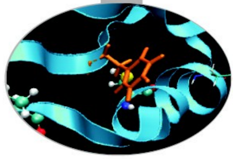
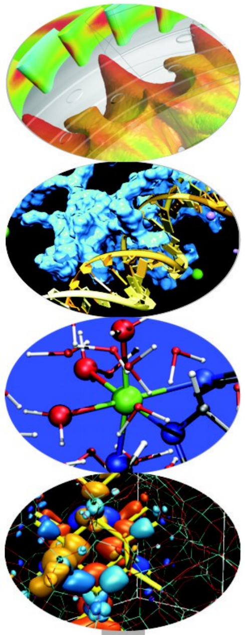


INTRODUCTION TO MPI – COLLECTIVE COMMUNICATIONS AND COMMUNICATORS

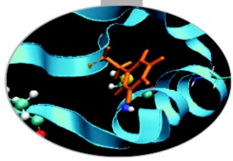


Introduction to Parallel Computing with MPI and OpenMP

18-19-20 november 2013

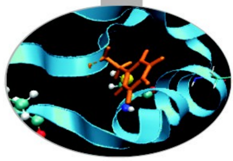
a.marani@cineca.it
f.affinito@cineca.it

SuperComputing Applications and Innovation



Part I:
Collective communications

SuperComputing Applications and Innovation



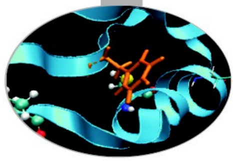
WHAT ARE COLLECTIVE COMMUNICATIONS?

Communications involving a group of processes

They are called by all the ranks involved in a communicator (or a group)

Collectives can be divided in three types:

- Synchronization collectives
- Message passing collectives
- Reduction collectives



PROPERTIES OF COLLECTIVE COMMUNICATIONS

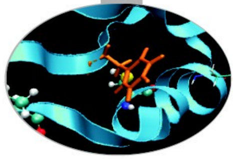
- Collective communications will not interfere with point-to-point
- Easier to read and to implement in a code
- All processes (in a communicator) call the collective function
- All collective communications are blocking (not true from MPI 3.0)
- No tags are required
- Receive buffers must match in size (number of bytes)

’s a safe communication mode!!

A SMALL EXAMPLE

Write a program that initializes an array of two elements as (2.0,4.0) only on task 0, and than sends it to all the other tasks

How can you do that with the knowledge you got so far?



POINT-TO-POINT SOLUTION

```
PROGRAM broad_cast_p2p
INCLUDE ’mpif.h’
INTEGER ierr, myid, nproc, root, i
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)
IF( myid .EQ. 0 ) THEN
    a(1) = 2.0
    a(2) = 4.0
END IF
IF( myid .EQ. 0 ) THEN
   DO i=1,nproc-1
         CALL MPI_ISEND(a,2,MPI_REAL,i,0,&
                   MPI_COMM_WORLD,ierr)
   ENDDO
ELSE
   CALL MPI_RECV(a,2,MPI_REAL,0,0,&
             MPI_COMM_WORLD,status,ierr)
ENDIF
WRITE(6,*) myid, ’: a(1)=’, a(1), ’a(2)=’, a(2)
CALL MPI_FINALIZE(ierr)
END PROGRAM
```

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv) {
  int myid, nproc, root, i;
  MPI_Status status;
  float a[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,
&nproc);

MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if ( myid ==0 ) {
     a[0] = 2.0;
     a[1] = 4.0;
     }
  if ( myid == 0 ) then {
     for (i=1;i<nproc;i++)
         MPI_Isend(a,2,MPI_FLOAT,i,0,
                  MPI_COMM_WORLD);
      }
  else {
     MPI_Recv(a,2,MPI_FLOAT,0,0,
              MPI_COMM_WORLD,&status);
     }
  printf(“%d : a[0]=, %f, a[1]=,
%f\n”,myid,a[0],a[1]);
  MPI_Finalize();
  return 0;
```

MPI BROADCAST

C *:*

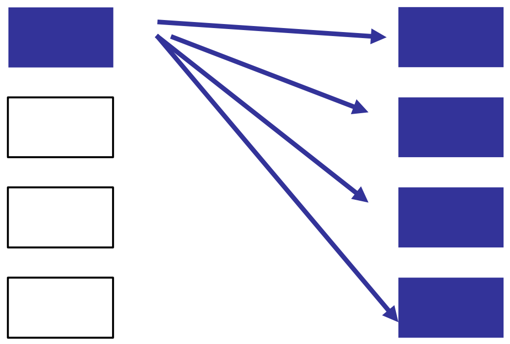
*int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

FORTRAN *:*

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Root process sends the buffer to all other processes with just one command!

Note that all processes must specify the same root and the same communicator

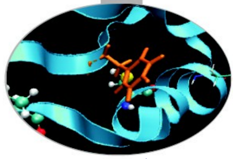


COLLECTIVE SOLUTION

```
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)

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
```

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

int main (int argc, char **argv) {
  int myid, nproc, root, i;
  MPI_Status status;
  float a[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,
&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  if ( myid ==0 ) {
     a[0] = 2.0;
     a[1] = 4.0;
     }

  MPI_Bcast (a,2,MPI_FLOAT,0,
          MPI_COMM_WORLD);
  printf("%d : a[0]=, %f, a[1]=,
%f\n",myid,a[0],a[1]);
  MPI_Finalize();
  return 0;
}
```

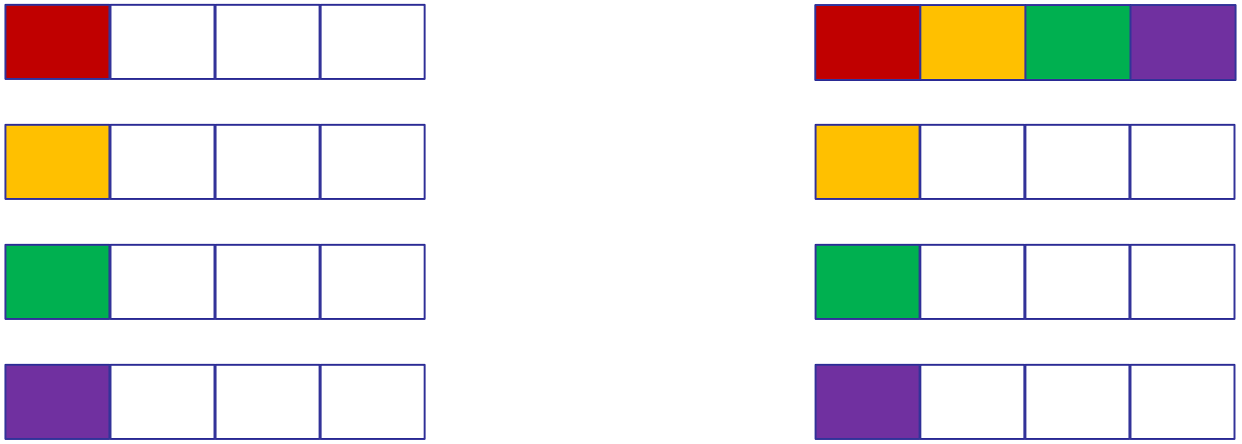
MPI GATHER



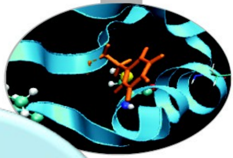
C :
*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)*

FORTRAN :
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

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.

