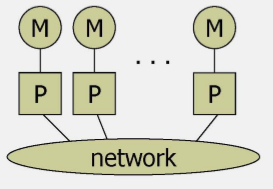
MPI

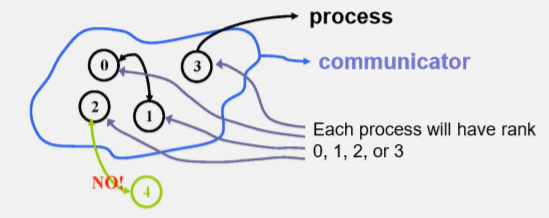
CONCETTI PRINCIPALI

Acronimo di **MESSAGE-PASSING-INTERFACE,** MPI è un classe utilizzata per andare ad implementare un parallelismo distribuito basato sullo scambio di messaggi tra processi. Non vi è un accesso ad una **Shared memory** ma ogni processo utilizza una parte di memoria.

Basato sul paradigma **SPMD:** lo stesso programma gira su ogni processo utilizzato per suddividere il problema in parti. Ogni processo esegue lo stesso codice ma con dati diversi (**Data Parallelism)**.

# COMMUNICATOR AND PROCESS RANK

* **COMMUNICATOR**
  + E’ un’astrazione che indica l’insieme dei processi che, nell’esecuzione del programma, comunicano tra loro.
  + **MPI\_COMM\_WORLD** è quello di default.
  + Possono comunicare solo processi nello stesso communicator.
  + **Intracommunicator**
    - Per comunicazioni all’interno di un gruppo.
  + **Intercommunicator**
    - Per comunicazioni tra gruppi.
* **PROCESS RANK**
  + Identificatore che va ad identificare un processo, all’interno del comunicatore, con un numero da 0 a N-1 (indicano i processi all’interno del communicator).



Ogni variabile “Globale” viene clonata per ogni processo in esecuzione ma **NON** viene mai condivisa.

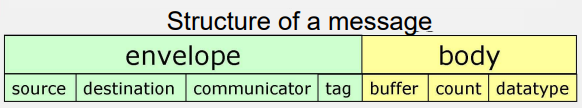
Non esiste una regione parallela.

# SEND AND RECEIVE

Il sistema di comunicazione tra i processi all’interno di un communicator si basa su due funzioni:

* **SEND**
  + Send (void \*sendbuf, int nelems, int dest)
* **RECEIVE**
  + Receive (void \*recvbuf, int nelems, int source)

Si utilizzano dei **tag** per unificare un messaggio.



Innanzitutto è necessario rendere esplicito che il concetto di blocco e sincronizzazione sono completamente diversi.

In realtà è più facile pensare al blocco come una forma di sincronizzazione con l'altro processo, ma non è del tutto vero. La sincronizzazione è un concetto in sé e parliamo di comunicazione sincrona se c'è un coordinamento effettivo in corso con l'altro processo e comunicazione asincrona se non c'è.

Il blocco si riferisce quindi solo al programma in attesa di riutilizzo sicuro dei dati dell'utente; nel caso sincrono una chiamata di blocco significa che i dati sono effettivamente trasferiti, nel caso asincrono significa solo che i dati sono stati trasferiti ad un buffer di sistema.

Una **comunicazione** può essere:

* **Bloccante**: il dato può essere utilizzato in sicurezza perché già trasferito nel buffer o nella memoria locale del ricevente [avviene la “completion” = le variabili sono state ricevute quindi pronte all’uso]:
  + **NO BUFFER**: i due processi devono essere per forza sincronizzati perché deve esser confermata la ricezione delle variabili (SINCRONO).
  + **CON BUFFER**: i due processi sono per logica ASINCRONI poiché chi effettua la comunicazione bloccante si assicura che il dato venga copiato nel buffer e può riutilizzarlo in sicurezza
* **Non Bloccante** (esiste SOLO con il BUFFER [non avrebbe senso infatti che un processo iniziasse ad inviare un messaggio disinteressandosene quando l’altro non sarebbe ancora pronto a ricevere]): il processo attraverso la Send manda i dati al buffer del processo ricevitore però ma non si assicura che questi siano effettivamente ricevuti dal processo ricevente. Può essere Sincrona o Asincrona.
  + SINCRONA se il processore "source" nel momento in cui deve riutilizzare le variabili mandate precedentemente attraverso una comunicazione non bloccante, avvia un processo di sincronizzazione per aspettare una risposta dall'altro per verificare che abbia ricevuto il messaggio. (MPI\_Wait, MPI\_Test = in modo continuo).
    - Per verificare se la comunicazione non bloccante è stata completata si utilizza una “flag” nella quale si memorizza true se la comunicazione è completata, no altrimenti.
  + ASINCRONA se il processo "source" continua il suo lavoro senza dover riutilizzare quelle variabili che ha mandato e quindi continua a non assicurarsi di nulla.

