Send e Receive

La forma generale dei parametri di una send/receive **bloccante**

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

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

buf	Array of type type
count	Number of element of buf to be sent (non bytes)
type	MPI type of buf
dest	Rank of the destination process
tag	Number identifying the message
comm	Communicator of the sender and receiver
status	Array of size MPI_STATUS_SIZE containing communication status information

Example:
```
MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
```

La forma generale dei parametri di una send/receive **non-bloccante**

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

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

controllo

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

La funzione non ritorna il controllo finché non viene finita la ricezione del messaggio.

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

Ritorna il valore TRUE oppure FALSE se è finita o meno la ricezione del messaggio.

Wildcards

MPI_RECV accetta wildcard:

- per ricevere da ogni source: **MPI_ANYSOURCE**
- per ricevere messaggi con ogni tag: **MPI_ANY_TAG**

Modo	Condizioni per il completamento (completion)	Blocking subroutine	Non-blocking subroutine
Standard send	Messaggio spedito (lo stato di ricezione è sconosciuto)	MPI_SEND	MPI_ISEND
Standard Receive	E' completato quando un messaggio è arrivato	MPI_RECV	MPI_IRECV
Synchronous send	E' completato solo quando anche la ricezione è completata	MPI_SSEND	MPI_ISSEND
Buffered send	E completato non appena il messaggio viene messo in un buffer MPI	MPI_BSEND	MPI_IBSEND
Ready send	E' sempre completato	MPI_RSEND	MPI_IRSEND

Send e Receive Simultanee

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendDataType,
                         int dest, int sendTag, //sender
                 void *recvbuf, int recvcount, MPI_Datatype recvDataType,
                         int source, int recvTag, //ricevente
                 MPI_Comm comm., MPI_Status *status);
```

Gli argomenti includono argomenti alle funzioni send e receive.

Se vogliamo utilizzare lo stesso buffer per entrambe le send e receive, utilizziamo:

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype dataType,
                                 int dest,int sendTag, int source, int recvTag,
                                       MPI_Comm comm, MPI_Status *status);
```

Comunicazioni Collettive

Barrier

```
int MPI_Barrier(MPI_COMM comm);
```

Broadcast (uno a molti)

Tutti i processi debbono specificare lo stesso valore per **root**, **rank** e **comm**;

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

send_buffer	in/out	starting address of send buffer
send_count	in	number of elements in send buffer
send_type	in	data type of elements in send buffer
rank	in	rank of root process
comm	in	MPI communicator

Example:
```
send_count = 1;
root = 0;
MPI_Bcast ( &a, &send_count, MPI_INT, root, comm);
```

MPI_Scatter (uno a molti)

sndcount è il numero di elementi spedito ad ogni processo e non la lunghezza di sndbuf.
Gli argomenti del sender hanno significato solo per il processo root;

```
int MPI_Scatter ( void *send_buffer, int send_count, MPI_datatype send_type,
                      void* recv_buffer, int recv_count, MPI_Datatype recv_type,
                          int rank, MPI_Comm comm );
```

send_buffer	in	Starting address of send buffer
send_count	in	Number of elements in send buffer to send to each process (not the total number sent)
send_type	in	Data type of send buffer elements
recv_buffer	out	Starting address of receive buffer
recv_count	in	Number of elements in receive buffer
recv_type	in	Data type of elements in receive buffer
rank	in	Rank of receiving process
comm	in	MPI communicator

Example:
```
send_count = 1;
recv_count = 1;
send_rank = 0;
MPI_Scatter( &a, send_count, MPI_REAL,
             &a, recv_count, MPI_REAL,
             send_rank, MPI_COMM_WORLD );
```

MPI Gather (molti a uno)

Il root process riceve dati da tutti i processi nel comunicatore. È l'opposto di MPI_Scatter.
rcvcount è il numero di elementi collezionati dal root per ogni processo, non la lunghezza di rcvbuf.
Gli argomenti del receiver hanno significato solo per il processo root.

```
int MPI_Gather( void *send_buffer, int send_count, MPI_datatype send_type,
                  void* recv_buffer, int recv_count, MPI_Datatype recv_type,
                    int recv_rank, MPI_Comm comm );
```

send_buffer	in	starting address of send buffer
send_count	in	number of elements in send buffer
sent_type	in	data type of send buffer elements
recv_buffer	out	starting address of receive buffer
recv_count	in	number of elements in receive buffer for a single receive
recv_type	in	data type of elements in receive buffer
recv_rank	in	rank of receiving process
comm	in	MPI communicator

