Introduction to MPI Part II Collective Communications and communicators

Gabriele Fatigati – {g.fatigati}@cineca.it
SuperComputing Applications and Innovation Department

Collective communications

Collective communications is a method of communication which involves all processes in a communicator:

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

It's a safe communication mode

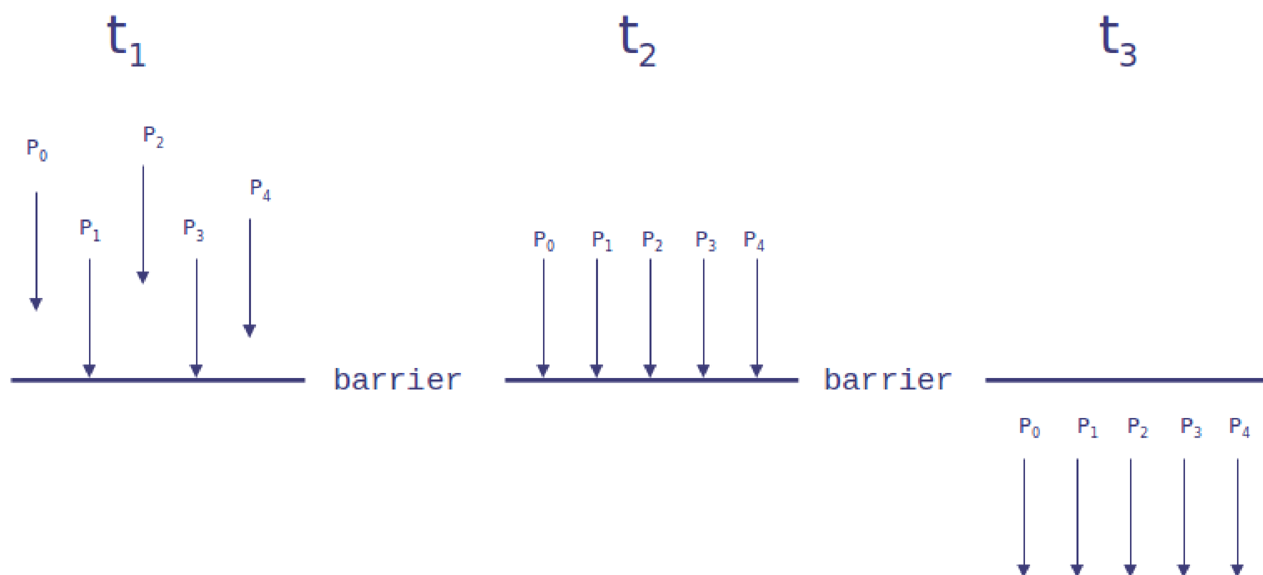
Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group) and are of three types:

- 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

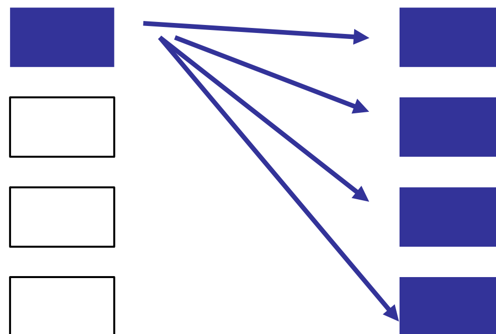
int MPI_Barrier(MPI_Comm comm);



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.



