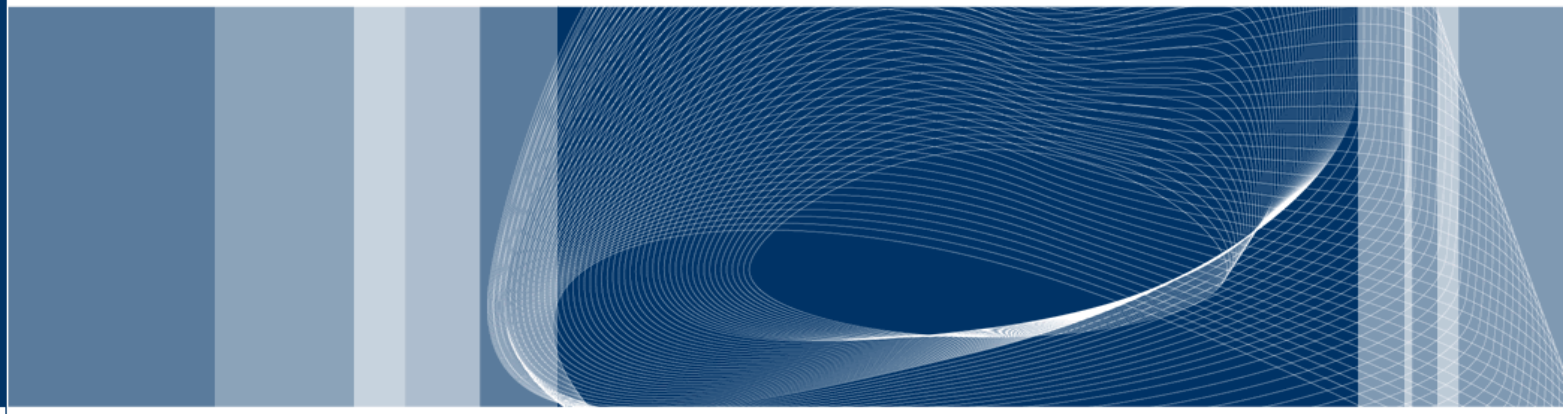
POLITECNICO DI MILANO



Introduzione a MPI

Algoritmi, Strutture Dati e Calcolo Parallelo

Daniele Loiacono

Riferimenti

- Introduction to Parallel Computing
  Blaise Barney, Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/parallel_comp/

- Anche pubblicato come Dr.Dobb's "Go Parallel"
  Introduction to Parallel Computing: Part 2
  Blaise Barney, Lawrence Livermore National Laboratory

Cos'è MPI?

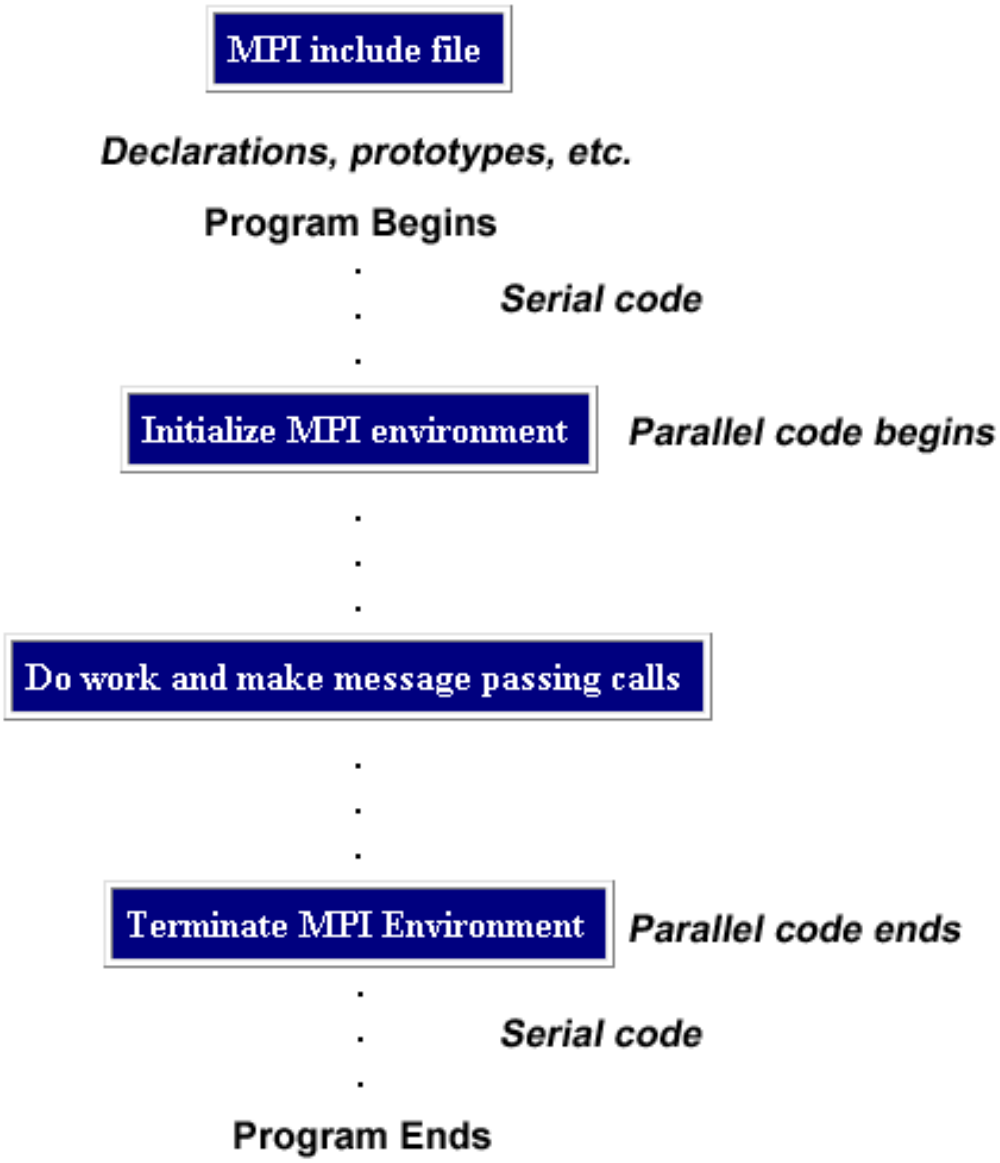
- MPI (Message Passing Interface) è uno standard sviluppato da ricercatori e industrie espressamente per il calcolo parallelo
- È soltanto una specifica e non una libreria
- Esistono diverse implementazioni C/C++ gratuite della libreria (OpenMPI, MPICH, etc.)
- Il goal di MPI è fornire un paradigma standard per la programmazione parallela che sia:
  - Pratico
  - Portabile
  - Efficiente
  - Flessibile

