elementi, di tipo *datatype*, ricevuti:

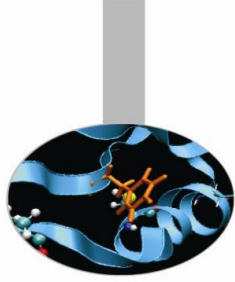
```
call mpi_get_count (status, datatype, count, ierr)
```

fortran

```
int mpi_get_count (MPI_Status *status, MPI_Datatype  datatype,
int *count )
```

C/C++

Controllo operazioni di comunicazione



Le funzioni:

fortran

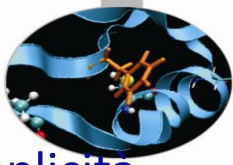
```
call mpi_wait (request_id, return_status, ierr)
call mpi_test (request_id, flag, return_status, ierr)
```

C/C++

```
int MPI_Wait (MPI_Request *request_id, MPI_Status
      *return_status)
int MPI_Test (MPI_Request *request_id, int *flag,
      MPI_Status *return_status)
```

permettono di verificare il completamento di un'operazione di comunicazione. La prima attende fino al completamento locale dell'operazione, la seconda non blocca l'esecuzione ma ritorna FLAG=.TRUE. se la comunicazione è localmente completa; in questo caso il parametro RETURN_STATUS riporta lo stato della ricezione, come per MPI_Recv.

mpi_waitall e mpi_testall



Sono disponibili funzioni analoghe che permettono di controllare una molteplicità di operazioni.

MPI_Waitall non ritorna finché tutte le comunicazioni di LIST_REQUEST non sono terminate e ritorna il vettore LIST_STATUS mentre la funzione **MPI_Testall** permette di verificare se tutte le operazioni di LIST_REQUEST sono terminate (FLAG=.TRUE.).

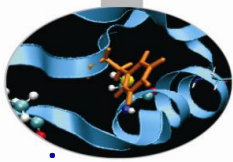
fortran

```
call mpi_waitall(count, list_requests, list_status,
ierr)
call mpi_testall(count, list_requests, flag,
list_status, ierr)
```

C/C++

```
int MPI_Waitall (int count, MPI_Request list_requests[],
MPI_Status list_status[] )
int MPI_Testall (int count, MPI_Request list_requests[],
int *flag, MPI_Status list_status[])
```

mpi_waitany e mpi_testany



La funzione **MPI_WAITANY** ritorna solo dopo che almeno una delle operazioni elencate in LIST_REQUEST è localmente completa. Viceversa la funzione **MPI_TESTANY** permette di verificare se almeno una delle operazioni è completa. Per entrambe INDEX è la posizione in LIST_REQUESTS dell'operazione completa e RETURN_STATUS è l'oggetto "status" della stessa.

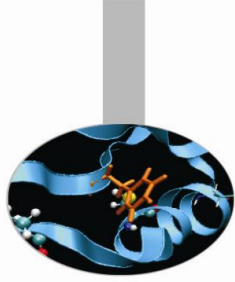
fortran

```
call mpi_waitany(count, list_requests, index,
      return_status, ierr)
call mpi_testany(count, list_requests, index, flag,
      return_status, ierr)
```

C/C++

```
int MPI_Waitany (int count, MPI_Request
      list_requests[], int *index, MPI_Status
      *return_status )
int MPI_Testany(int count, MPI_Request
      list_requests[], int *index, int *flag, MPI_Status
      *return_status )
```

mpi_waitsome e mpi_testsome



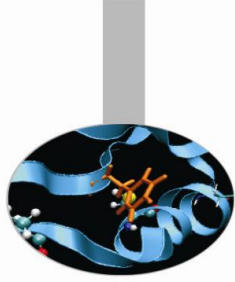
Infine le funzioni **MPI_WAITSOME** e **MPI_TESTSOME** permettono di verificare che qualcuna delle operazioni elencate in LIST_REQUEST è completa:

fortran

```
call mpi_waitsome (count, list_requests, count_done,
      list_index, list_status, ierr)
call mpi_testsome(count, list_requests, count_done,
      list_index, list_status, ierr)
```

C/C++

```
int MPI_Waitsome (int incount, MPI_Request
      list_requests[], int *outcount, int list_index[],
      MPI_Status list_status[] )
int MPI_Testsome(int incount, MPI_Request
      array_of_requests[], int *outcount, int
      array_of_indices[],MPI_Status array_of_statuses[])
```

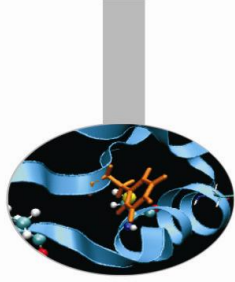


Esercizio (ping-pong)

Il ping-pong è forse il più semplice esempio di comunicazione punto a punto.