Example:
```
send_count = 1;
recv_count = 1;
recv_rank = 0;
MPI_Scatter( &a, send_count, MPI_REAL,
             &a, recv_count, MPI_REAL,
             recv_rank, MPI_COMM_WORLD );
```

AllGather (molti a molti)

MPI_Gather "raccoglie" i messaggi sul processo root. Se necessario, si potrebbe fare una broadcast per inviare il dato a tutti i processi.
In quest'ultimo caso, conviene utilizzare MPI_Allgather (stessa sintassi di MPI_Gather).

MPI_REDUCE (molti a uno)

La funzione di riduzione è generalmente utilizzata con array.

```
int MPI_Reduce( void* send_buffer, void* recv_buffer, int count,
                  MPI_Datatype data_type, MPI_Operation, int rank,
                      MPI_Comm comm );
```

send_buffer	in	address of send buffer
recv_buffer	out	address of receive buffer
count	in	number of elements in send buffer
datatype	in	data type of elements in send buffer
operation	in	reduction operation
rank	in	rank of root process
comm	in	MPI communicator

Predefined Reduction Operation Handles

Predefined operation handle	**Function**
MPI_MAX	Maxinium
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPILXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

Example:
```
MPI_Reduce( sbuf, rbuf, 6, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
```

MPI_ALLREDUCE (molti a molti)

Esegue una operazione di riduzione comune attraverso tutti i processi in un gruppo e piazza il risultato in tutti i processi.

Operazione MPI	**Funzione**
MPI_MAX	Massimo
MPI_MIN	Minimo
MPI_SUM	Somma
MPI_PROD	Prodotto
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Example:
```
count = 1;
MPI_Allreduce( sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
```

MPI Datatypes

MPI_Type_struct

```
struct buff_layout
{
      int i_val[3];
      int d_val[5];
} buffer;
```

```
array_of_types[0] = MPI_INT;
array_of_blocklengths[0] = 3;
array_of_deplacements[0] = 0;
array_of_types[1] = MPI_DOUBLE;
array_of_blocklengths[1] = 5;
array_of_deplacements[1] = …;
```

```
MPI_Type_struct(2, array_of_blocklengths, array_of_deplacements,
                  array_of_types, &buff_datatype);

MPI_Type_commit(&buff_datatype);

MPI_Send(&buffer, 1 , buff_datatype, ……);
```

MPI_Type_contiguous (rappresentante una riga di un array)

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Example:

```
MPI_Datatype rowtype;
MPI_Type_contiguous(SIZE, MPI_FLOAT, & rowtype); //SIZE = 4;
MPI_Type_commit(& rowtype);
MPI_Type_free(& rowtype); //de-allocazione tipo nuovo;

MPI_Send(&a[2][0], 1, rowtype, … …); // a[4][4];
MPI_Recv(b, SIZE, MPI_FLOAT, … …); // float b[SIZE];
```

MPI_Type_vector (rappresentante una colonna di un array)

```
int MPI_Type_vector(int count, int blocklength, int stride, //=num pass tra bloc
                        MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Example:

```
MPI_Datatype columntype;
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, & columntype); //SIZE = 4;
MPI_Type_commit(& columntype);
MPI_Type_free(& columntype);

MPI_Send(&a[0][1], 1, columntype, … …); // a[4][4];
MPI_Recv(b, SIZE, MPI_FLOAT, … …); // float b[SIZE];
```

MPI_Type_indexed (rappresentante elementi arbitrari di una array)

```
MPI_Type_indexed(int count, int* blocklengths, int* displacements,
                        MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Example:

```
int count = 2;
blocklengths[0]= 4; displacements[0]= 5;
blocklengths[1]= 2; displacements[1]= 12;

MPI_Datatype indextype;
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, & indextype);

MPI_Send(&a, 1, indextype, … …);
MPI_Recv(b, NUM_ELEMNTS, MPI_FLOAT, … …); //NUM_ELEMNTS = 6;
```

PACK / UNPACK (paccare/spaccare dati in locazioni contigue o non)