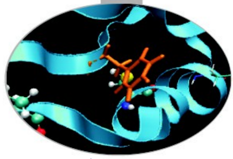


GATHER EXAMPLE (C)



```
#include <mpif.h>
#include <stdio.h>

int main (int argc, char** argv) {
     int myid, nproc, count, i;
     float A[16], B[2];
     MPI_Init(ierr);
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);

     b[0] = (float) myid;
     b[1] = (float) myid;
    count = 2;
    MPI_Gather(b, count, MPI_FLOAT, a, count, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if ( myid == 0 ) {
      for (i=0; i<count*nproc; i++)
           printf("%d : a[%d]=%f \n", myid, i, a[i]");
    }

    MPI_Finalize();
    return 0;
}
```

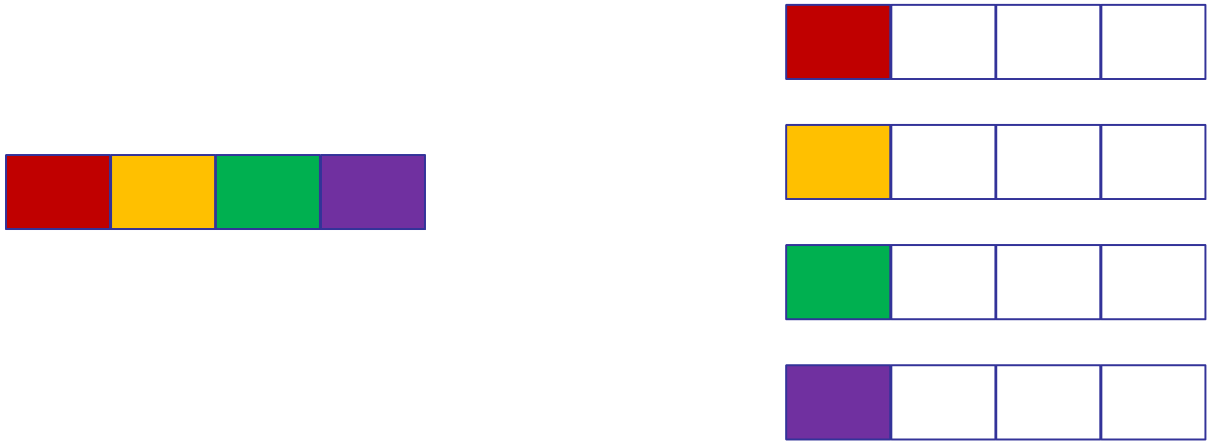
MPI SCATTER



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

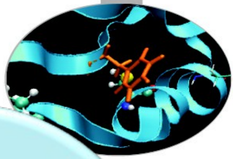
Fortran :
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

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.

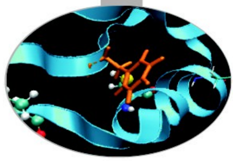


SCATTER EXAMPLE (FORTRAN)

```
PROGRAM scatter
INCLUDE 'mpif.h'

INTEGER ierr, myid, nproc, count, 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)

IF( myid .eq. 0 ) THEN
DO i = 1, 16
a(i) = REAL(i)
END DO
END IF

count = 2
CALL MPI_SCATTER(a, count, MPI_REAL, b, count, MPI_REAL, root, &
MPI_COMM_WORLD, ierr)
WRITE(6,*) myid, ': b(1)=', b(1), 'b(2)=', b(2)

CALL MPI_FINALIZE(ierr)
END
```



SCATTERV & GATHERV



What if the message that has to be scattered/gathered should not be split equally among processes?

C :
*int MPI_Scatterv(void *sendbuf, int *sendcnt, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)*

