# 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[])
```

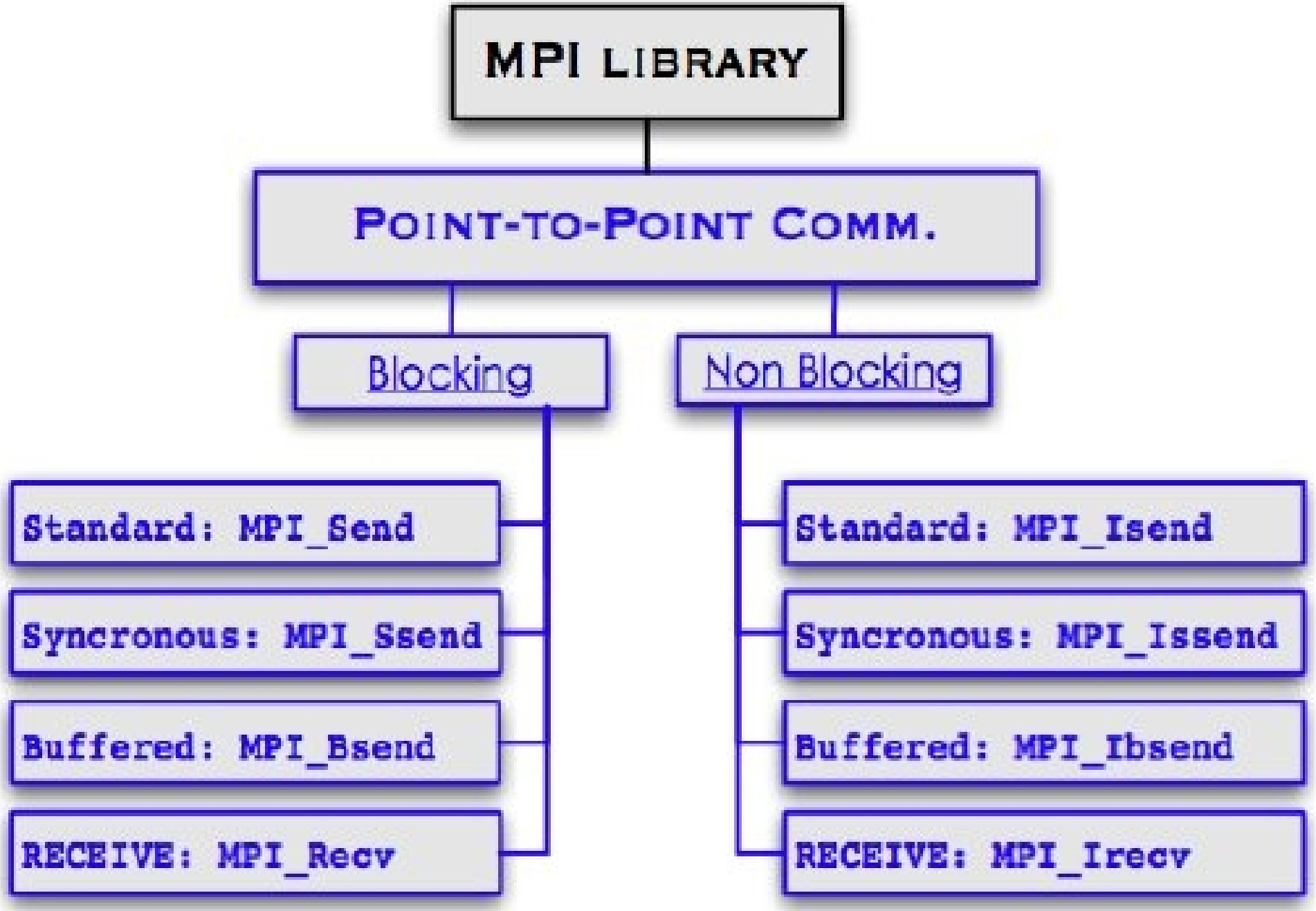
```
int MPI_Testany(int count, MPI_Request array_of_requests[],
        int *index, int *flag, MPI_Status *status);
```