Summer School on PARALLEL COMPUTING

Example

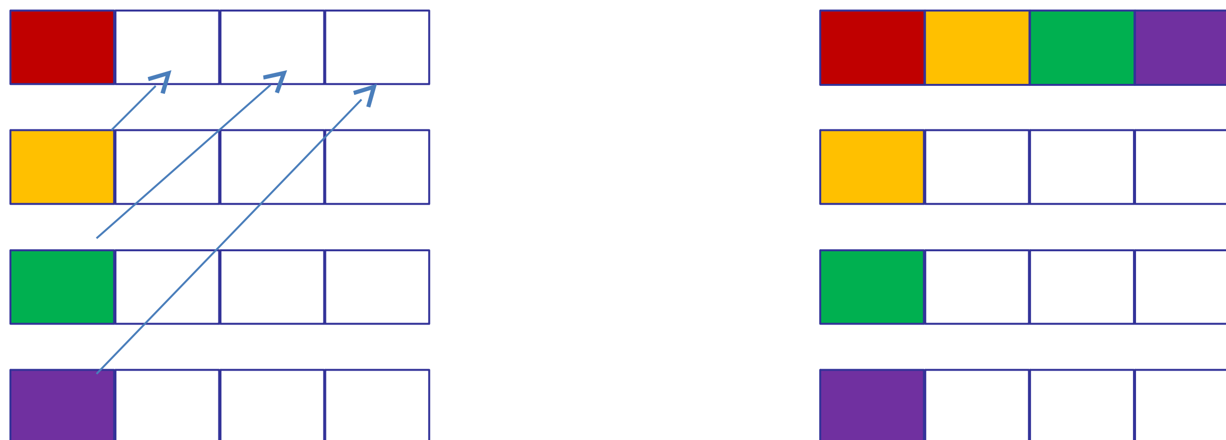
```
PROGRAM broad_cast
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF( myid .EQ. 0 ) THEN
    a(1) = 2.0
    a(2) = 4.0
END IF
CALL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

CINECA

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. *recvcnt* parameter is the count of elements received per process, not the total summation of counts from all processes.

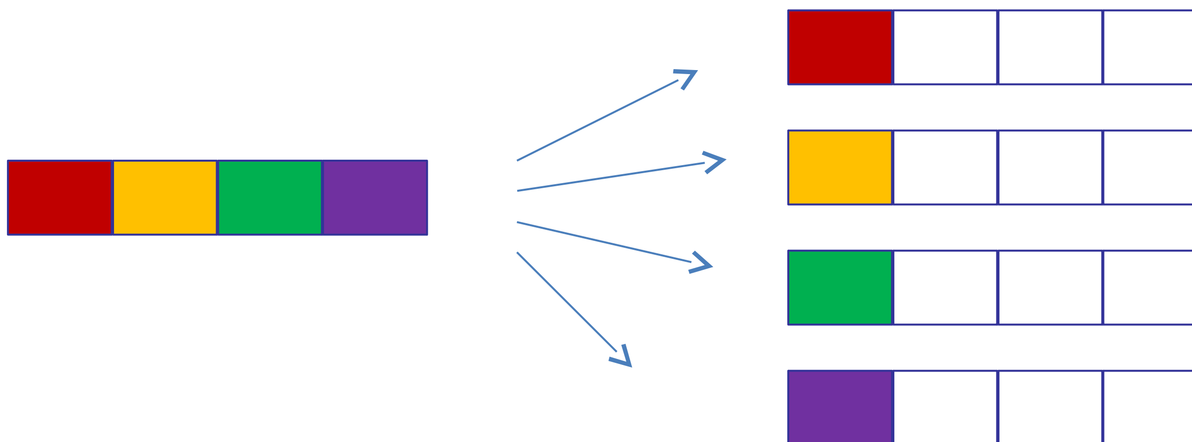
*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,*
*void *recvbuf, int recvcnt, MPI_Datatype recvtype,*
int root, MPI_Comm comm)



MPI Scatter

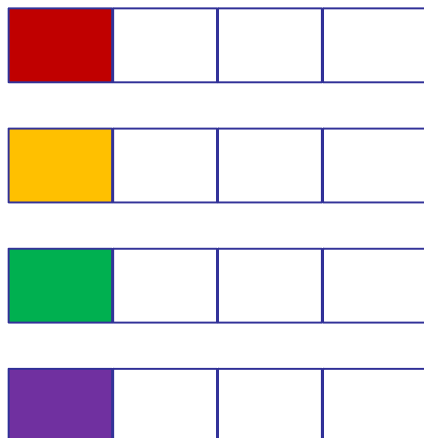
The root sends a message. The message is split into n equal segments, the i-th segment is sent to the i-th process in the group and each process receives this message.

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



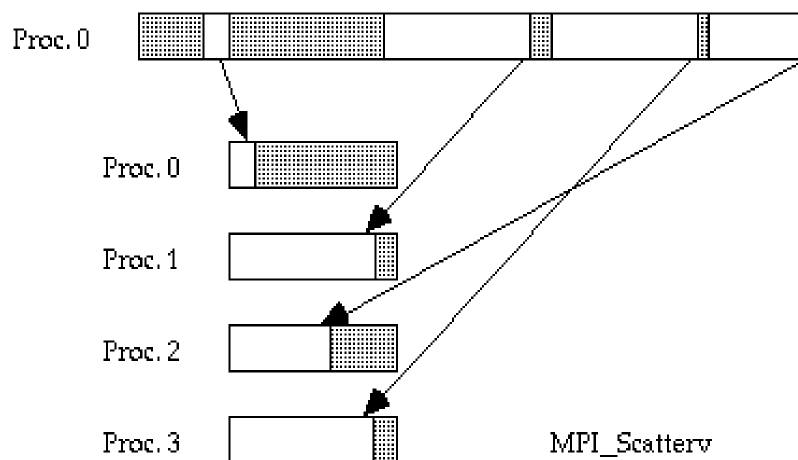
There are possible combinations of collective functions.
For example,
MPI Allgather
is a combination of a gather + a broadcast

*int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, 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. **Sendbuf, sendcounts significant only at root.**

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs,
                 MPI_Datatype sendtype, void *recvbuf, int recvcount,
                 MPI_Datatype recvtype,  int root, MPI_Comm comm)
```