Outline

- Elementi base
- Gestione avanzata della comunicazione
- Comunicazioni collettive
- Comunicatori

Elementi Base

Struttura di un programma MPI



Hello world in MPI

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Compilare ed eseguire un programma MPI

- Lo standard MPI non specifica come eseguire i programmi
- Ogni implementazione fornisce appositi strumenti (programmi o script) per compilare ed eseguire un programma MPI
- Ad esempio in OpenMPI e in MPICH
  - Per compilare

    ```
    mpicc -o myprog myprog.c
    ```
  - Per eseguire un progama

    ```
    mpirun -np <N> myprog
    ```
    - Dove `-np <N>` specifica che il programma sarà composto da N processi paralleli

Scoprire informazioni sull'ambiente

- Ogni processo può usare run-time due funzioni per sapere
  - Quanti processi partecipano a questa computazione
  - Localizzarsi all'interno della computazione (cioè scoprire il suo ID)
- `MPI_Comm_size` restituisce il numero di processi della computazione:

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

- `MPI_Comm_rank` restituisce il *rank* (o ID) del processo, che è sempre compreso tra 0 e n-1 (dove n è il numero di processi che partecipano alla computazione)

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

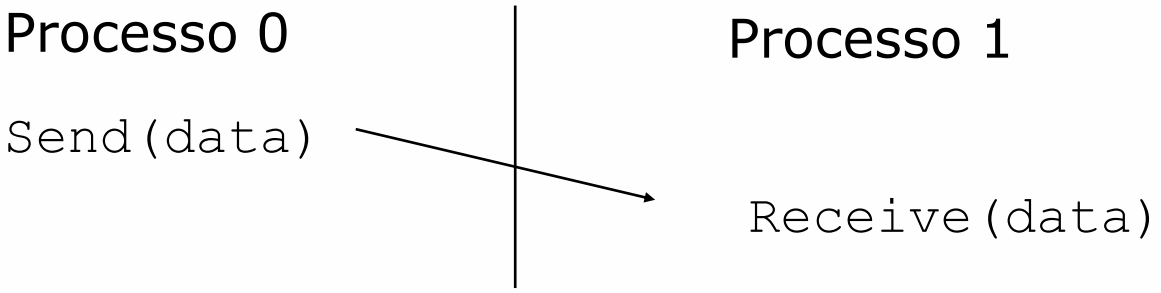
- In MPI i processi possono essere raggruppati attraverso oggetti chiamati *comunicatori.*
- `MPI_COMM_WORLD` è il comunicatore di default che comprende *tutti* i processi che partecipano alla computazione

Un nuovo Hello World!

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello world! I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Scambio di Messaggio

- In MPI la comunicazione fra i processi si basa sullo scambio di mesaggi



- Lo standard MPI definisce:
  - Come "data" deve essere rappresentato
  - Come specificare il destinatario del messaggio
  - Come deve essere implementata la comunicazione