Vedere gli allegati:

- Esempio C – ping-pong.c.txt
- Esempio F90 – ping-pong.f.txt



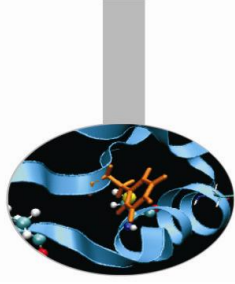
Comunicazioni collettive

Le comunicazioni tra processori **rivestono un ruolo molto importante** in un programma parallelo per cui è fondamentale **cercare di ottimizzarle**.

Un modo per fare questo è quello di utilizzare **comunicazioni collettive**, che sono molto utili anche perché, se MPI è implementato correttamente, le loro **prestazioni non dipendono dall'architettura** della macchina sulla quale verrà eseguito il programma.

Per le comunicazioni collettive non sono utilizzate etichette. Tutte le comunicazioni collettive sono bloccanti.

Con una spedizione collettiva, un insieme di dati viene reso disponibile per ogni processo del gruppo specificato, senza bisogno di definire una routine di tipo MPI_ricezione per i processi che effettivamente dovranno ricevere i dati. Infatti ogni processo appartenente al gruppo COMM chiama tale routine, di conseguenza i processi riceventi sono al corrente di dovere attendere dati provenienti dal processo ROOT.



Comunicazioni collettive

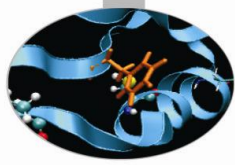
Le comunicazioni collettive MPI possono essere di 2 tipi: **trasferimento di dati** e **computazioni globali**.

Le funzioni di trasferimento dei dati sono di tre tipi:

- **broadcast** - i dati vengono diffusi a tutti i processi
- **gather** - i dati vengono raccolti da tutti i processi
- **scatter** - i dati vengono dispersi a tutti i processi

Le funzioni di computazione globale permettono di svolgere operazioni collettive e sono di due tipi:

- **riduzione**
- **scansione**



broadcast

Quando si vuole inviare lo stesso insieme di dati a tutti i processi facenti parte dello stesso gruppo, si possono utilizzare le consuete subroutine di spedizione specificando tutti i processi a cui si intende spedire l'insieme di dati, oppure si può utilizzare la subroutine **MPI_BCAST** che ha la seguente sintassi:

```
type :: array
integer :: count, datatype, root, comm, ierror
call  MPI_BCAST  ( array, count, datatype, root, comm, ierror )
```

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

dove: `array` = insieme di dati da inviare

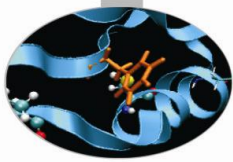
`count` = numero di elementi dell'insieme di dati da inviare

`datatype` = tipo di dati inviato

`root` = processo che possiede i dati da condividere con gli altri

`comm` = gruppo di processi che condivideranno i dati

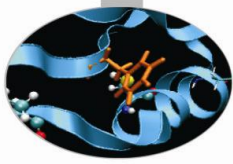
`ierror` = codice di errore



Esempio (broadcast)

```
subroutine GetData (a, b, n, my_rank)

real :: a, b
integer :: n, my_rank, ierr
include 'mpif.h'

if (my_rank == 0) then
                print *, 'Enter a, b, and n'
         read *, a, b, n
endif

call MPI_BCAST (a, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (b, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST (n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )

end subroutine GetData
```

gather e gatherv



fortran

```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, ROOT, &
          & COMM, IERROR, DISP(comm_size)
call MPI_Gather ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
          & RECV_COUNT, RECV_TYPE, ROOT, COMM, IERROR )
```

C/C++

```
int MPI_Gather ( void *send_buf, int send_count, MPI_Datatype
          sendtype, void *recv_buf, int recv_count, MPI_Datatype
          recv_type, int root, MPI_Comm comm )
```

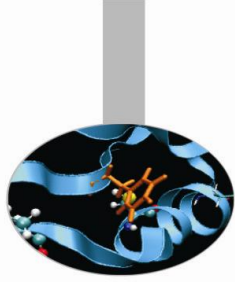
oppure

fortran

```
call MPI_Gatherv ( SEND_BUF, SEND_COUNT, SEND_TYPE, RECV_BUF, &
          & RECV_COUNT, DISP, RECV_TYPE, ROOT, COMM, IERROR )
```