```
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[]);
```

**Esempio (19_nonblock):** semplice esempio di comunicazione point-to-point non bloccante

**Esercizio (20_ring):** Consideriamo il solito insieme di processi con topologia ad anello. Ogni processo spedisce il valore del proprio rank a destra. A regime ogni processo spedisce a destra il valore che riceve da sinistra, fino che non avra' ottenuto di nuovo il proprio rank, allora si ferma.  Il tutto usando MPI_Isend. (Stampante la somma dei valori ricevuti da ogni processo)


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.

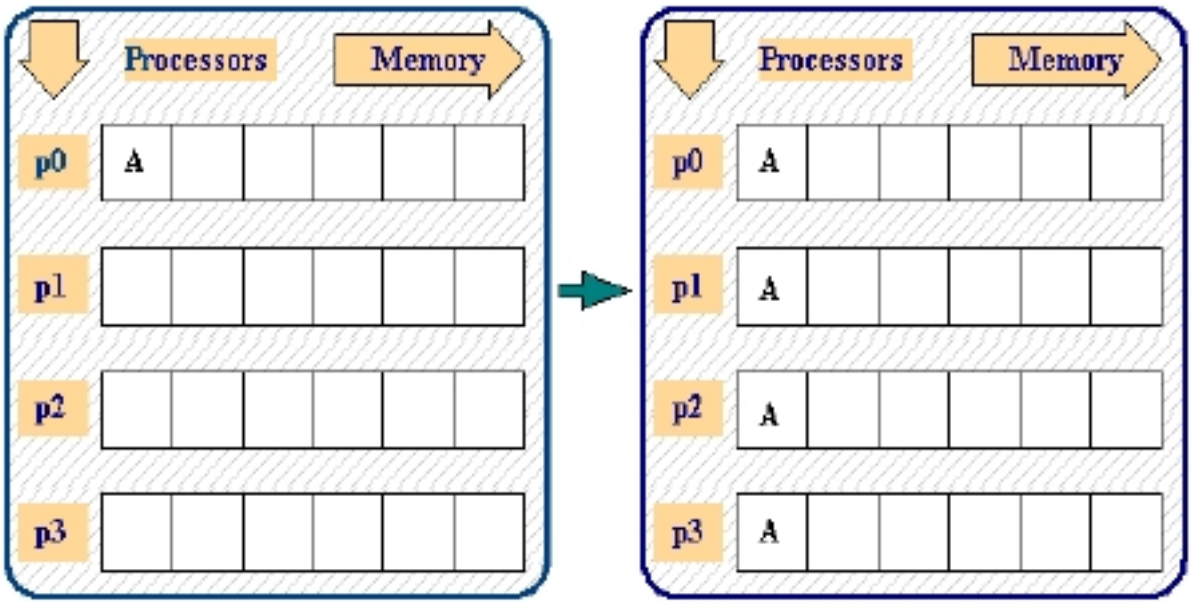
```
$ mpiexec -n 2 ./alloc 1000
I am proces   0 of   2 my name is cg00 (./alloc)
I am proces   1 of   2 my name is cg06 (./alloc)
Send time 3: 0.100014 s
Recv time 3: 0.100176 s