Deadlock

* Momento di stallo in cui un processo sta aspettando che un altro faccia qualcosa e viceversa.

# COMUNICAZIONI PERSISTENTI

Servono per evitare la ripetizione delle comunicazioni con gli stessi argomenti e a ridurre dunque, l’overhead che ne consegue.

* Vengono create attraverso degli oggetti, i quali dopo aver svolto il loro compito, devono essere deallocati.
* Vengono inizializzato un metodo start che andrà a sostituire le funzioni di Send e Recv in modo che queste non siano richiamate in un determinato algoritmo più e più volte.
* Il metodo start ha come parametro un oggetto “request” di tipo send o recv.
* Essendo comunicazioni non bloccanti, si dovrà richiamare la funzione wait per verificare l’effettivo completamento della comunicazione.

# MESSAGE ORDER PRESERVATION

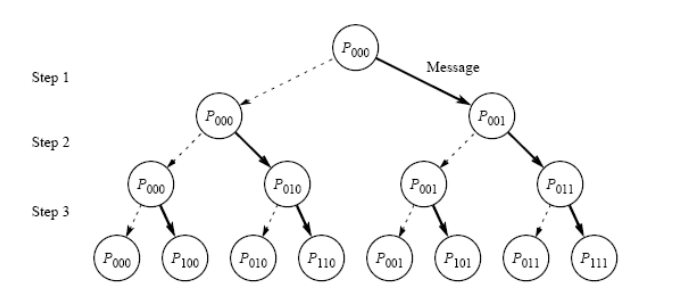
Una caratteristica fondamentale di MPI è la garanzia del mantenimento dell’ordine delle comunicazioni. Con più comunicazioni tra processi, viene rispettato sempre l’ordine di spedizione. Non vi è un accavallamento tra spedizioni.

# COMUNICAZIONI COLLETTIVE

Riguardano più processi nello stesso comunicatore. Questo tipo di comunicazioni sono di tipo **blocking** (comunicazione che può essere seguita in modo **safe**), non richiedono l’utilizzo dei tag e non possiamo utilizzare i tipi derivati di MPI (quei tipi creati dall’utente). Il buffer di ricezione, ovviamente deve essere delle stesse dimensioni del buffer di invio.

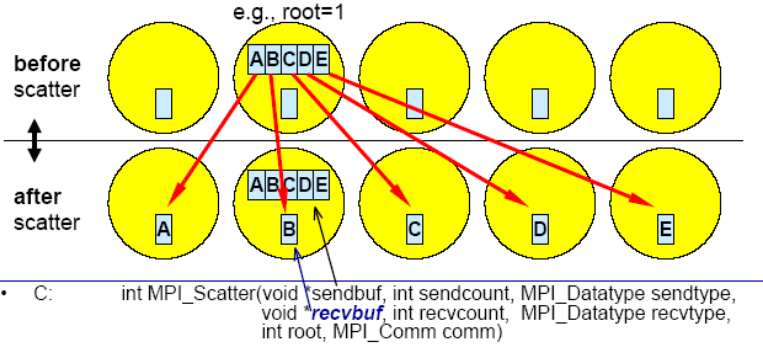
# MPI BROADCAST

Serve per andare a spedire a più processi un valore gestito dal processo iniziale (**ONE TO MANY COMMUNICATION**). E’ molto più efficiente delle altre tipologie di **multiple sends**.



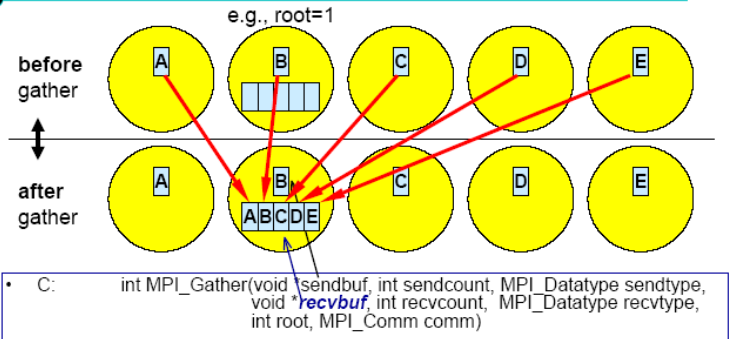
# MPI SCATTER

Più potente della broadcast nell’utilizzo dei vettori **contigui**. Questa funzione è capace di dividere la struttura che deve essere mandata in parti uguali fra i vari processi che devono invece, ricevere.



