MPI_COMM_WORLD

Un comunicatore e' un handler che rappresenta un gruppo di processiin grado di comunicare

- L'handler del comunicatore di default e' MPI_COMM_WORLD
- Ogni processo puo' utilizzare MPI_COMM_WORLD solo dopo la chiamata a MPI_Init
- Un comunicatore definisce un contesto di comunicazione
- All'interno di un comunicatore ogni processo ha un identificativo unico
- Il nome del comunicatore da utilizzare e' un argomento obbligatorio di tutte le funzioni di comunicazione
- E' possibile utilizzare nello stesso programma piu' di un comunicatore
- In generale, due processi possono comunicare solo se fanno parte dello stesso comunicatore
- In molti casi reali e' sufficiente utilizzare il solo comunicatore di default MPI_COMM_WORLD

Info dal comunicatore

Un processo puo' ottenere alcune informazioni utili dal comunicatore di cui fa parte

- Communicator size
  - Un processo puo' determinarela dimensione di un comunicatore di cui fa parte con una chiamata alla funzione MPI_Comm_size
  - La “size” di un comunicatore e' un intero
- Process rank
  - Un processo puo' determinare il proprio identificativo(rank) in un comunicatore con una chiamata a MPI_Comm_rank
  - I rank dei processi che fanno parte di un comunicatore sono numeri interi, consecutivi ed il piu' piccolo e' lo 0

Binding

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

In ambo i casi **comm** e' il comuincatore di cui si vole conoscere la “size” o del quale si vuole conoscere il rank del processo chiamante. (perche' le due funzioni necessitano puntatori per **rank** e **size** ?)

Provate adesso a realizzare compilare ed eseguire un programma che determini per ogni processo rank e size del comunicatore MPI_COMM_WORLD. (**primompi**)

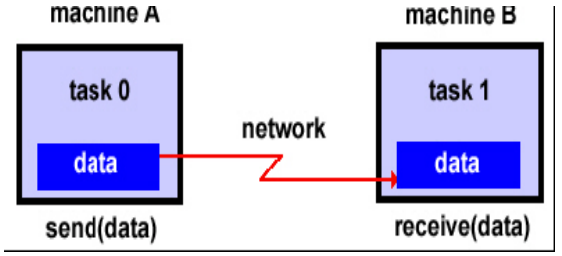
Processor Name

```
int MPI_Get_processor_name(char *name, int
*resultlen);
```

E' possibile cosi' determinare il nome del processore nel quale sta girando il processo . **resultlen** (< **MPI_MAX_PROCESSOR_NAME**) contiene il numero di caratteri in **name**,

Modificare il programma primompi aggiungendo anche la stampa a video del nome del “processore” (**primompiext**)

Comunicazione



Nel modello di programmazione message passing

- ogni processo accede direttamente solo alla propria area di memoria
- la cooperazione tra processi avviene attraverso operazioni esplicite di comunicazione
- l'operazione elementare di comunicazione e' la comunicazione "point-to-point"
- la comunicazione point-to-point vede coinvolti due processi:
  - Il processo sender invia un messaggio
  - Il processo receiver riceve il messaggio inviato
- I dati non sono trasferiti senza la partecipazione esplicita dei due processi

Messaggio

Cos'e' un messaggio

- In una generica operazione di send e receive un messaggio consiste in un certo blocco di dati da trasferire tra i processi

Un messaggio e' costituito da:

- Envelope contiene l'identificativo del messaggio, la sorgente e la destinazione dello stesso
- Body contiene i dati da inviare

L' envelope contiene:

- source: l'identificativo del processo che lo invia
- destination: l'identificativo del processo che lo deve ricevere
- communicator: l'identificativo del gruppo di processi cui appartengono sorgente e destinazione del messaggio
- tag: un identificativo che classifica il messaggio

Il body contiene:

- buffer: i dati del messaggio
- datatype: il tipo di dati contenuti nel messaggio
- count: il numero di occorrenze di tipo datatype contenute nel messaggio

I principali MPI_Datatype

MPI_Datatype	C type
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	signed int

Comunicazione punto a punto

- E’ la funzionalita' fondamentale disponibile nella libreria MPI
- Coinvolge sempre due processi: uno invia un messaggio e l’altro lo riceve
- Per mandare un messaggio il processo source effettua una chiamata ad una funzione MPI, in cui specifica il rank del processo destination nell’appropriato comunicatore (ad esempio MPI_COMM_WORLD)
- Anche il processo destination deve effettuare una chiamata ad una funzione per ricevere il messaggio

MPI_Send

Il processo sorgente effettua una chiamata ad una primitiva con la quale specifica in modo univoco envelope e body del messaggio da inviare:

- l'identita' della sorgente e' implicita (il processo che effettua l'operazione)
- Gli altri elementi che completano la struttura del messaggio
  - identità della destinazione
  - comunicatore da utilizzare
  - dati

  sono determinati esplicitamente dagli argomenti che il processo sorgente passa alla funzione di send

MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
             int dest, int tag, MPI_Comm comm)
```

Tutti gli argomenti sono di input

- **buf** e' l'indirizzo iniziale del send buffer
- **count** e' di tipo int e contiene il numero di elementi del send buffer
- **dtype** e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del send buffer
- **dest** e' di tipo int e contiene il rank del receiver all'interno del comunicatore comm
- **tag** e' di tipo int e contiene l'identificativo del
- messaggio
- **comm** e' di tipo MPI_Comm ed e' il comunicatore in cui avviene la send

MPI_Recv

Il processo destinazione chiama una primitiva, dai cui argomenti e' determinato “in maniera univoca” l’envelope del messaggio da ricevere

MPI confronta l’envelope del messaggio in ricezione con quelli dell’insieme dei messaggi ancora da ricevere (pending messages) e

- se il messaggio e' presente viene ricevuto
- altrimenti l’operazione non puo' essere completata fino a che tra i pending messages ce ne sia uno con l’envelope richiesto

Il processo di destinazione deve disporre di un’area di memoria sufficiente per salvare il body del messaggio

MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
  int src, int tag, MPI_Comm comm, MPI_Status *status)
```

- [OUT] **buf** e' l'indirizzo iniziale del receive buffer
- [IN] **count** e' di tipo int e contiene il numero di elementi del receive buffer
- [IN] **dtype** e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del receive buffer
- [IN] **src** e' di tipo int e contiene il rank del sender all'interno del comunicatore comm
- [IN] **tag** e' di tipo int e contiene l'identificativo del messaggio
- [IN] **comm** e' di tipo MPI_Comm ed e' il comunicatore in cui avviene la send
- [OUT] **status** e' di tipo MPI_Status e conterra' informazioni sul messaggio che e' stato ricevuto

Comunicazione

La comunicazione point-to-point e' la funzionalita' di comunicazione fondamentale disponibile in MPI

- Concettualmente la comunicazione point-to-point e' molto semplice:
  - Un processo invia un messaggio
  - Un altro processo lo riceve

- Nella pratica le cose sono meno semplici:
  - Il ritorno da una primitiva di comunicazione (ovvero la possibilita' di eseguire l'istruzione successiva) puo' avvenire quando
    - la comunicazione e' completata (**blocking send**)
    - la comunicazione e' avviata (**non-blocking send**)
  - Esistono diverse modalità di completamento di un'operazione di send

Modalita' di comunicazione

Ottica globale

- **SINCRONA:** il mittente sa se il messaggio e' arrivato o meno (ad esempio fax)
- **ASINCRONA:** Il mittente non sa se il messaggio e' arrivato o meno (ad esempio lettera)

Ottica locale

- **BLOCCANTE:** il controllo e' restituito al processo che ha invocato la primitiva di comunicazione solo quando la stessa e' stata **completata**
- **NON BLOCCANTE:** il controllo e' restituito al processo che ha invocato la primitiva di comunicazione quando la stessa e' stata eseguita. Il controllo sull'effettivo **completamento** della comunicazione deve essere fatto in seguito, nel frattempo il processo può eseguire altre operazioni

Completamento della comunicazione

Quando un comunicazione si puo' considerare completata ?

- Una comunicazione point-to-point e' considerata **completata localmente** quando le aree di memoria coinvolte nel trasferimento di dati possono essere riutilizzate in modo “sicuro”
  - dal punto di vista di chi spedisce dati la comunicazione e' completata se l'area di memoria puo' essere sovrascritta
  - dal punto di vista di chi riceve dati la comunicazione e' completata se le variabili ricevute possono essere utilizzate
- Dal punto di vista **globale** una comunicazione puo' essere considerata completata se e solo se tutti i processi coinvolti hanno completato le rispettive operazioni relative alla comunicazione. Parafrasando un'operazione di comunicazione e' completata globalmente se e solo se e' completata localmente su tutti i processi coinvolti

Classificare i tipi di comunicazione

Modalita' di blocking

SEND Non-Blocking: Ritorna “immediatamente”. Il buffer del messaggio non deve essere sovrascrittosubito dopo il ritorno al processo chiamante, ma si deve controllare che la SEND sia completata localmente.

SEND Blocking: Ritorna quando la SEND e' completata localmente. Il buffer del messaggio puo' essere sovrascritto subito dopo il ritorno al processo chiamante.

RECEIVE Non-Blocking: Ritorna “immediatamente”. Il buffer del messaggio non deve essere letto subito dopo il ritorno al processo chiamante, ma si deve controllare che la RECEIVE sia completata localmente.