$ 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

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)
```

.

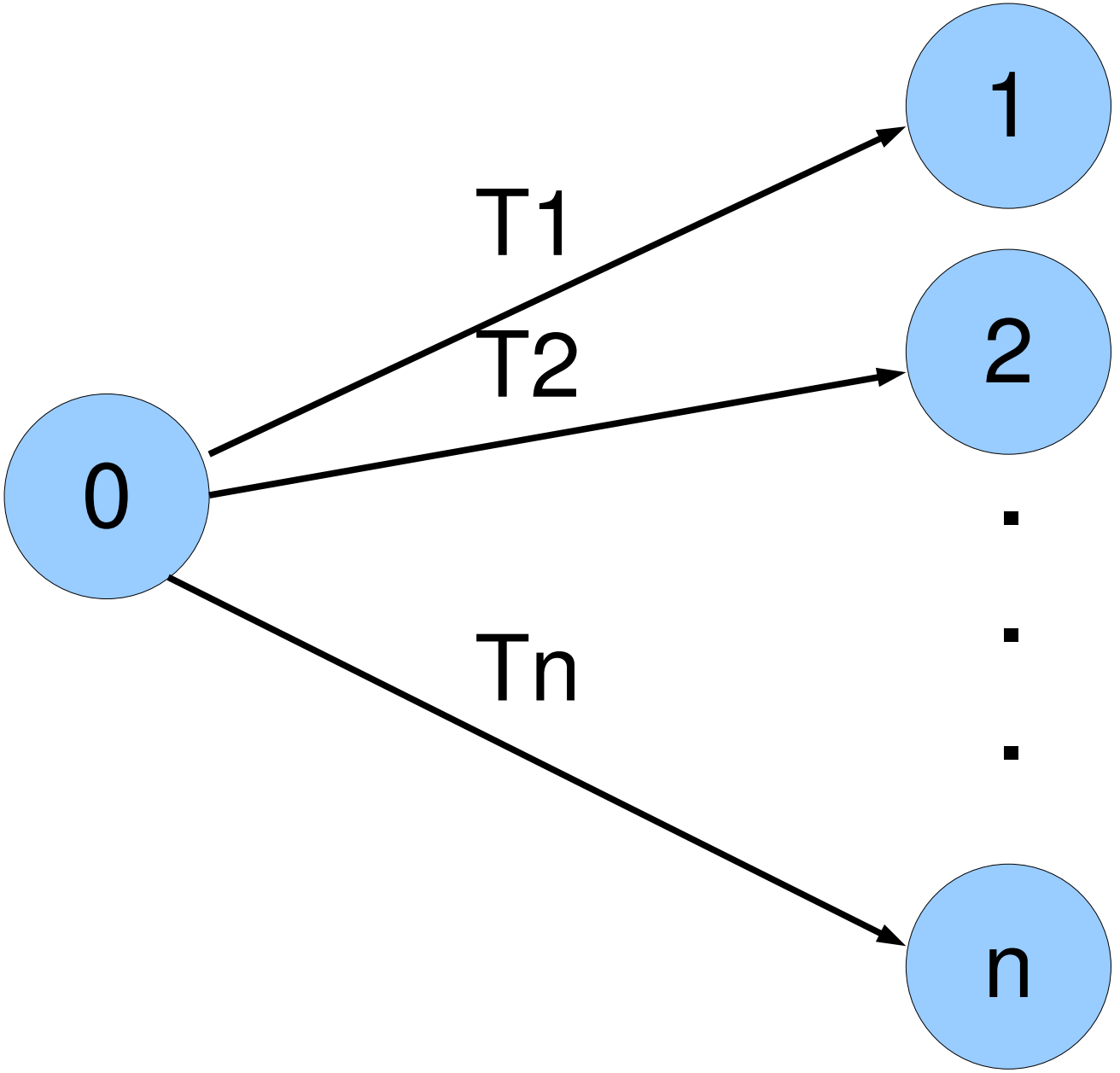
- [IN/OUT] buf e' l'indirizzo del send/receive buffer
- [IN] count e' di tipo int e contiene il numero di elementi del buffer
- [IN] dtype e' di tipo MPI_Datatype e descrive il tipo di ogni elemento del buffer
- [IN] root è di tipo int e contiene il rank del processo root della broadcast
- [IN] comm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

# MPI_Bcast

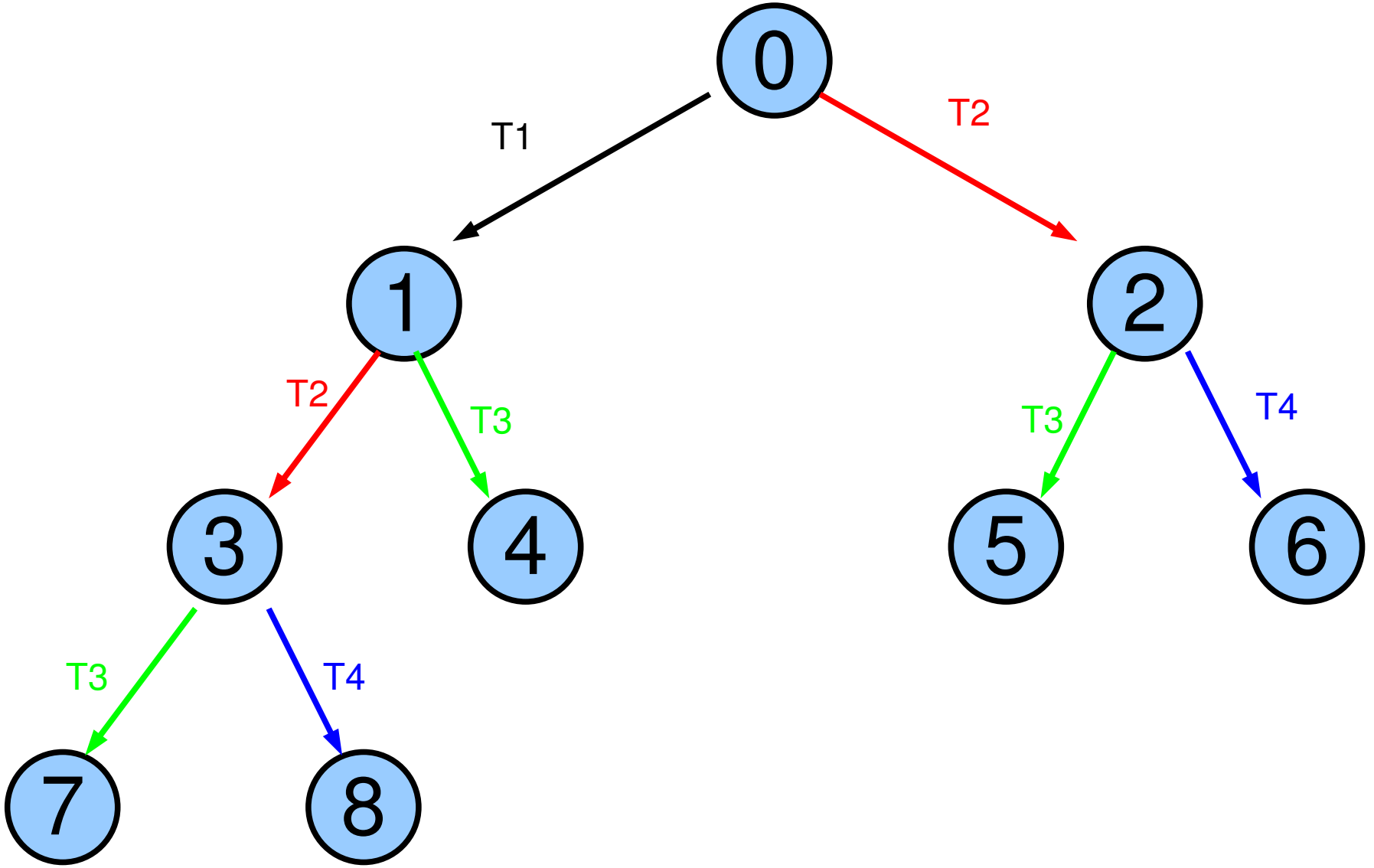
Proviamo a fare un confronto di tempi:

```
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
```

# Broadcast



# Broadcast



# Broadcast

**Esercizio (22_bcasttree):** provate ad implementare il broadcast secondo l'albero binario riportato in figura

```
$ mpiexec -n 9 ./bcast 10000000
Bcast mine time: 7.968725 s  --> 8 comunicazioni
Bcast mine tree time: 4.483855 s
Bcast time: 2.982095 s
```

# Heap

Heap operazioni elementari

```
sinistro (i)
 return 2i

desto (i)
  return 2i + 1

padre (i)
  return [i/2]
```

dove heapsize[A] ≤ n e' la lunghezza dello Heap.
([i/2] = (int) (i/2))