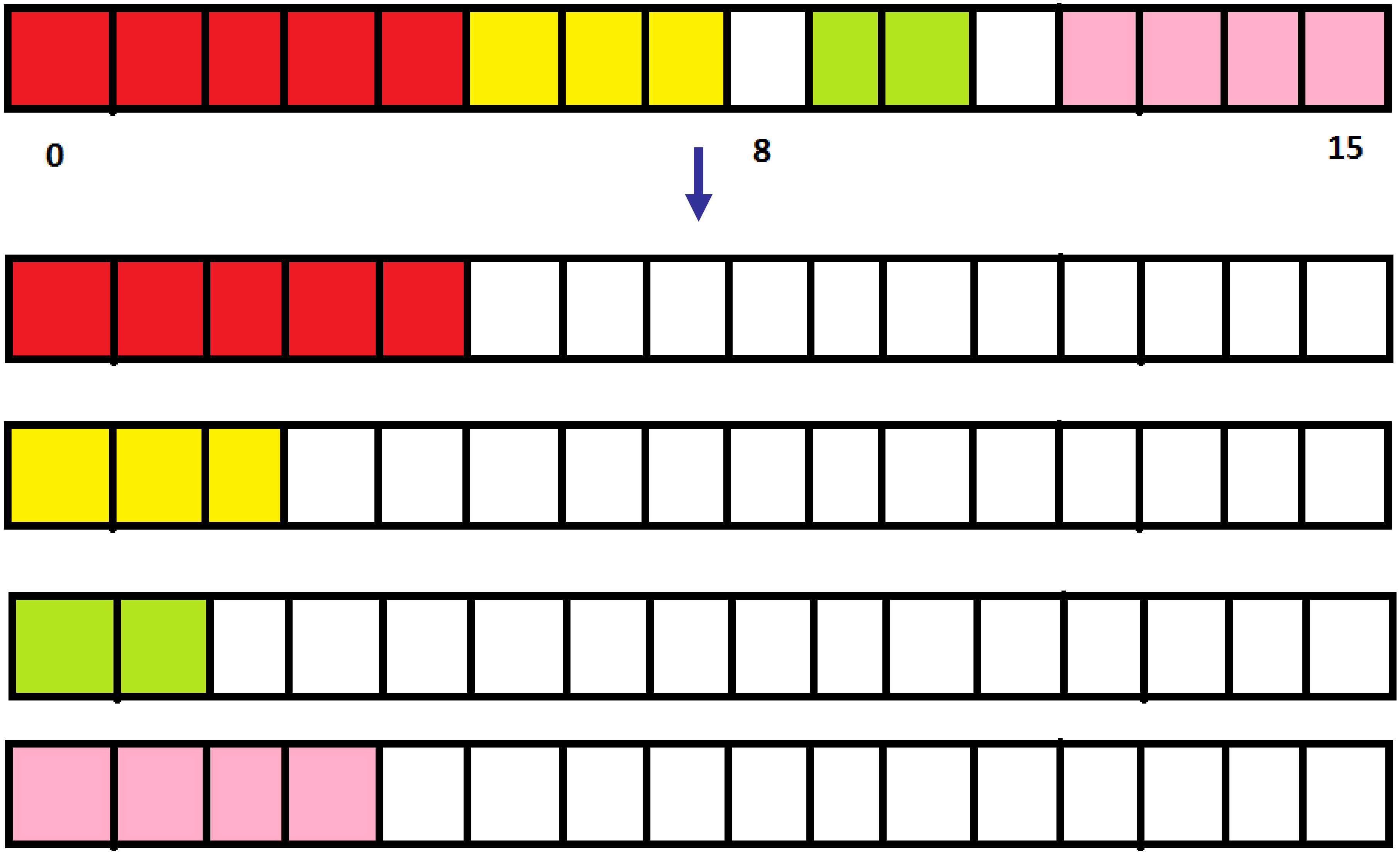
Fortran :
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, IERROR

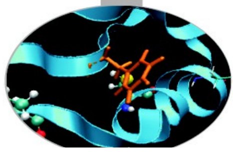
sendcounts/recvcounts is an array of integers stating how many elements should be considered for each process

displs is an array of integers stating the position of the starting element for each process