Example:

```
float* a_ptr, b_ptr;
int* n_ptr;
char buffer[100];
int position;


//pack
position = 0;

MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
//position has been incremented;
MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);

MPI_Bcast(buffer, 100, MPI_PACKED, … …); //broadcast;

//unpack
position = 0;
MPI_Unpack(buffer, 100, &position, a_ptr, 1, MPI_FLOAT, MPI_COMM_WORLD);
MPI_Unpack(buffer, 100, &position, b_ptr, 1, MPI_FLOAT, MPI_COMM_WORLD);
MPI_Unpack(buffer, 100, &position, n_ptr, 1, MPI_INT, MPI_COMM_WORLD);
```

Comunicatori e Gruppi

MPI_Comm_group (estrazione del gruppo da un comunicatore)

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
```

MPI_Group_incl (creazione di un nuovo gruppo)

```
MPI_Group_incl(MPI_Group old_group, int count, int* members,
                  MPI_Group *new_group);
```

Variable Name	Type	in/out	Description
old_group	Group	in	Handle del gruppo vecchio
count	int	in	Numeri di processi del gruppo
members	int*	in	Array di dimensioni count dei ranghi dei processi "vecchi" da includere in new_group
new_group	Group	out	Handle del gruppo nuovo

MPI_Group_excl (creazione di un nuovo gruppo)

```
MPI_Group_incl(MPI_Group old_group, int count, int* nonmembers,
                  MPI_Group *new_group);
```

Variable Name	Type	in/out	Description
old_group	Group	in	Handle del gruppo vecchio
count	int	in	Numeri di processi non membri del nuovo gruppo
nonmembers	int*	in	Array di dimensioni count dei ranghi dei processi da escludere in new_group
new_group	Group	out	Handle del gruppo nuovo

Partizionamento (partiz. di un comm in sotto gruppi)

```
Int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

Partiziona il comunicatore comm. in tanti sottogruppi secondo **color**.
Se **key** di un processo è più piccolo del **key** di un altro, il rango del primo processo nel comunicatore sarà inferiore del secondo.
Altrimenti MPI assegnerà arbitrariamente il loro rango.

Example:

```
MPI_Group orig_group, new_group;
MPI_Comm new_comm;

MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
MPI_Group_incl(orig_group, count, ranks, &new_group);
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank(new_group, &new_rank); //ottengo il rango
MPI_Comm_rank(new_group, &new_rank); //ottengo il rango
```

Topologie Virtuali

Creazione Topologia Cartesiana

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                    int reoder, MPI_Comm *comm_cart);
```

comm_old	Rappresenta il vecchio comunicatore
comm_cart	Rappresenta il nuovo comunicatore (cartesiano)
ndims	Numero di dimensioni della topologia
dims	Array che rappresenta la lunghezza di ogni dimensione
periods	Rappresenta la toroidalità ([i]=1 se i è toroidale)
reoders	Se false, il rango di ogni processo del nuovo comunicatore è identico a quello vecchio

Partizionamento topologie virtuali

```
int MPI_Cart_sub(MPI_Comm comm_cart, int* keep_dims, MPI_Comm *comm_subcart);
```

Se keep_dims[i] è vero allora la dimensione i-esima è mantenuta nella nuova sotto-topologia.
Le coordinate di un processo nella sotto-topologia si ottengono semplicemente dalle coordinate nella topologia originaria, scartando le coordinate "non mantenute".

Uso Topologie cartesiane

1) Permettono la conversione coordinate-rango e rango-coordinate.

```
int MPI_Cart_rank(MPI_Comm comm_cart, int* coords, int* rank);

int MPI_Cart_coord(MPI_Comm_cart, int rank, int maxdims, int *cords);
```

2) shift lungo un dimensione.

```
int MPI_Cart_shift(MPI_comm comm_cart, int dir, int s_step,
                        int* rank_source,int* rank_dest);
```

comm_cart	Handle del comunicatore
dir	Direzione o coordinate dello SHIFT (integer): dir= 0 -> shift lungo "x"; dir= 1 -> shift lungo "y";
s_step	Displacement (integer): s_step= 1 -> dest= vicino a "destra"; s_step= -1 -> dest= vicino a "sinistra"; s_step= 2 …
rank_source	Rango del processo mittente (integer)
rank_dest	Rango del processo destinatario (integer)