C/C++

```
int MPI_Gatherv ( void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int
*recv_count, int *disp, MPI_Datatype recv_type, int root, MPI_Comm comm )
```



gather e gatherv

SEND_COUNT indica il numero di elementi spediti da ogni processo;

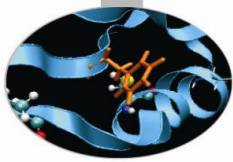
RECV_COUNT indica il numero di elementi ricevuti dal processo;

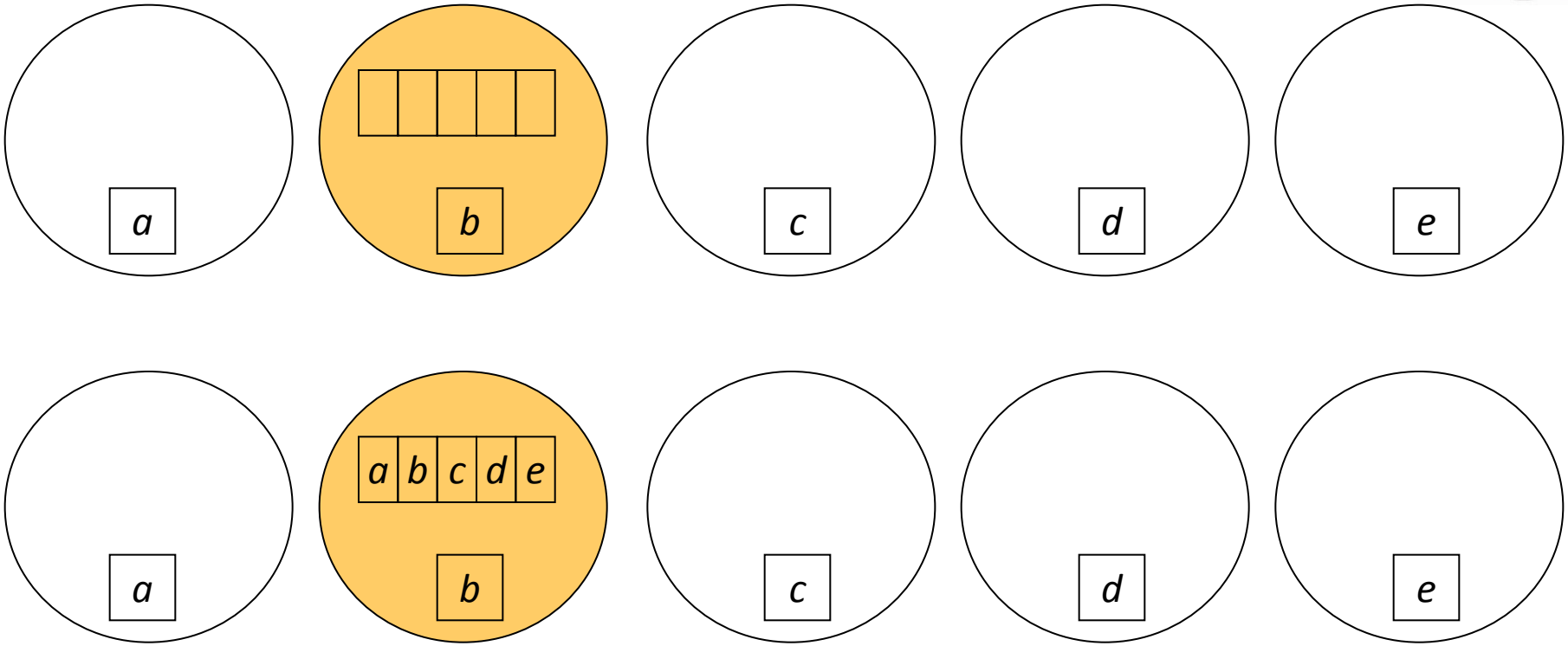
Le varianti delle funzioni con carattere finale 'v' rappresentano un’estensione di funzionalità:

RECV_COUNT (*) è un vettore (ogni processo può inviare un numero differente di elementi)

DISP (*) è un vettore che indica la posizione, nel buffer di ricezione, in cui mettere i dati ricevuti dal processo i-esimo