RECEIVE Blocking: Ritorna quando la RECEIVE e' completata localmente. Il buffer del messaggio puo' essere letto subito dopo il ritorno.

Modalita' di comunicazione

La modalita' di una comunicazione punto-punto specifica quando un'operazione di SEND puo' iniziare a trasmettere e quando puo' ritenersi completata. MPI prevede 4 modalita':

- **Syncronous Send:** La SEND puo' iniziare (a trasmettere) anche se la RECEIVE corrispondente non e' iniziata. Tuttavia, la SEND e' completata solo quando si ha garanzia che il processo destinatario ha eseguito e completato la RECEIVE.
- **Buffered Send:** Simile alla modalita' sincrona per l'inizio della SEND. Tuttavia, il completamento e' sempre
- indipendente dall'esecuzione della RECEIVE. (Il messaggio pue' essere bufferizzato per garantire il funzionamento)
- **Standard Send:** La SEND puo' iniziare (a trasmettere) anche se la RECEIVE corrispondente non è iniziata. La semantica del completamento puo' essere sincrona o buffered.
- **Ready Send:** La SEND puo' iniziare (a trasmettere) assumendo che la corrispondente RECEIVE e' iniziata. Tuttavia, il completamento e' sempre indipendente dall'esecuzione della RECEIVE.
- **Receive:** e' completata quando il messaggio e' arrivato

Comunicazione

In MPI vi sono diverse combinazioni possibili tra SEND/RECEIVE sincrone e asincrone, bloccanti e non bloccanti

SEND 8 tipi

MPI_[-,I][-,R,S,B]send
[-,I] = modalita' di blocking
[-,R,S,B] = modalita' di comunicazione

RECEIVE 2 tipi

MPI_[-,I]recv
[-,I] = modalita' di blocking

MPI_Ssend

```
int MPI_Ssend(void *buf,int count, MPI_Datatype dtype, int
dest, int tag, MPI_Comm comm)
```

- buf e' l'indirizzo iniziale del send buffer
- count e' di tipo int e contiene il numero di elementi del send buffer
- dtype e' di tipo MPI_Datatype e descrive il tipo di
- ogni elemento del send buffer dest e' di tipo int e contiene il rank del receiver all'interno del comunicatore comm
- tag e' di tipo int e contiene l'identificativo del messaggio
- comm e' di tipo MPI_Comm ed e' il comunicatore in cui avviene la send

Il processo mittente si blocca finche' il messaggio non e' stato ricevuto dal destinatario. E' la moidalita' di comunicazione piu' semplice ed affidabile, ma si possno avere notevoli perdite di tempo.

MPI_Bsend

- Una buffered send e' completata immediatamente, non appena il processo ha copiato il messaggio su un opportuno buffer di trasmissione
- Il programmatore non puo' assumere la presenza di un buffer di sistema allocato per eseguire l'operazione, ma deve effettuare un'operazione di
  - BUFFER_ATTACH per definire un'area di memoria di dimensioni opportune come buffer per il trasferimento di messaggi
  - BUFFER_DETACH per rilasciare le aree di memoria di buffer utilizzate
- Pro
  ritorno immediato dalla primitiva di comunicazione
- Contro
  E' necessaria la gestione esplicita del buffer
  Implica un'operazione di copia in memoria dei dati da trasmettere

MPI_Bsend

```
int MPI_Bsend(void *buf, int count, MPI_Datatype dtype, int
dest, int tag, MPI_Comm comm)
```

- buf e' l'indirizzo iniziale del send buffer
- count e' di tipo int e contiene il numero di elementi del send buffer
- dtype e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del send buffer
- dest e' di tipo int e contiene il rank del receiver all'interno del comunicatore comm
- tag e' di tipo int e contiene l'identificativo del messaggio
- comm e' di tipo MPI_Comm ed e' il comunicatore in cui avviene la send

Gestione dei buffer

```
MPI_Buffer_attach(void *buf, int size)
```

Consente al processo sender di “allocare” il buffer per una successiva chiamata di MPI_Bsend

- .buf e' l’indirizzo iniziale del buffer da “allocare”
- .size e' un int e contiene la dimensione in byte del buffer da “allocare” (size + MPI_BSEND_OVERHEAD).

```
MPI_Buffer_detach(void *buf, int *size)
```

Consente di “deallocare” il buffer creato con MPI_Buffer_attach

- .buf e' l’indirizzo iniziale del buffer da allocare
- .size e' un int e contiene la dimensione in byte del buffer

MPI_Send

- Una standard send e' completata quando il processo sender puo' riutilizzare l'area di memoria contenente i dati oggetto del trasferimento
- Il programmatore non puo' assumere che l'operazione sara' completata prima che:
  - il processo destinazione inizi la ricezione
  - il processo destinazione termini la ricezione
- Una standard send puo' essere implementata come
  - una syncronous send
  - una buffered send, utilizzando pero' un buffer di sistema
- Nel caso in cui il messaggio da inviare non entri completamente nel buffer di sistema, la standard send si comporta come una syncronous send