SCATTERV & GATHERV

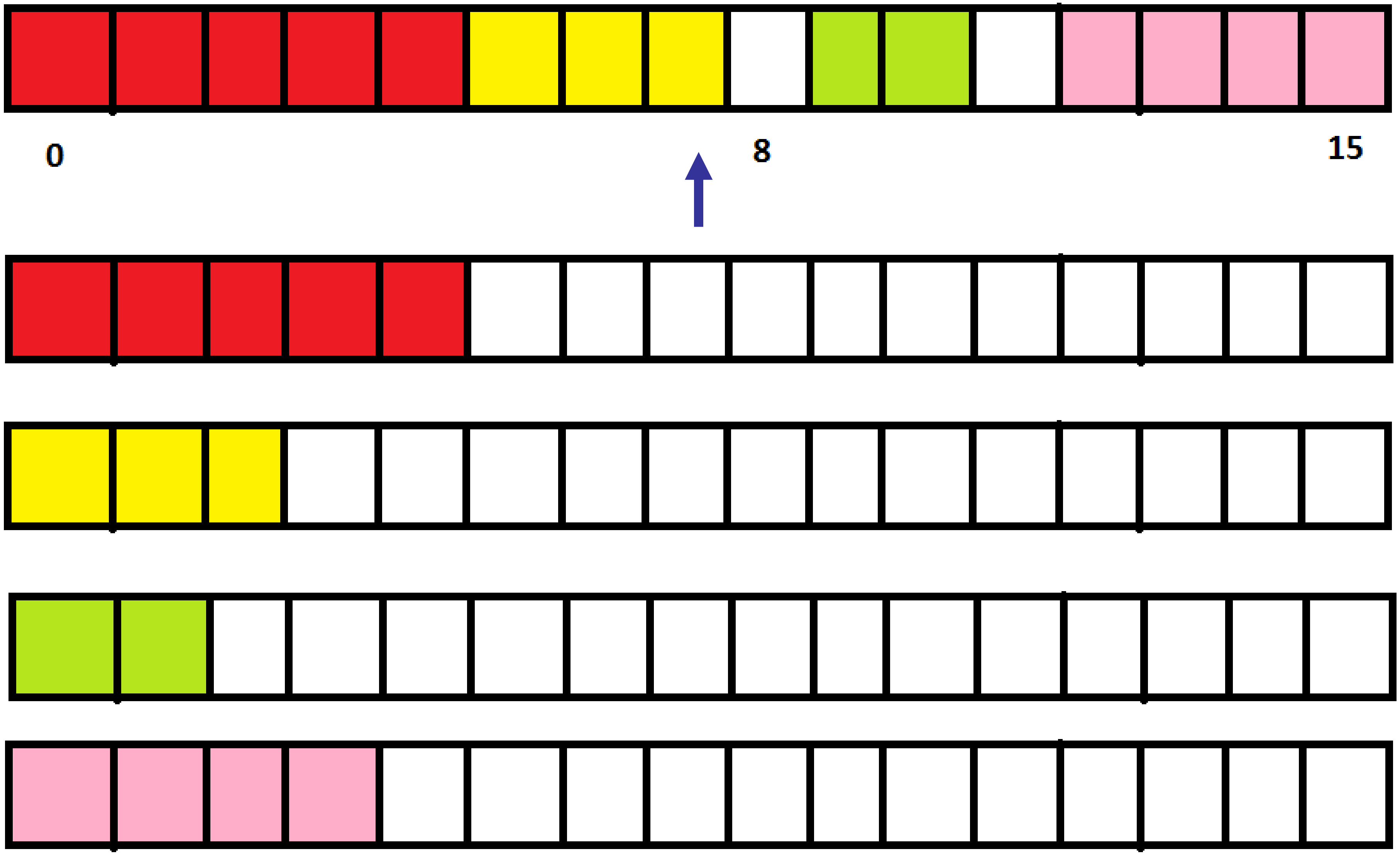
SCATTERV sendcounts=(5,3,2,4) displs=(0,5,9,12)





SCATTERV & GATHERV

GATHERV   recvcounts=(5,3,2,4) displs=(0,5,9,12)



