



**23<sup>rd</sup> Summer  
School on  
PARALLEL  
COMPUTING**

# Introduction to MPI Part II Collective Communications and communicators

Andrew Emerson, Fabio Affinito {a.emerson,f.affinito}@cineca.it  
SuperComputing Applications and Innovation Department





# Collective communications



- Collective communications will not interfere with point-to-point
- **All** processes (in a communicator) call the collective function
- All collective communications are blocking (in MPI 2.0)
- No tags are required
- Receive buffers must match in size (number of bytes)
- Can only be used with MPI predefined datatypes -not with MPI Derived datatypes

*It's a safe communication mode*



Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group) and are of three types:

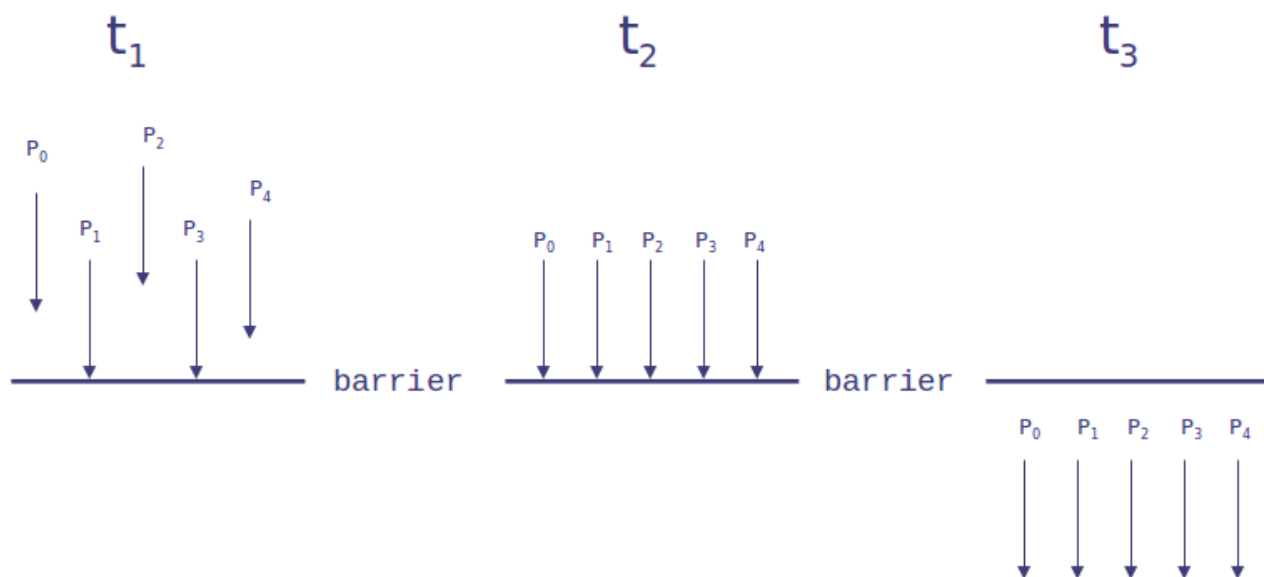
- Synchronization (e.g. Barrier)
- Data Movement (e.g. Broadcast or Gather/scatter)
- Global Computation (e.g. reductions)



## MPI Barrier

It stops all processes within a communicator until they are synchronized

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

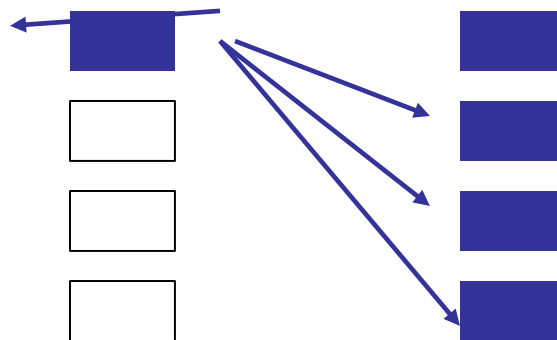




## MPI Broadcast

*Int MPI\_Bcast (void \*buf, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm)*

Note that all processes must specify the same root and same comm.





```
PROGRAM broad_cast
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, root
  INTEGER status(MPI_STATUS_SIZE)
  REAL A(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  root = 0
  IF( myid .EQ. 0 ) THEN
    a(1) = 2.0
    a(2) = 4.0
  END IF
  CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
  WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
  CALL MPI_FINALIZE(ierr)
```



## MPI Gather

Each process, root included, sends the content of its send buffer to the root process. The root process receives the messages and stores them in the rank order.