Definizione dei dati in MPI

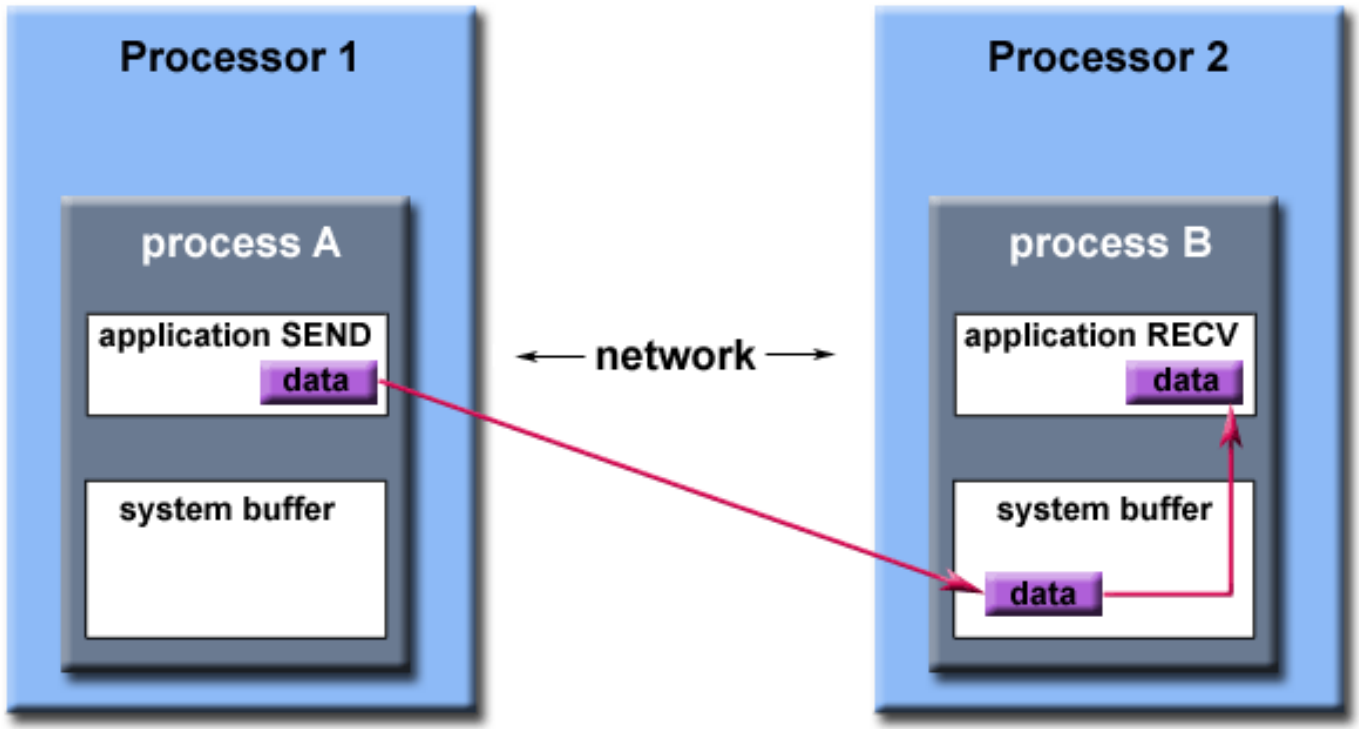
- Per inviare un dato attraverso un messaggio, occorre definire
  - *buffer:* l'indirizzo di memoria che contiene i dati
  - *count:* il numero di elementi da inviare
  - *type:* tipo di dato
- I tipi di dati usati nei messaggi sono definiti dallo standard MPI
  - Esiste un tipo di dato per ciascuno dei tipi di dato più comuni in C (e.g., `MPI_INT`, `MPI_DOUBLE_PRECISION`, `MPI_CHAR`)
  - Esistono inoltre primitive per costruire in maniera ricorsiva strutture dati più complesse (e.g., coppie di interi e float)
- La scelta di ridefinire dei tipi di dato specifici per la comunicazione garantisce la portabilità e la possibilità di scrivere facilmente applicazione che coinvolgono macchine eterogenee

Specificare i destinatari di un messaggio

- Il destinatario di un messaggio viene individuato specificando:
  - *comunicatore*
  - *tag*
  - *rank*
- Il *comunicatore* identifica uno specifico *contesto* di comunicazione a cui appartiene un *gruppo* di processi
  - Esiste un comunicatore di default `MPI_COMM_WORLD` a cui appartengono tutti i processi
- Il *tag* è un'etichetta (definita come un numero intero) che consente di differenziare ulteriormente il messaggio all'interno del comunicatore
- Il *rank* identifica in maniera precisa il processo destinatario

Gestione della comunicazione

- MPI prevede diverse modalità per gestire la comunicazione tra i processi, la modalità standard è *bloccante* e *asincrona*



MPI_Send

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

- `buf`, `count` e `datatype` specificano completamente il contenuto del messaggio
- `dest`, `tag` e `comm` specificano il destinatario del messaggio
- La funzione ritorna, quando i dati sono stai copiati nel buffer di sistema del destinatario (e quindi la memoria locale puntata da `buf` può essere riutilizzata in sicurezza)
- Quando la funzione ritorna non c'è alcuna garanzia che il processo destinatario abbia ricevuto il messaggio
- Se il buffer di sistema è pieno, la funzione resta bloccata in attesa che si liberi