COLLECTIVE COMBINATIONS

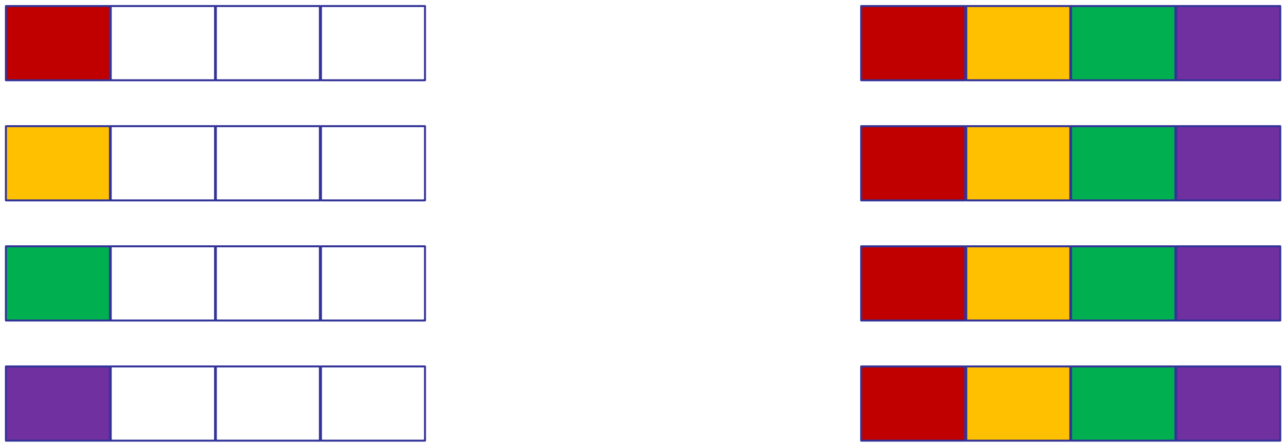
There are functions that combine the effects of two collective functions!
For example, **MPI Allgather** is a combination of a gather + a broadcast

C :

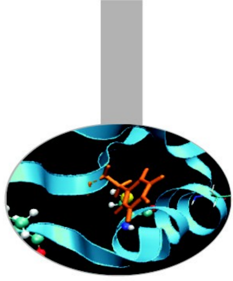
*int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

Fortran :

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVTYPE, COMM, IERROR



MPI ALLTOALL



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

C *:*

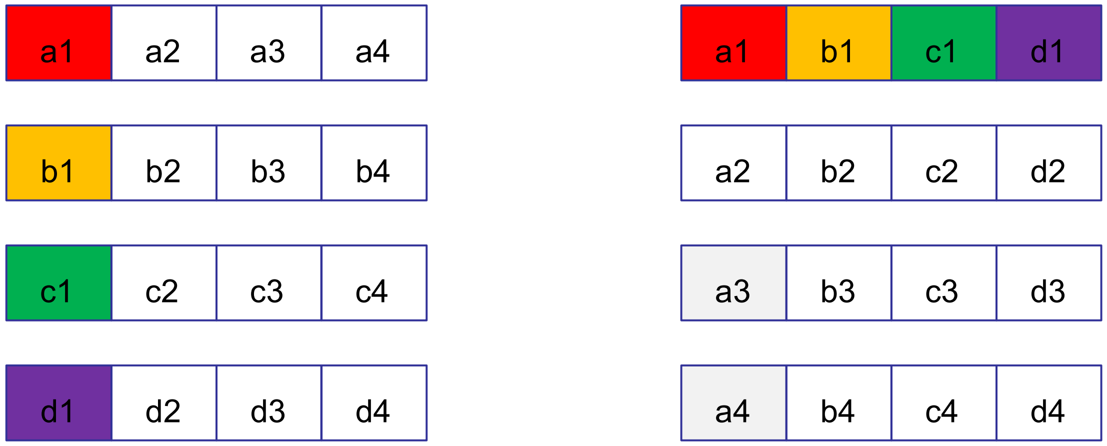
*int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