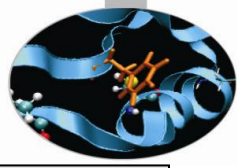
gather





Ogni processo del gruppo specificato spedisce il contenuto di `send_buf` al processo identificato come `root`, il quale concatena i dati ricevuti nella variabile `recv_buf` ordinandoli secondo il numero d'ordine dei processi che hanno inviato i dati.

scatter e scatterv

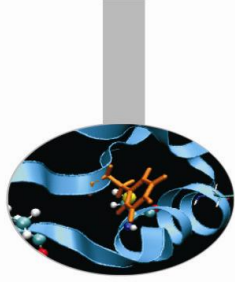


fortran

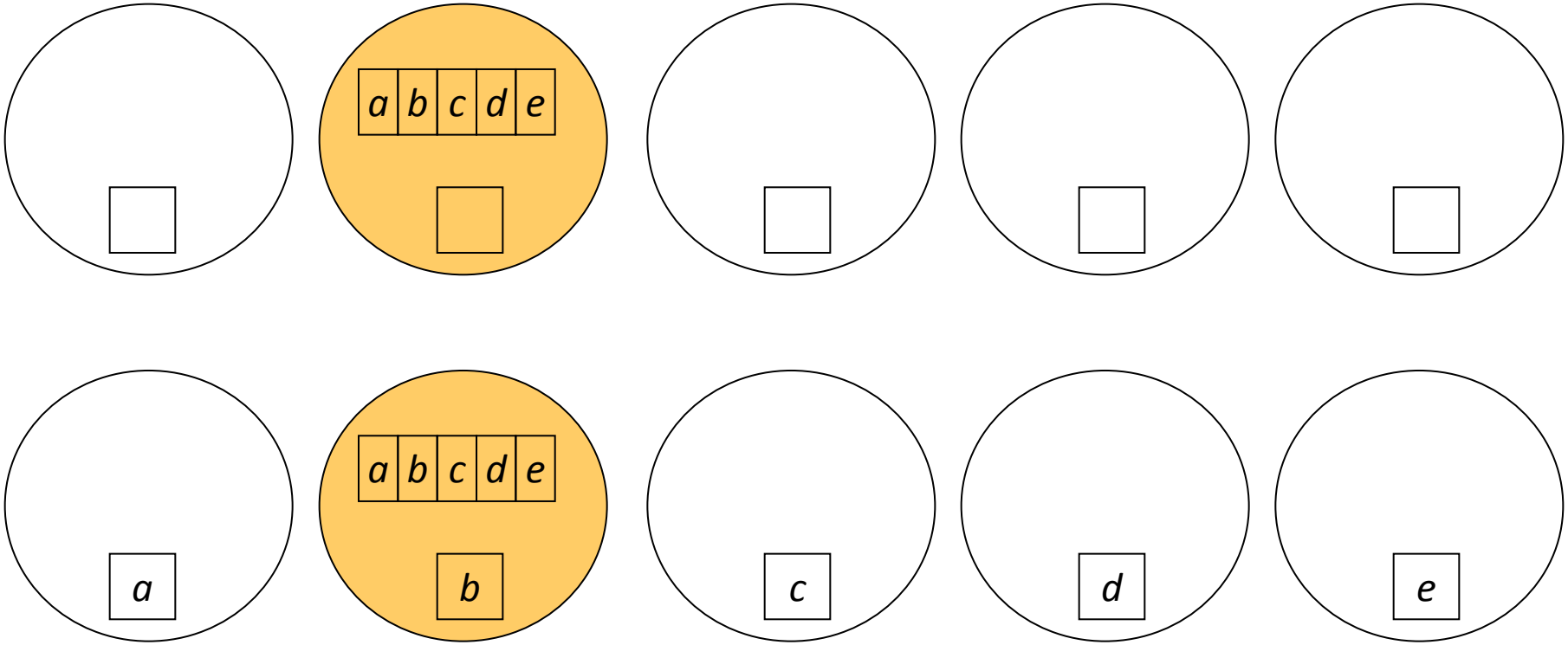
```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT, &
                & RECV_TYPE, ROOT, COMM, IERROR


call MPI_Scatter ( SEND_BUF, SEND_COUNT, SEND_TYPE,
RECV_BUF, RECV_COUNT, RECV_TYPE, ROOT, COMM, IERROR )
```

```
int MPI_Scatter ( void *send_buf, int send_count,
MPI_Datatype send_type, void *recv_buf, int recv_count,
MPI_Datatype recv_type, int root, MPI_Comm comm )
```