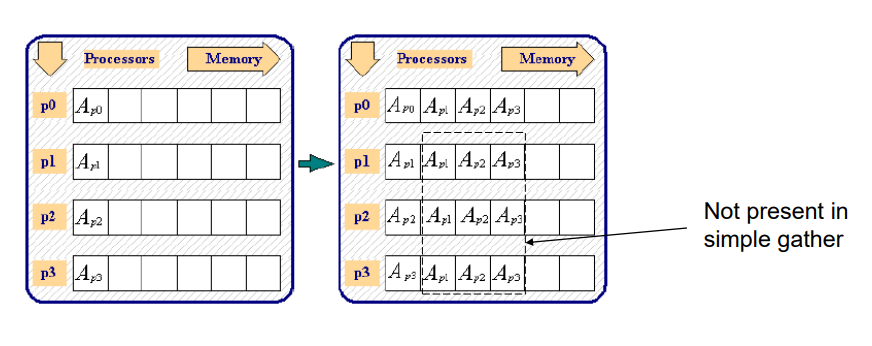
# MPI GATHER

**MANY TO ONE COMMUNICATION.** Ha il funzionamento inverso della Scatter: Più processi inviano i risultati delle loro computazioni al processo root che li elabora per il risultato finale.



# ALL GATHER

**MANY TO MANY COMMUNICATION**. L’unione tra una Gather e una broadcast: ogni processo invia i propri dati sia al root che a tutti gli altri slave.



# REDUCTION

Usata per eseguire operazioni su dati distribuiti tra processi diversi. I dati vengono presi, ridotti e salvati nel processo di **root** (master) o in tutti i processi, il tutto in automatico tramite una funzione.

Racchiude un sacco di funzionalità e in genere, è più efficiente dei metodi equivalenti.



# MPI DATATYPES

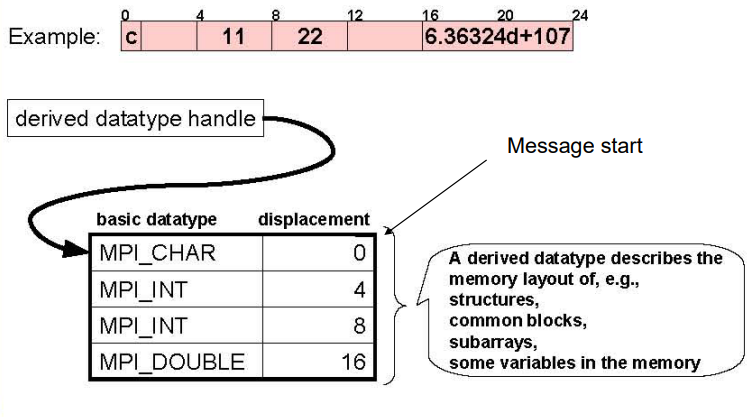
Esistono 3 metodologie per raggruppare dati in messaggi individuali (per dati **NON** contigui principalmente):

* Parametro count
* Tipi Derivati
* MPI\_PACK / MPI\_UNPACK [comprimono e dividono una porzione di dati] -> NON efficiente poiché molto overhead.

# DERIVED DATATYPES

Sono tipologie di dati derivati dai tipi di dato base che MPI mette a nostra disposizione. Per definire un nuovo tipo di dato si effettua una **mappatura** di un dato base in relazione al suo tipo base e al suo displacement (l’allocazione in memoria).

Prima di potremo utilizzare è necessario farne il commit (una specie di dichiarazione).



Per la creazione di un dato derivato MPI mette a disposizione tantissime e variegate funzioni:

* MPI\_Type\_contiguous (dato formato da più elementi adiacenti)
* MPI\_Type\_vector (elementi equidistanti)
* MPI\_Type\_indexed (elementi arbitrari)
* MPI\_Pack (il master impacchetta il messaggio)
* MPI\_Unpack (gli slave devono spacchettare il messaggio ricevuto)