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



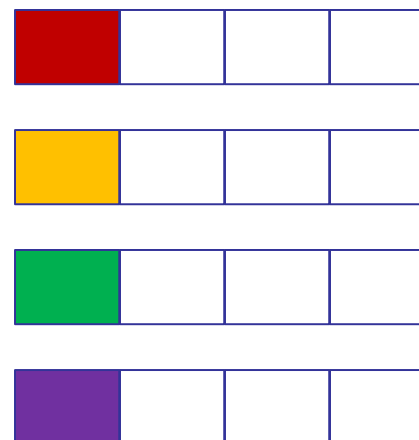




## MPI Scatter

The root sends a message. The message is split into  $n$  equal segments, the  $i$ -th segment is sent to the  $i$ -th process in the group and each process receives this message.

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





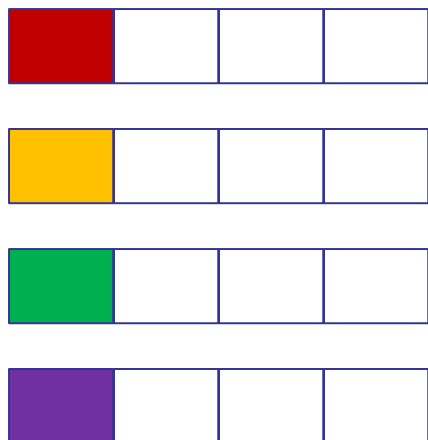
There are possible combinations of collective functions.

For example,

### **MPI Allgather**

is a combination of a gather + a broadcast

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
void *recvbuf, int recvcount, MPI_Datatype recvtpe,  
MPI_Comm comm)
```



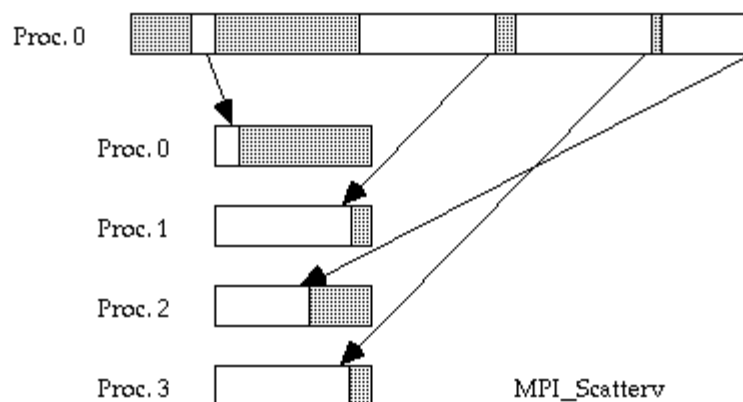


For many collective functions there are extended functionalities.

*For example it's possible to define the length of arrays to be scattered or gathered with*

**MPI\_Scatterv**

**MPI\_Gatherv**

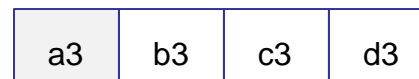




## MPI All to all

This function makes a redistribution of the content of each process in a way that each process know the buffer of all others. It is a way to implement the matrix data transposition.

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





## Reduction

Reduction operations permits us to

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root process (MPI\_Reduce) or
- Store the result on all processes (MPI\_Allreduce)



## Predefined reduction operations

MPI op	Function
MPI_MAX	Maximum
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
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location



```
PROGRAM scatter
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, nsnd, i
  REAL A(16), B(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  root = 0
  IF( myid .eq. root ) THEN
    DO i = 1, 16
      a(i) = REAL(i)
    END DO
  END IF
  nsnd = 2
  CALL MPI_SCATTER(a, nsnd, MPI_REAL, b, nsnd, &
    & MPI_REAL, root, MPI_COMM_WORLD, ierr)
  WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)
  CALL MPI_FINALIZE(ierr)
END
```