MPI_Recv

Standard RECEIVE blocking

- Il processo destinazione usa un solo tipo di RECEIVE blocking, indipendentemente dalla particolare funzionalita' di SEND utilizzata nel processo sorgente
- Gli argomenti di tipo envelope sono necessari per richiedere ad MPI la ricezione del giusto messaggio
- I dati ricevuti sono salvati in memoria a partire dalla locazione argomento della funzione di receive
- Al completamento dell'operazione i dati ricevuti posso essere utilizzati

Deadlock

- Si parla di deadlock quando 2 (o piu') processi sono bloccati ed ognuno e' in attesa dell'altro per riprendere l'esecuzione
- Utilizzare operazioni di comunicazione di tipo blocking senza un corretto protocollo di comunicazione puo' portare a situazioni di deadlock

- Esempio:

```
if (myrank = 0)
  Standard SEND A to Process 1
  RECEIVE B from Process 1
else if (myrank = 1)
  Standard SEND B to Process 0
  RECEIVE A from Process 0
endif
```

- in quali condizioni genera un deadlock?

Deadlock

- L'esempio precedente genera un deadlock se la standard send e' implementata come una syncronous send
- In questo caso semplice il deadlock si evita facendo attenzione all'ordine delle chiamate:

```
if (myrank = 0)
  Standard SEND A to Process 1
  RECEIVE B from Process 1
else if (myrank = 1)
  RECEIVE A from Process 0
  Standard SEND B to Process 0
endif
```

MPI_Sendrecv

- La funzionalita' MPI Send-Receive e' utile quando un processo deve sia inviare che ricevere dati
- In tali casi una MPI Send-Receive consente di utilizzare un unico costrutto, evitando situazioni di deadlock
- MPI Send-Receive e' un'estensione delle comunicazioni point-to-point
- La MPI Send-Receive e' completamente compatibile con le altre funzionalita' di comunicazione point-to-point, ovvero puo' essere utilizzata per:
  - Inviare un messaggio che sara' ricevuto tramite una funzione di receive
  - Ricevere un messaggio inviato tramite una funzione di send
- E' utile per implementare un pattern di comunicazione di tipo shift, utilizzando funzioni blocking

MPI_Sendrecv

```
int MPI_Sendrecv(void *sbuf,int scount, MPI_Datatype
s_dtype, int dest, int stag, void *dbuf, int dcount,
MPI_Datatype d_type, int src, int dtag,MPI_Comm comm,
MPI_Status *status)
```

- [IN] sbuf e' l'indirizzo iniziale del send buffer
- [IN] scount contiene la size del send buffer
- [IN] s_dtype descrive il tipo di ogni elemento del send buffer
- [IN] dest e' il rank del receiver all'interno del comunicatore comm
- [IN] stag e' l'identificativo del messaggio da inviare
- [OUT] dbuf e' l'indirizzo iniziale del receive buffer
- [IN] dcount contiene la size del receive buffer
- [IN] d_dtype descrive il tipo di ogni elemento del receive buffer
- [IN] src e' il rank del receiver all'interno del comunicatore comm
- [IN] dtag e' l'identificativo del messaggio da ricevere
- [IN] comm e' il comunicatore in cui avviene la sendrecv
- [OUT] status conterra' informazioni sul messaggio

Non-blocking

- Una comunicazione non-blocking e' tipicamente costituita da tre fasi successive:
  - L'inizio della operazione di send/receive del messaggio
  - Lo svolgimento di un'attivita' che non implichi l'accesso ai dati coinvolti nella operazione di comunicazione avviata
  - L'attesa per il completamento della comunicazione
- Pro
  - Performance: una comunicazione non-blocking consente di:
    - sovrapporre fasi di comunicazioni con fasi di calcolo
    - ridurre gli effetti della latenza di comunicazione
  - Le comunicazioni non-blocking evitano situazioni di deadlock
- Contro
  - La programmazione di uno scambio messaggi con funzioni di comunicazione non-blocking e' (leggermente) piu' complicata

MPI_Isend

- Dopo che la spedizione e' stata avviata il controllo torna al processo sender

- Prima di riutilizzare le aree di memoria coinvolte nella comunicazione, il processo sender deve controllare che l'operazione sia stata completata, attraverso opportune funzioni della libreria MPI

- La semantica di MPI definisce anche per la send non-blocking le diverse modalita' di completamento della fase di comunicazione (quindi standard MPI_Isend, synchronous MPI_Issend, ready MPI_Irsend e buffered MPI_Ibsend)