MPI_Gatherv

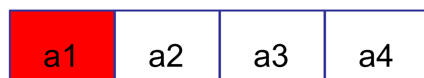
Gathers into specified locations from all processes in a group. **Recvbuf, Recvcounts significant only at root.**

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, const int *recvcounts, const int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

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 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
```

```
0 : res(1)=   4.000000     res(2)=   8.000000
```

Performance issues

- 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.

Some studies show that around 80% transfer time is in collectives.

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

The group routines are primarily used to specify which processes should be used to construct a communicator.

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 (no communication is needed)

- 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, i.e. groups/communicators are dynamic - they can be created and destroyed during program execution.

Using MPI Groups

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. When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

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

Group Creation

MPI_COMM_WORLD

group 1

0 2 3 1

group 2

4 5 7 6

comm1

comm2

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

- **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 rare 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.

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: Indicates that the underlying groups have identical members in the same rank order. These communicators differ only by context.
- 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!

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char **argv) {
    int rank, new_rank, nprocs, sendbuf, recvbuf, ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm new_comm;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
    if (rank < nprocs/2)
        MPI_Group_incl(orig_group, nprocs/2, ranks1, &new_group);
    else
       MPI_Group_incl(orig_group, nprocs/2, ranks2, &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);
    printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
    MPI_Finalize();
    return 0;
```

```
rank= 2 newrank= 2 recvbuf= 6
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 7 newrank= 3 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 6 newrank= 2 recvbuf= 22
```

- **MPI_COMM_SPLIT**(comm, color, key, newcomm, ierr)

This routine creates as many new groups and communicators as there are distinct values of color.
(processes in the same color are in the same communicator).

The **rank**ings 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}

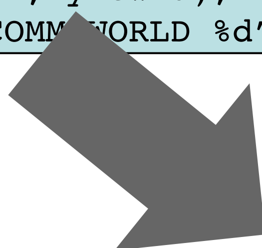
{f}

Drop U, count number of colors, for each colors select keys in order.

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,color);
```



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.

int MPI_Comm_split(*MPI_Comm old_comm, int color, int key, MPI_Comm *new_comm*)

For a 2D logical grid, create subgrids of rows and columns

```
c**logical 2D topology with nrow rows and mcol columns
irow = Iam/mcol !! logical row number
jcol = mod(Iam, mcol) !! logical column number
comm2D = MPI_COMM_WORLD
call MPI_Comm_split(comm2D, irow, jcol, row_comm, ierr)
call MPI_Comm_split(comm2D, jcol, irow, col_comm, ierr)
```

Figure a.
2D logical Grid

Figure b.
3 Row Subgrids

Figure c.
2 Column Subgrids

Iam	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1

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