```
PROGRAM gather
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, nsnd, i
  REAL A(16), B(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  root = 0
  b(1) = REAL( myid )
  b(2) = REAL( myid )
  nsnd = 2
  CALL MPI_GATHER(b, nsnd, MPI_REAL, a, nsnd,
    & MPI_REAL, root, MPI_COMM_WORLD, ierr)
  IF( myid .eq. root ) THEN
    DO i = 1, (nsnd*nproc)
      WRITE(6,*) myid, ': a(i)=', a(i)
    END DO
  END IF
  CALL MPI_FINALIZE(ierr)
END
```





```
PROGRAM reduce
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, root
  REAL A(2), res(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  root = 0
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
    MPI_COMM_WORLD, ierr)
  IF( myid .EQ. 0 ) THEN
    WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
  END IF
  CALL MPI_FINALIZE(ierr)
END
```



## Performance issues

- Much hidden communication takes place with collective communication.
- Hardware vendors work hard to provide optimized collective calls but performances will vary according to implementation.
- Because of forced synchronization, collective communications may not always be the best solution.



## MPI communicators and groups



Many users are familiar with the mostly used communicator:

## **MPI\_COMM\_WORLD**

A **communicator** can be thought as a handle to a **group**.

- a group is a ordered set of processes
  - each process is associated with a rank
  - ranks are contiguous and start from zero

Groups allow collective operations to be operated on a subset of processes



Typical usage:

1. Extract handle of global group from MPI\_COMM\_WORLD using MPI\_Comm\_group
2. Form new group as a subset of global group using MPI\_Group\_incl
3. Create new communicator for new group using MPI\_Comm\_create
4. Determine new rank in new communicator using MPI\_Comm\_rank
5. Conduct communications using any MPI message passing routine
6. When finished, free up new communicator and group (optional) using MPI\_Comm\_free and MPI\_Group\_free



## **Intracommunicators**

are used for communications within a single group

## **Intercommunicators**

are used for communications between two disjoint groups



## Group management:

- All group operations are local
- Groups are not initially associated with communicators
- Groups can only be used for message passing within a communicator
- We can access groups, construct groups, destroy groups



## Group accessors:

### **-MPI\_GROUP\_SIZE**

This routine returns the number of processes in the group

### **-MPI\_GROUP\_RANK**

This routine returns the rank of the calling process inside a given group





## Group constructors

Group constructors are used to create new groups from existing ones (initially from the group associated with `MPI_COMM_WORLD`; you can use `mpi_comm_group` to get this).

Group creation is a local operation: no communication is needed

After the creation of a group, no communicator has been associated to this group, and hence no communication is possible within the new group



**-MPI\_COMM\_GROUP(comm,group,ierr)**

This routine returns the group associated with the communicator comm

**-MPI\_GROUP\_UNION(group\_a, group\_b, newgroup, ierr)**

This returns the ensemble union of group\_a and group\_b

**-MPI\_GROUP\_INTERSECTION(group\_a, group\_b, newgroup, ierr)**

This returns the ensemble intersection of group\_a and group\_b

**-MPI\_GROUP\_DIFFERENCE(group\_a, group\_b, newgroup, ierr)**

This returns in newgroup all processes in group\_a that are not in group\_b, ordered as in group\_a



**-MPI\_GROUP\_INCL**(group, n, ranks, newgroup, ierr)

This routine creates a new group that consists of all the n processes with ranks ranks[0]... ranks[n-1]

*Example:*

group = {a,b,c,d,e,f,g,h,i,j}

n = 5

ranks = {0,3,8,6,2}

newgroup = {a,d,i,g,c}



**-MPI\_GROUP\_EXCL**(group,n,ranks,newgroup,ierr)

This routine returns a newgroup that consists of all the processes in the group after removing processes with ranks: ranks[0]..ranks[n-1]

*Example:*

group = {a,b,c,d,e,f,g,h,i,j}

n = 5

ranks = {0,3,8,6,2}

newgroup = {b,e,f,h,j}



## Communicator management

Communicator access operations are local, not requiring interprocess communication

Communicator constructors are collective and may require interprocess communications

We will cover in depth only intracommunicators, giving only some notions about intercommunicators.



## Communicator accessors

**-MPI\_COMM\_SIZE(comm,size,ierr)**

Returns the number of processes in the group associated with the comm

**-MPI\_COMM\_RANK(comm,rank,ierr)**

Returns the rank of the calling process within the group associated with the comm

**-MPI\_COMM\_COMPARE(comm1,comm2,result,ierr)**

Returns:

- MPI\_IDENT if comm1 and comm2 are the same handle
- MPI\_CONGRUENT if comm1 and comm2 have the same group attribute
- MPI\_SIMILAR if the groups associated with comm1 and comm2 have the same members but in different rank order
- MPI\_UNEQUAL otherwise



## Communicator constructors

**-MPI\_COMM\_DUP**(comm, newcomm, ierr)

This returns a communicator newcomm identical to the communicator comm

**-MPI\_COMM\_CREATE**(comm, group, newcomm, ierr)

This collective routine must be called by all the process involved in the group associated with comm. It returns a new communicator that is associated with the group. MPI\_COMM\_NULL is returned to processes not in the group.

Note that group must be a subset of the group associated with comm!



## A practical example:

```
CALL MPI_COMM_RANK (...)
```

```
CALL MPI_COMM_SIZE (...)
```

```
CALL MPI_COMM_GROUP (MPI_COMM_WORLD,wgroup,ierr)
```

define something..

```
CALL MPI_COMM_GROUP_EXCL(wgroup....., newgroup...)
```

```
CALL MPI_COMM_CREATE(MPI_COMM_WORLD,newgroup,newcomm,ierr)
```





**-MPI\_COMM\_SPLIT(comm, color, key, newcomm, ierr)**

This routine creates as many new groups and communicators as there are distinct values of color.

The rankings in the new groups are determined by the value of the key.

MPI\_UNDEFINED is used as the color for processes to not be included in any of the new groups



Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	1	1	3	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Both process a and j are returned `MPI_COMM_NULL`

3 new groups are created

{i, c, d}

{k, b, e, g, h}

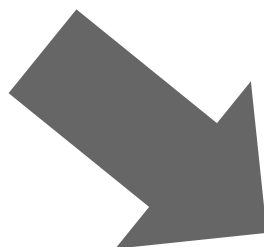
{f}



MPI provides functions to manage and to create **groups** and **communicators**.

**MPI\_comm\_split**, for example, creates a communicator...

```
if(myid%2==0){  
    color=1;  
}else{  
    color=2;  
}  
MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,&subcomm);  
MPI_COMM_RANK(subcomm,mynewid);  
printf("rank in MPICOMM_WORLD %d",myid,"rank in Subcomm %d",mynewid);
```



I am rank 2 in MPI\_COMM\_WORLD, but 1 in Comm 1.  
I am rank 7 in MPI\_COMM\_WORLD, but 3 in Comm 2.  
I am rank 0 in MPI\_COMM\_WORLD, but 0 in Comm 1.  
I am rank 4 in MPI\_COMM\_WORLD, but 2 in Comm 1.  
I am rank 6 in MPI\_COMM\_WORLD, but 3 in Comm 1.  
I am rank 3 in MPI\_COMM\_WORLD, but 1 in Comm 2.  
I am rank 5 in MPI\_COMM\_WORLD, but 2 in Comm 2.  
I am rank 1 in MPI\_COMM\_WORLD, but 0 in Comm 2.



## Destructors

*The communicators and groups from a process' viewpoint are just handles.*

Like all handles, there is a limited number available: you could (in principle) run out!

**-MPI\_GROUP\_FREE(group, ierr)**

**-MPI\_COMM\_FREE(comm,ierr)**



## Intercommunicators

Intercommunicators are associated with 2 groups of disjoint processes.

Intercommunicators are associated with a remote group and a local group

The target process (destination for send, source for receive) is its rank in the remote group.

A communicator is either intra or inter, never both



## Practical info

Yes, ok, but how can I write the right functions?

<http://www.mpi-forum.org/docs/mpi-2.2>




A.2. C BINDINGS

531

A.2 C Bindings

A.2.1 Point-to-Point Communication C Bindings

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

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

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

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

`int MPI_Cancel(MPI_Request *request)`

`int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

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

`int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,  
MPI_Status *status)`

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

`int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,`

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23



## From C bindings to Fortran bindings

- In Fortran all function are transformed in subroutines and they don't return a type
- All functions have an additional argument (ierror) of type integer
- All MPI datatypes in Fortran are defined as integers

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

46  
47  
48

```
21 MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)  
22 <type> BUF(*)  
23 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR  
24
```





Now we can seriously start to work...



BLD010844 [RF] © www.visualphotos.com



# TITOLO SLIDE

Testo slide