MPI_Isend

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int
    dest, int tag, MPI_Comm comm, MPI_Request *request)
```

.

.[IN] buf e' l'indirizzo iniziale del send buffer
.[IN] count e' di tipo int e contiene il numero di elementi del send buffer
.[IN] dtype e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del send buffer
.[IN] dest e' di tipo int e contiene il rank del receiver all'interno del comunicatore comm
.[IN] tag e' di tipo int e contiene l'identificativo del messaggio
.[IN] comm e' di tipo MPI_Comm ed e' il comunicatore in cui avviene la send
.[OUT] request e' di tipo MPI_Request e conterra' l'handler necessario per referenziare l'operazione di send

MPI_Irecv

- Dopo che la fase di ricezione e' stata avviata il controllo torna al processo receiver

- Prima di utilizzare in sicurezza i dati ricevuti, il processo receiver deve verificare che la ricezione sia completata, attraverso opportune funzioni della libreria MPI

- Una receive non-blocking puo' essere utilizzata per ricevere messaggi inviati sia in modalità blocking che non blocking

MPI_Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int
    src, int tag, MPI_Comm comm, MPI_Request *request)
```

- [OUT] buf e' l'indirizzo iniziale del receive buffer
- [IN] count e' di tipo int e contiene il numero di elementi del receive buffer
- [IN] dtype e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del receive buffer
- [IN] src e' di tipo int e contiene il rank del sender all'interno del comunicatore comm
- [IN] tag e' di tipo int e contiene l'identificativo del messaggio
- [IN] comm e' di tipo MPI_Comm ed è il comunicatore in cui avviene la send
- [OUT] request e' di tipo MPI_Request e conterrà l'handler necessario per referenziare l'operazione di receive

Test di completamento

Quando si usano comunicazioni point-to-point non blocking e' essenziale assicurarsi che la fase di comunicazione sia completata per

- utilizzare i dati del buffer (dal punto di vista del receiver)
- riutilizzare l'area di memoria coinvolta nella comunicazione (dal punto di vista del sender)

La libreria MPI mette a disposizione dell'utente due tipi di funzionalita' per il test del completamento di una comunicazione

- tipo WAIT : consente di fermare l'esecuzione del processo fino a quando la comunicazione in argomento non sia completata
- tipo TEST: ritorna al processo chiamante un valore TRUE se la comunicazione in argomento e' stata completata, FALSE altrimenti

Test di completamento

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

- [IN] request e' l'handler necessario per referenziare la comunicazione di cui attendere il completamento
- [OUT] status conterra' lo stato (envelope) del messaggio di cui si attende il completamento

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

- [OUT] flag conterra' TRUE se la comunicazione e' stata completata, FALSE altrimenti
- [OUT] status conterra' lo stato (envelope) del messaggio di cui si attende il completamento

non-blocking

- Utilizzando comunicazioni non-blocking, puo' accadere che, in un certo istante, ne risultino avviate diverse
- MPI dispone di primitive che consentono ad un processo di testare/attendere il completamento di una serie di comunicazioni che lo coinvolgono

- Esistono tre tipologie di primitive di questo tipo, quelle che consentono di verificare il completamento di
  - tutte le comunicazioni in corso (MPI_Testall / MPI_Waitall)
  - una ed una sola delle comunicazioni in corso (MPI_Testany / MPI_Waitany)
  - alcune (alemno una) delle comunicazioni in corso (MPI_Testsome / MPI_Waitsome)

```
int MPI_Waitany (int count, MPI_Request array_of_requests[],
                 int *index, MPI_Status *status)
```