FORTRAN *:*

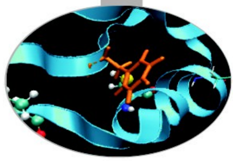
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR



REDUCTION OPERATIONS



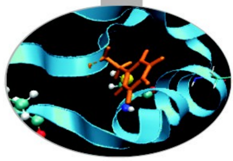
Reduction operations permit to:

- 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)

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

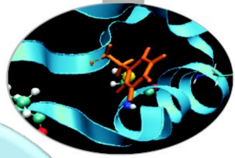
***FORTRAN** :*
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

LIST OF REDUCTIONS



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

EXAMPLE: SUM REDUCTION (FORTRAN)

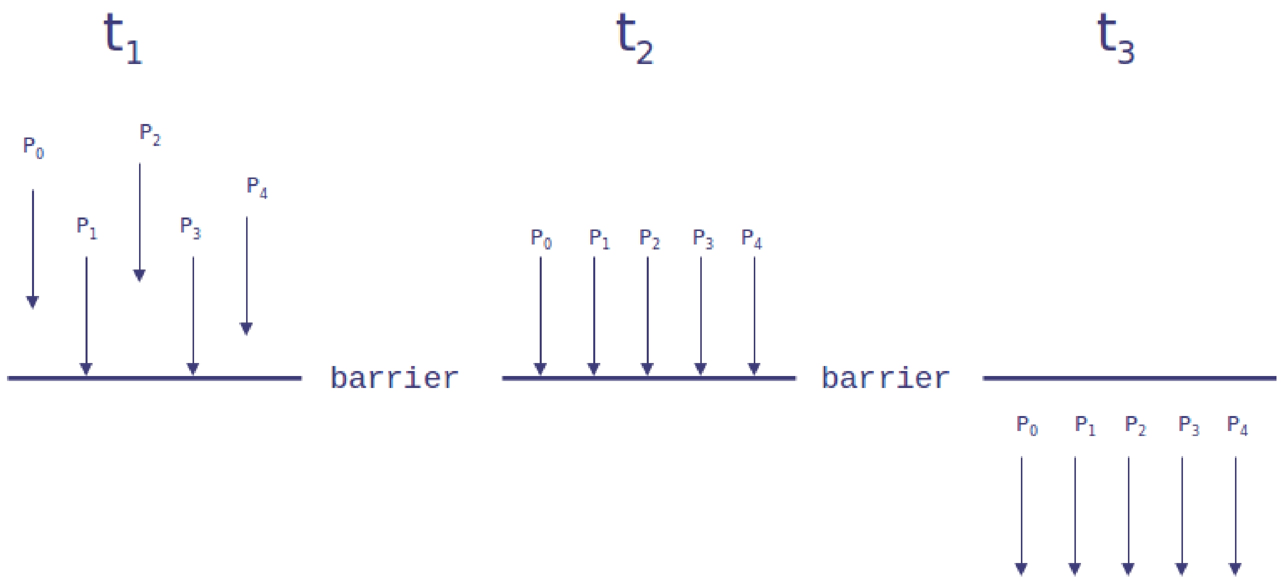


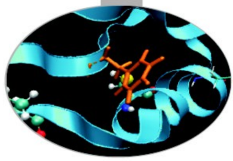
```
PROGRAM reduce
INCLUDE ’mpif.h’
INTEGER ierr, myid, nproc, root
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
a(1) = 2.0
a(2) = 4.0
CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root, &
MPI_COMM_WORLD,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

SYNCHRONIZATION COMMAND: MPI BARRIER

It stops all processes within a communicator until they are synchronized

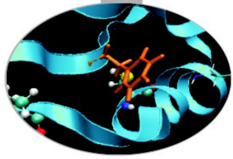
int MPI_Barrier(MPI_Comm comm);
CALL MPI_BARRIER(COMM,IERROR)





Part II:
MPI communicators and groups

WHAT ARE COMMUNICATORS?



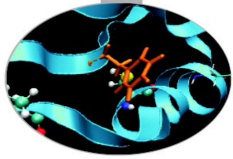
Many users are familiar with the mostly used communicator:
MPI_COMM_WORLD

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

DEFINITIONS & PROPERTIES



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:

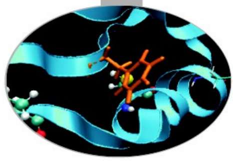
- **MPI_GROUP_SIZE**

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

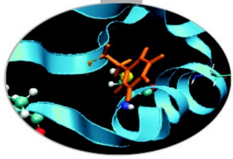


Group constructors are used to create new groups from existing ones (initially from the group associated with MPI_COMM_WORLD; you can use mpi_comm_group to get this).

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

GROUP CONSTRUCTORS



- **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

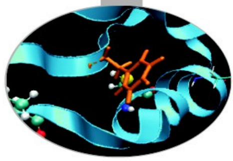
- **MPI_GROUP_INTERSECTION**(group_a, group_b, newgroup, ierr)

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

GROUP CONSTRUCTORS



- **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]

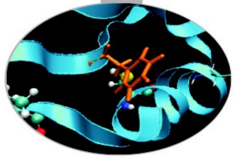
Example:
group = {a,b,c,d,e,f,g,h,i,j}
n = 5
ranks = {0,3,8,6,2}
newgroup = {a,d,i,g,c}