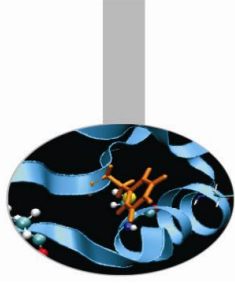
C/C++



scatter



Il processo identificato come `root` distribuisce il contenuto di `send_buf` tra i processi che fanno parte del gruppo specificato. I dati vengono distribuiti secondo il numero identificativo dei processi coinvolti.



allgather e allgatherv

fortran

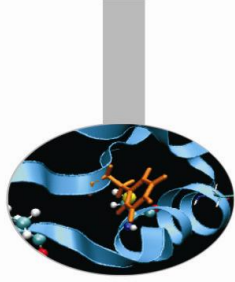
```
type :: SEND_BUF(*), RECV_BUF(*)
integer :: SEND_COUNT, SEND_TYPE, RECV_COUNT,        RECV_TYPE,
COMM, IERROR


call MPI_Allgather ( SEND_BUF, SEND_COUNT, SEND_TYPE,
       RECV_BUF, RECV_COUNT, RECV_TYPE, COMM, IERROR )
```

C/C++

```
int MPI_Allgather ( void *send_buf, int send_count, MPI_Datatype
send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_Comm comm )
```

Con questa subroutine è possibile raccogliere il contenuto delle variabili `send_buf` di ogni processo in ogni processo. Ha lo stesso effetto di una sequenza di chiamate alla subroutine `MPI_Gather`, in ognuna delle quali un differente processo viene identificato come root.

Funzioni di computazione globale



E' anche possibile svolgere operazioni collettive, tramite l'uso di funzioni di computazione globale, che sono di due tipi:

- Riduzione
- Scansione

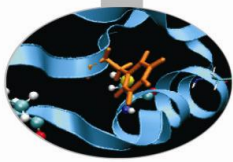
Per ogni operazione di riduzione il risultato può essere:

- memorizzato in un singolo processo
- memorizzato in tutti i processi
- un vettore disperso su tutti i processi

Sono disponibili 3 funzioni di computazione globale:

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter

Reduce



Con questa subroutine, detta di riduzione, tutti i processi di un gruppo forniscono dati che vengono combinati con operazioni quali l'addizione, la ricerca dei valori massimi o minimi, operazioni logiche e altre ancora. Il risultato di tali operazioni è un dato che viene assegnato al processo identificato come ROOT. La sintassi da utilizzare è la seguente:

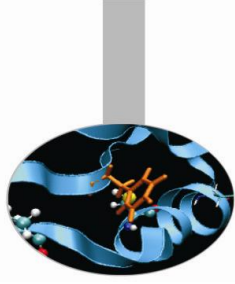
fortran

```
type :: sent_array, recv_array
integer :: count, datatype, op, root, comm, ierror
call MPI_REDUCE ( sent_array, recv_array, count,
datatype, op, &
      & root, comm, ierror )
```

C/C++

```
int MPI_Reduce ( void *sent_array, void   *recv_array,
int count, MPI_Datatype datatype,   MPI_Op op, int root,
MPI_Comm comm )
```

Reduce



`sent_array` = dati spediti al processo che opererà la riduzione

`recv_array` = dati risultanti dalla riduzione

`count` = numero di elementi dell'insieme di dati

`datatype` = tipo dei dati su cui si opera

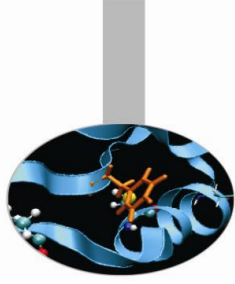
`op` = operazione di riduzione (es.: MPI_SUM, MPI_MAX, …)

`root` = processo in cui vengono salvati i dati risultanti

`comm` = gruppo di processi coinvolti nell'operazione

`ierror` = codice di errore

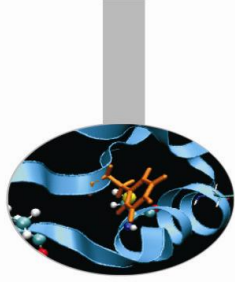
In questo modo, il processo prescelto svolge un'operazione sui dati che riceve da tutti i processi del gruppo specificato e ne salva il risultato in un'apposita area di memoria. La routine `MPI_Reduce` deve essere chiamata da ogni processo coinvolto nell'operazione.