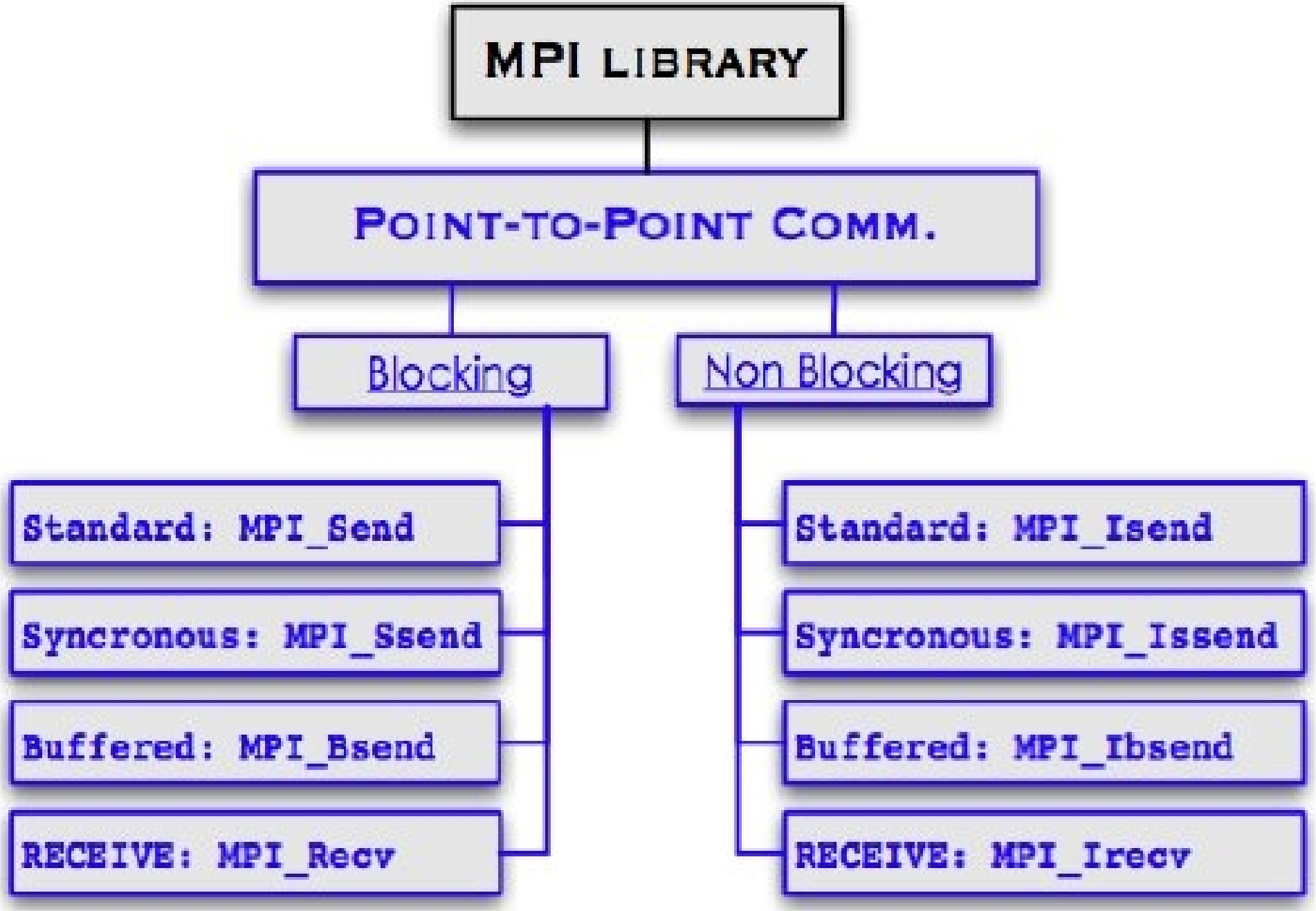
```
int MPI_Waitsome(int incount, MPI_Request
array_of_requests[], int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
```

```
int MPI_Waitall (int count, MPI_Request array_of_requests[],
                 MPI_Status array_of_statuses[])
```

```
int MPI_Testany(int count, MPI_Request array_of_requests[],
        int *index, int *flag, MPI_Status *status);
```

```
int MPI_Testsome(int incount, MPI_Request
array_of_requests[], int *outcount, int array_of_indices[],
        MPI_Status array_of_statuses[]);
```

```
int MPI_Testall (int count, MPI_Request array_of_requests[]
        , int *flag, MPI_Status array_of_statuses[]);
```


MPI LIBRARY
POINT-TO-POINT COMM.
Blocking
Non Blocking
Standard: MPI_Send
Syncronous: MPI_Ssend
Buffered: MPI_Bsend
RECEIVE: MPI_Recv
Standard: MPI_Isend
Syncronous: MPI_Issend
Buffered: MPI_Ibsend
RECEIVE: MPI_Irecv

Collettive

Alcune sequenze di comunicazione sono cosi' comuni nella pratica della programmazione parallela che la libreria MPI prevede una serie di routine specifiche per eseguirle: **le funzioni di comunicazione collettiva**

- Le comunicazioni collettive sono sempre costruite sulle funzioni di comunicazione point-to-point, ma
  - utilizzano i piu' efficienti algoritmi noti per l'operazione che implementano
  - Nascondono all'utente della libreria MPI sequenze di send/receive sovente complicate
- Una comunicazione collettiva coinvolge tutti i processi di un comunicatore
- Qualora si voglia effettuare una comunicazione collettiva in cui siano coinvolti solo una frazione dei processi di MPI_COMM_WORLD, e' necessario definire un comunicatore ad hoc

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

- [IN] comm e' di tipo MPI_Comm ed e' il comunicatore a cui appartengono i processi da sincronizzare

Nella programmazione parallela ci sono occasioni in cui alcuni processi non posso procedere fino a quando altri processi non hanno completato una operazione.