MPI_Recv

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

- `buf`, `count` e `datatype` permettono di specificare dove il cotenuto del messaggio deve essere memorizzato, la sua dimensione **massima** e il suo tipo di dato
- `source` , `tag` e `comm` specificano il mittente, il tag e il comunicatore del messaggio atteso. È possibile usare le wildcard `MPI_ANY_SOURCE` e `MPI_ANY_TAG` se non si desidera specificare il rank del mittente o il tag del messaggio
- permette di ottenere ulteriore informazioni sul messaggio:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count )
```

- La funzione ritorna, quando i dati sono stai copiati dal buffer di sistema all'indirizzo di memoria specificato (e quindi i dati sono disponibili nella memoria locale all'indirizzo specificato da `buf`)

Esempio: ping

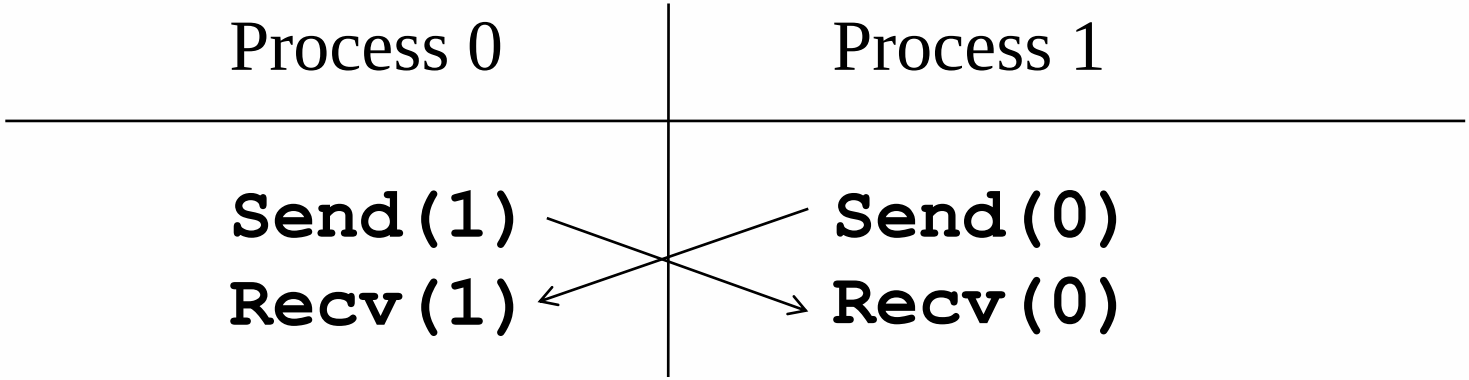
```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x'; MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  dest = 1; source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);}
else if (rank == 1) {
  dest = 0; source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);}

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank,
   count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Gestione avanzata della comunicazione

Modi di comunicazione

- Oltre alla modalità standard (asincrona) MPI definisce altre modalità di comunicazione per l'invio di un messaggio:
  - **Sincrono** (`MPI_Ssend`): il messaggio non viene copiato nel buffer di sistema e la funzione non ritorna finché il destinatario non inizia la *receive*
  - **Buffered** (`MPI_Bsend`): è analoga alla modalità standard ma permette di specificare un buffer dedicato per evitare che la *send* si possa bloccare
  - **Ready** (`MPI_Rsend`): può essere utilizzata solo quando si ha la garanzia che il destinatario ha già iniziato la corrispondente *receiv*
- Tutte le versioni elencate finora sono definite nello standard MPI come *bloccanti*
- La funzione `MPI_Recv` può essere usata per ricevere i messaggi inviati con tutte le versioni della *send*

Deadlock con funzioni bloccanti



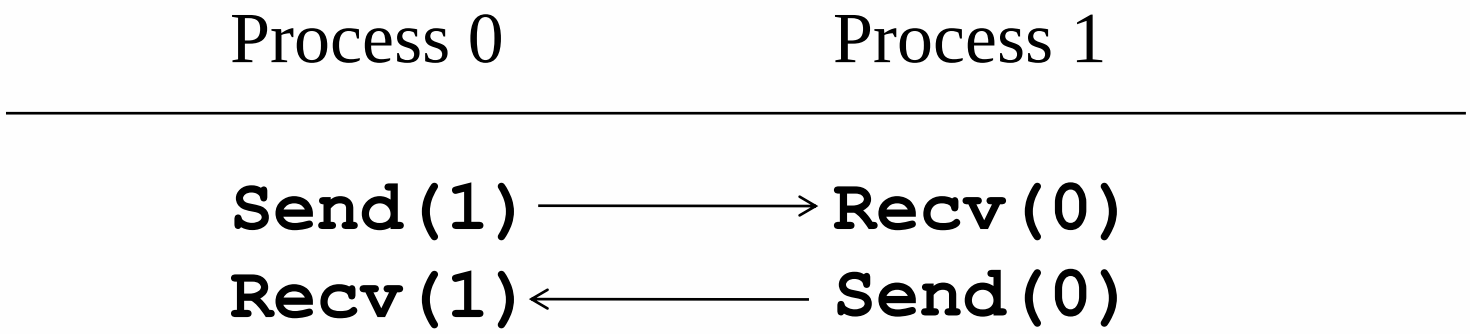
- Se le send sono sincrone si verifica sicuramente un deadlock
- Anche se le send sono asincrone si può generare un deadlock
  - Messaggi di grandi dimensioni
  - Spazio nel buffer di memoria insufficiente
- Il corretto funzionamento del programma dipende dalla disponibilità del buffer di sistema (codice *unsafe*)

Esempio Deadlock

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Ssend(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Ssend(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

Come evitare i deadlock?