Reduce

Nella seguente tabella sono riportate le operazioni di riduzione utilizzabili:

MPI_MAX	massimo
MPI_MIN	minimo
MPI_SUM	somma
MPI_PROD	prodotto
MPI_LAND	AND logico
MPI_BAND	bit AND
MPI_LOR	OR logico
MPI_BOR	bit OR
MPI_LXOR	XOR logico
MPI_BXOR	bit XOR
MPI_MAXLOC	valore e posizione del massimo
MPI_MINLOC	valore e posizione del minimo



Allreduce

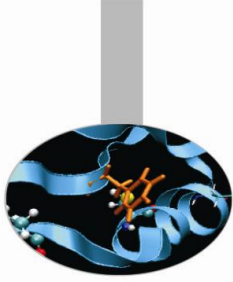
fortran

```
type :: OPERAND(*), RESULT(*)
integer :: COUNT, DATATYPE, OP ,COMM, IERROR
call MPI_AllReduce ( OPERAND, RESULT, COUNT, DATATYPE, OP,
       COMM, IERROR )
```

C/C++

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

In questo caso, il risultato di una operazione di riduzione viene memorizzato nella variabile result appartenente ad ogni processo del gruppo specificato.



Reduce & Scatter

```
<type>, IN :: SENDBUF(*)
<type>, OUT :: RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR


call MPI_REDUCE_SCATTER (SENDBUF, RECVBUF, RECVCOUNTS,
   DATATYPE, OP, COMM, IERROR)
```

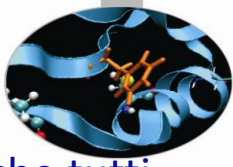
fortran

```
int MPI_Reduce_scatter ( void *sendbuf, void *recvbuf,
       int *recvcounts, MPI_Datatype datatype, MPI_Op op,
       MPI_Comm comm )
```

C/C++

Questa funzione unisce le caratteristiche delle subroutines MPI_Reduce e MPI_Scatter.

Sincronizzare i processi: MPI_Barrier



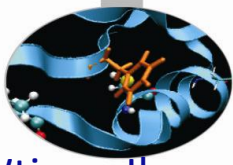
Spesso è utile e necessario imporre delle barriere di sincronizzazione per assicurare che tutti o una parte dei processi siano giunti ad un punto prestabilito del codice. Tuttavia è bene non esagerare nell'uso di questo strumento se non per i momenti in cui è davvero necessario: un uso indiscriminato ed eccessivo della barriera, infatti, può provocare un netto peggioramento delle prestazioni del codice. La sintassi da usare è la seguente:

```
integer :: comm, ierror
call MPI_BARRIER ( comm, ierror )
```
fortran

```
int MPI_Barrier ( MPI_Comm comm )
```
C/C++

dove comm indica il gruppo di processi che si vuole sincronizzare alla barriera mentre ierror è un codice di errore.

In pratica, questa subroutine ha la funzione di bloccare ogni processo fino a quando tutti i processi hanno chiamato la subroutine stessa.



Valutazione delle prestazioni

Per misurare l'efficienza del programma MPI mette a disposizione la funzione MPI_Wtime. Il tempo è ritornato in secondi, con una risoluzione ritornata da MPI_Wtick.

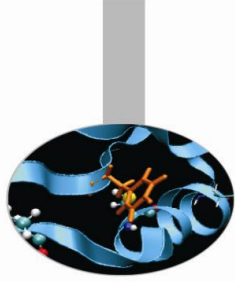
fortran

```
DOUBLE PRECISION t1, t2, tempo
t1 = MPI_WTIME ( )
…
t2 = MPI_WTIME ( )
tempo = t2 – t1
```

C/C++

```
double t1, t2, tempo
t1 = MPI_Wtime()
```

I valori ritornati sono in doppia precisione. La misura è locale al processo che la esegue. Se la variabile MPI_WTIME_IS_GLOBAL è definita e vale .TRUE., il valore temporale ritornato dalla MPI_Wtime è sincronizzato per tutti i processi.



Note sulla Compilazione

Per codici paralleli che utilizzano MPICH, compilare usando:

```
mpif90 -o <nome eseguibile> <file 1> <file 2> … <file n>
```

dove i <file n> rappresentano i simbolici da compilare.

Per lanciare eseguibili in parallelo utilizzare:

```
mpirun -np <numero processori> <nome eseguibile>
```

Per lanciare eseguibili in parallelo su più nodi:

```
mpirun -np <numero processori> -machinefile <lista nodi> \
       <nome eseguibile>
```