Chiamare una funzione di tipo BARRIER comporta che tutti i processi del comunicatore argomento si fermano in attesa che l'ultimo non abbia eseguito la stessa chiamata

La BARRIER e' eseguita in software, dunque puo' comportare un certo overhead: inserire chiamate BARRIER solo se strettamente necessario

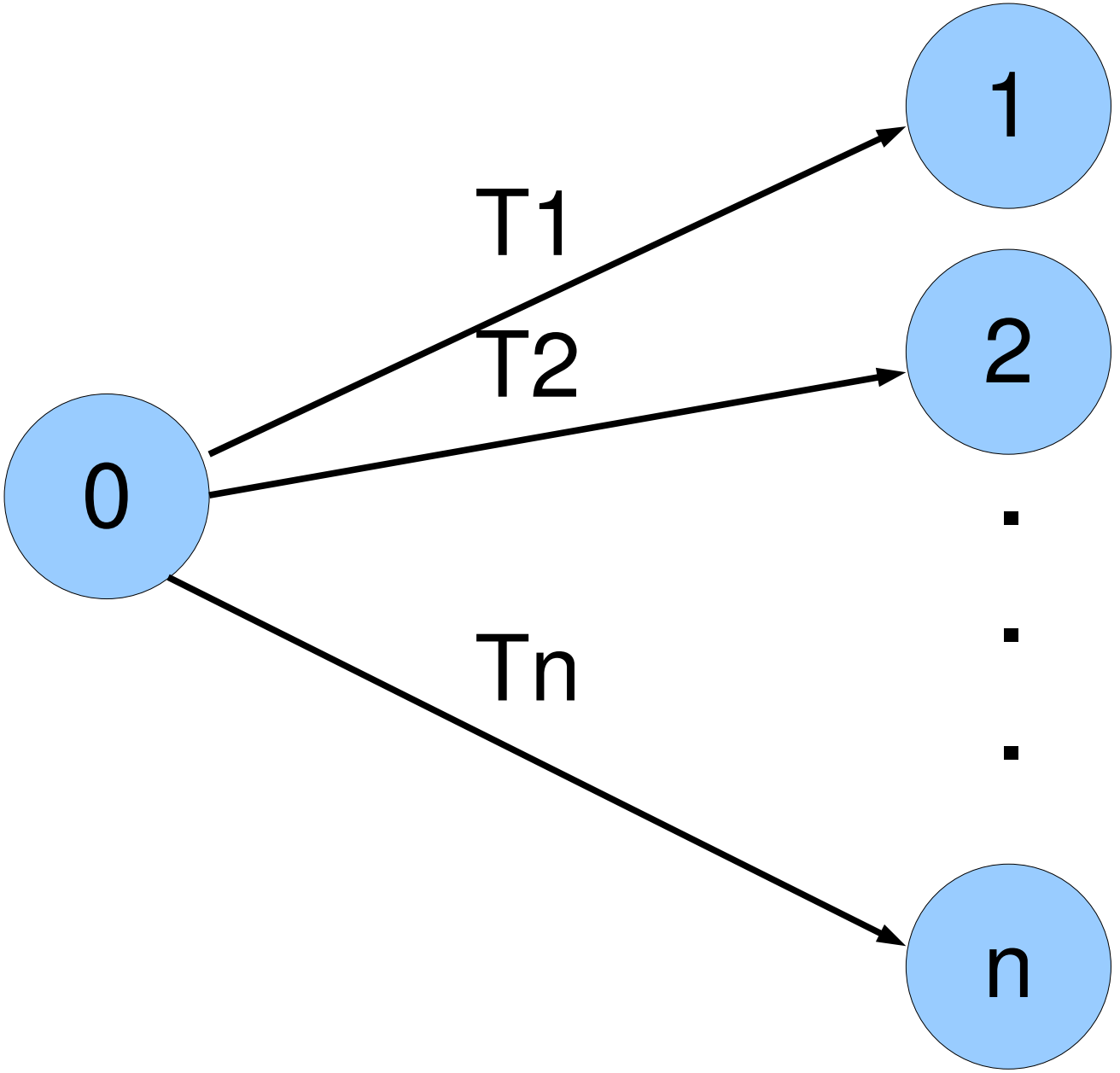
Esempio (20_barrier): cosa succede con o senza barrier ? Senza barrier il processo “recv” attende dati mentre il processo “send” riempie la matrice.

```
$ mpiexec -n 2 ./alloc 1000
I am proces   0 of   2 my name is cg00 (./alloc)
I am proces   1 of   2 my name is cg06 (./alloc)
Send time 3: 0.100014 s
Recv time 3: 0.100176 s

$ mpiexec -n 2 ./alloc 1000
I am proces   1 of   2 my name is cg06 (./alloc)
I am proces   0 of   2 my name is cg00 (./alloc)
Send time 3: 0.101689 s
Recv time 3: 0.165475 s
```