- I deadlock possono essere evitati progettando in maniera corretta la comunicazione fra i processi



Esempio: evitare i deadlock

- Il processo i invia un messaggio al processo i + 1 e riceve un messaggio dal Processo i – 1 (considerando la lista dei processi circolare)

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Ssend(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                                        MPI_COMM_WORLD);
...
```

- Si crea una dipendenza circolare e abbiamo un deadlock!

Esempio: evitare i deadlock (2)

- Riprogettiamo la comunicazione per interrompere la dipenda circolare che causa il deadlock:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
   MPI_Ssend(a, 10, MPI_INT, (myrank+1)%npes, 1,MPI_COMM_WORLD);
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
   MPI_Ssend(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Comunicazioni non bloccanti

- MPI definisce anche delle funzioni di send e recv non bloccanti:

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

- Le funzioni ritornano immediatamente, la memoria all'indirizzo `buf` non può essere scritta/letta finché non si è certi della conclusione dell'operazione
- Esiste una versione non bloccante di ogni *send* (sincrona, ready, …)
- MPI mette a disposizione due funzioni bloccanti per poter aspettare la conclusione di send/receive non bloccanti (tramite l'handle request):

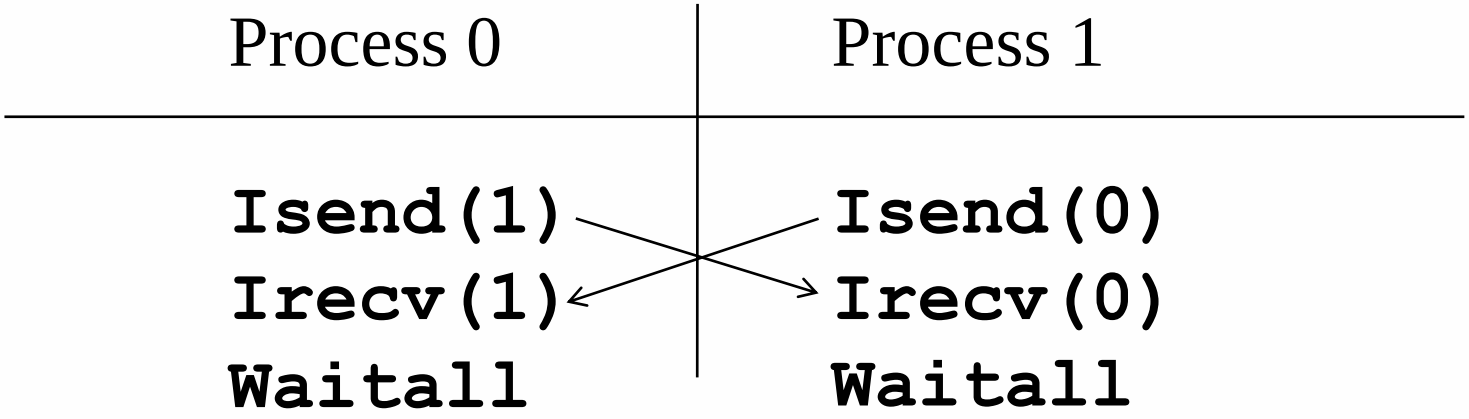
```
MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_Waitall(int count, MPI_Request *array_of_requests,
            MPI_Status *array_of_statuses)
```

Esempio comunicazioni non bloccanti

```
…
int x;
…
MPI_Request req;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
        MPI_Isend(&x, 1, MPI_INT, 1, msgtag,
                                MPI_COMM_WORLD, &req);
   compute(); // fa qualcosa
   MPI_Wait(&req, status);
} else if (myrank == 1) {
   MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Ancora sui deadlock

- Le funzioni non bloccanti consentono di evitare i deadlock



- In virtù della natura non bloccante non si verifica il deadlock

Esempio: comunicazioni non bloccanti

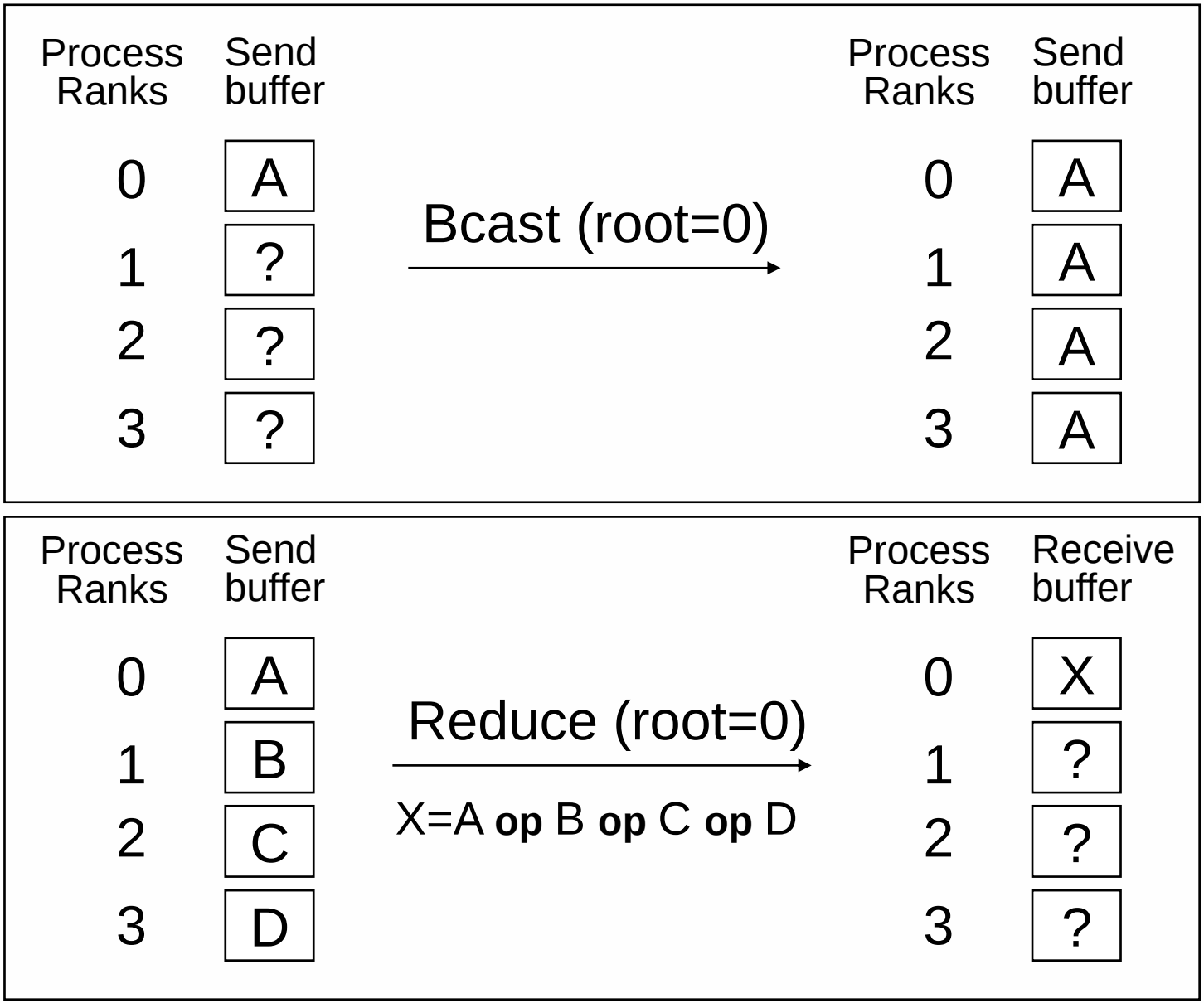
```
int main(int argc, char *argv[]) {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

/* Do some work here */
MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

Comunicazioni collettive

Comunicazioni collettive

- MPI fornisce funzioni per gestire alcune situazioni tipiche:
  - un processo *root* deve inviare un messaggio a tutti gli altri processi di un comunicatore
  - un processo *root* deve suddividere un messaggio fra tutti i processi di un comunicatore
  - tutti i processi di un comunicatore devono inviare un messaggio allo stesso processo *root*
  - tutti i processi un comunicatore devono cooperare per calcolare un risultato da inviare ad un processo *root*
- In questi casi, invece di utilizzare *send* e *receive*, è possibili ricorrere a funzioni più semplici ed efficaci
  - `MPI_Bcast`
  - `MPI_Reduce`
  - `MPI_Scatter`
  - `MPI_Gather`

Broadcast e Reduce



MPI_Bcast e MPI_Reduce

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

- Invia un messaggio dal processo con rank `root` a tutti i processi del comunicatore `comm` (incluso se stesso)
- La funzione viene invocata sia dal processo *root* sia dai destinari che devono ricevere il messaggio
- La funzione ritorna quando il messaggio è disponibile nella memoria locale all'indirizzo contenuto in `buffer`

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

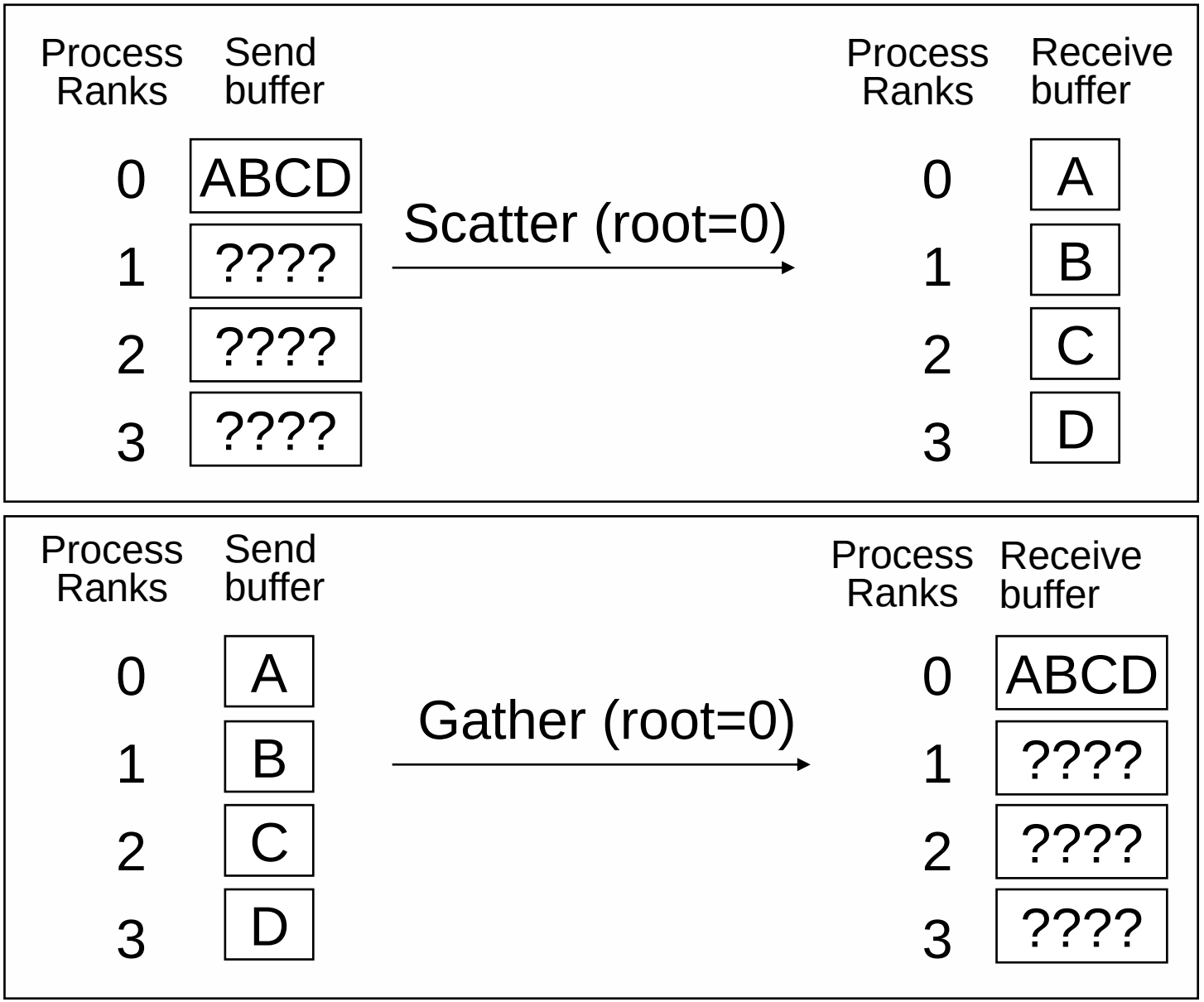
- La funzione viene invocata da tutti i processi di un comunicatore che cooperano a calcolare un risultato
- Ogni processo invia il proprio contributo che si trova in `sendbuf`
- Tutti i contributi vengono riuniti tramite l'operazione `op` e la funzione termina quando il risultato finale è disponibile nel processo *root* all'indirizzo `recvbuf`

Operazioni di riduzione

- È inoltre possibile definire nuove operazioni di riduzione attraverso la funzione `MPI_Op_create`

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Scatter e Gather



Esempio: MP_reduce e MPI_MAXLOC