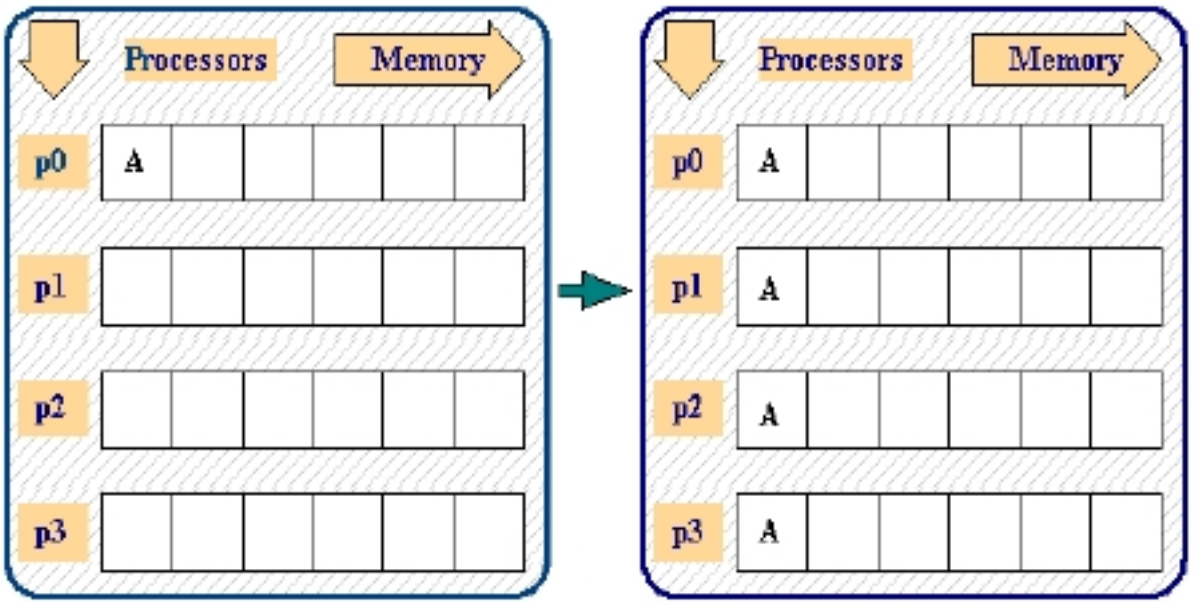
Attenti a dove posizionate le chiamate collettive

Broadcast



Broadcast

E' un'operazione **One to All.** La funzionalita' di BROADCAST consente di copiare dati dalla memoria di un processo root (p0 nella figura) alla stessa locazione degli altri processi appartenenti al comunicatore utilizzato.



Esercizio: provate ad implementare una funziona che faccia il broadcast di un vettore di double.

```
$ mpiexec -n 9 ./bcast 5000000
I am proces   0 of   9 my name is cg00 (./bcast)
I am proces   2 of   9 my name is cg03 (./bcast)
I am proces   1 of   9 my name is cg02 (./bcast)
I am proces   4 of   9 my name is cg04 (./bcast)
I am proces   3 of   9 my name is cg01 (./bcast)
I am proces   5 of   9 my name is cg06 (./bcast)
I am proces   6 of   9 my name is cg08 (./bcast)
I am proces   7 of   9 my name is cg07 (./bcast)
I am proces   8 of   9 my name is cg05 (./bcast)
Bcast time: 3.994586 s
```

MPI_Bcast

```
int MPI_Bcast(void* buf, int count, MPI_Datatype
dtype, int root, MPI_Comm comm)
```

.

- [IN/OUT] buf e' l'indirizzo del send/receive buffer
- [IN] count e' di tipo int e contiene il numero di elementi del buffer
- [IN] dtype e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del buffer
- [IN] root è di tipo int e contiene il rank del processo root della broadcast
- [IN] comm e' di tipo MPI_Comm ed e' il comunicatore cui appartengono i processi coinvolti nella broadcast

MPI_Bcast

Proviamo a fare un confronto di tempi:

```
21_bcast]$ mpiexec -n 9 ./bcast 10000000
I am proces   0 of   9 my name is cg00 (./bcast)
I am proces   2 of   9 my name is cg03 (./bcast)
I am proces   1 of   9 my name is cg02 (./bcast)
I am proces   3 of   9 my name is cg01 (./bcast)
I am proces   4 of   9 my name is cg04 (./bcast)
I am proces   5 of   9 my name is cg06 (./bcast)
I am proces   6 of   9 my name is cg08 (./bcast)
I am proces   7 of   9 my name is cg07 (./bcast)
I am proces   8 of   9 my name is cg05 (./bcast)
Bcast mine time: 8.010572 s
Bcast time: 2.981955 s
```