```
…
struct {
   double val;
   int   rank;
} in, out;
int i, myrank, root;
double myval;
…
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
in.val = myval;
in.rank = myrank;
MPI_Reduce( in, out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
/* Nel processo root è adesso disponibile il risultato */
if (myrank == root) {
   printf("Max is %lf from process %d\n",out.val,out.rank);
…
```

MPI_Scatter e MPI_Gather

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
  sendtype, void *recvbuf, int recvcount, MPI_Datatype
  recvtype, int root,MPI_Comm comm)
```

```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
  sendtype,void *recvbuf, int recvcount, MPI_Datatype
  recvtype, int root,MPI_Comm comm)
```

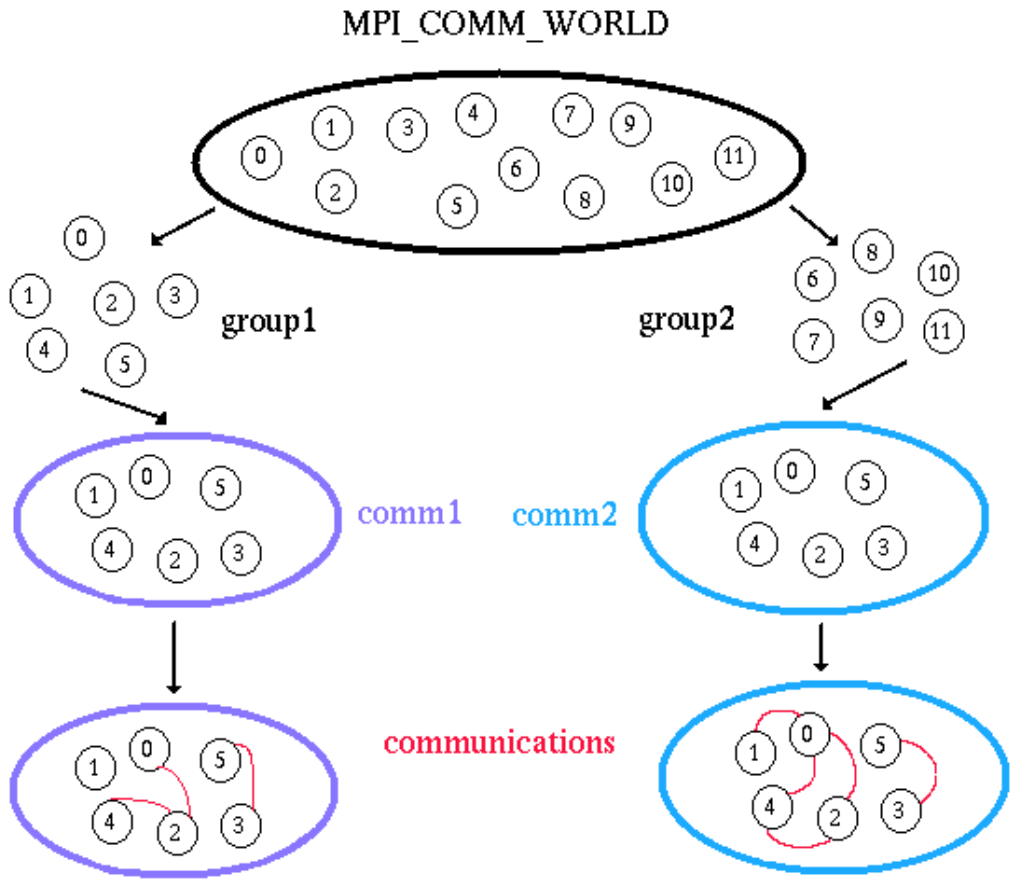
- Vengono invocate da tutti i processi nel comunicatore
- I parametri di invio nella `MPI_Scatter` e i parametri di ricezione nella `MPI_Gather` sono importanti solo nel processo *root*

Esempio: scatter di una matrice

```
int main(int argc, char *argv[])  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[4][4] = { … };
float recvbuf[4];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == 4) {
  source = 0;
  sendcount = 4;
  recvcount = 4;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
    MPI_FLOAT,source,MPI_COMM_WORLD);
  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

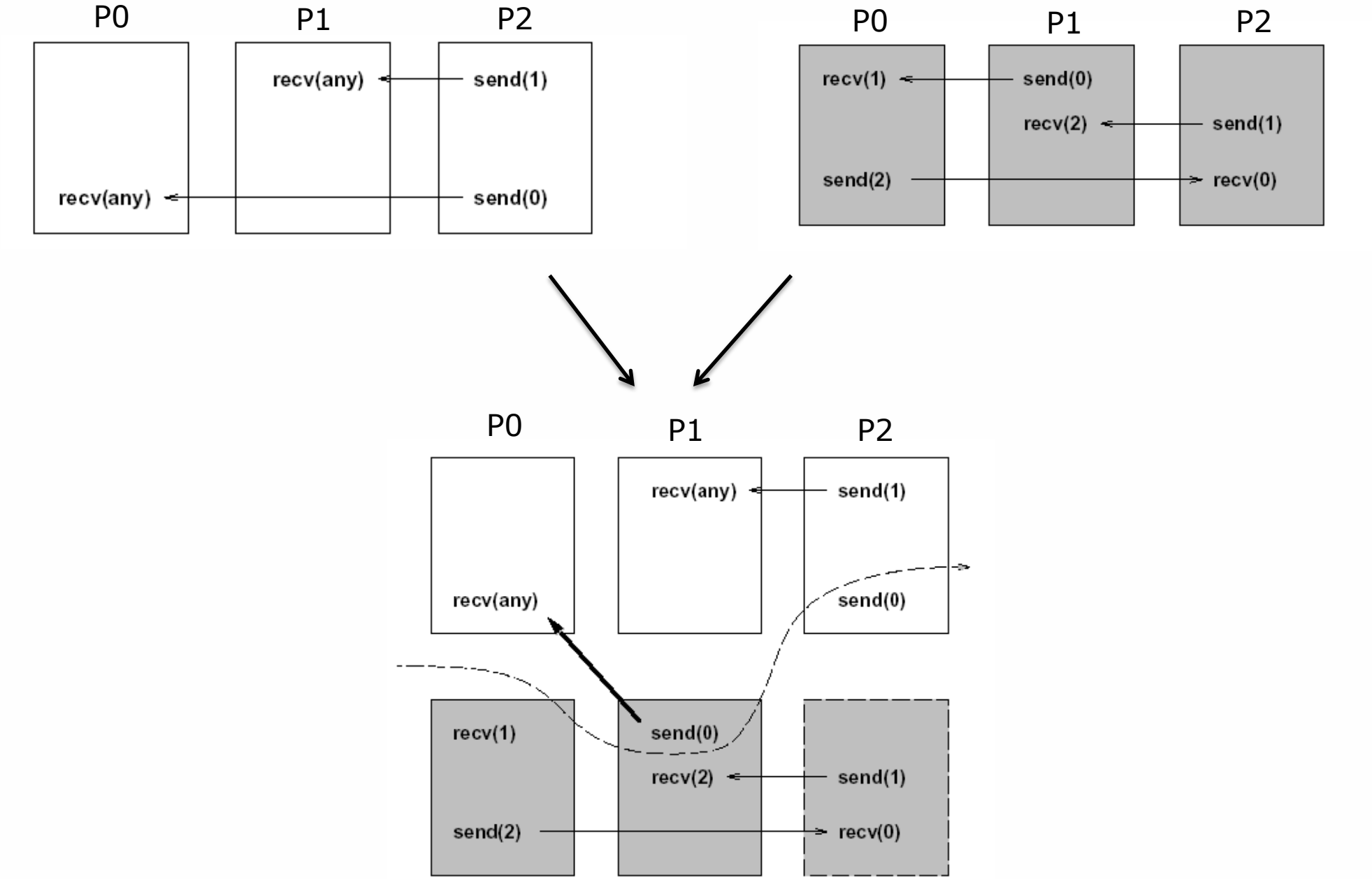
Comunicatori

Comunicatori e gruppi



- Funzioni per gestire gruppi e comunicatori
  - `MPI_Comm_create`
  - `MPI_Comm_group`
  - `MPI_Comm_split`

Comunicatori vs Tag



Comunicatori vs Tag