- **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

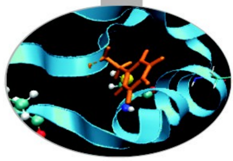


Communicator access operations are local, not requiring interprocess communication

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)

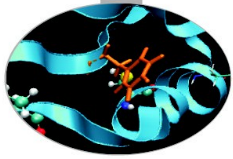
Returns the rank of the calling process within the group associated with the comm

- **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.

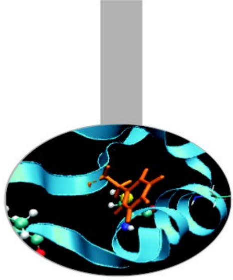
Note that the new group must be a subset of the group associated with comm!

EXAMPLE (C)

```
#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;
}
```

Hypothesis: nprocs=8 credits: http://static.msi.umn.edu

MPI COMM SPLIT



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

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

- *comm* is the old communicator
- *color* is an array of integers specifying on which group should a process belong to in the new communicator
- *key* is an array of integer that defines the rank that the process will get in the new communicator, that will be ssigned in increasing order depending on the associated key value
- *newcomm* is the new communicator

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

MPI_UNDEFINED is used as a color when the process shouldn't be included in any of the new groups

MPI COMM SPLIT

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

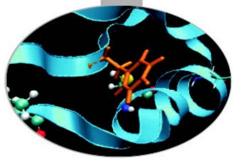
- {i, c, d}
- {k, b, e, g, h}
- {f}

MPI COMM SPLIT – EXAMPLE (C)

```
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_WORL %d”,myid,”rank in Subcomm %d”,mynewid);
```

```
I am rank 2 in M                               mm 1.
I am rank 7 in                                  2.
I am rank 0
I am ran                                         .
I am ra          WO                             1.
I am              ORLD,                     mm 2.
I a               RLD, but 2 i        omm 2.
I am ra            LD, but 0 in Comm 2.
```

DESTRUCTORS

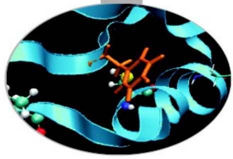


The communicators and groups from a process' viewpoint are just handles.
Like all handles, there is a limited number available: you could (in principle) run out!

MPI_GROUP_FREE(group, ierr)
MPI_COMM_FREE(comm,ierr)

Remember to free your handles after they are no longer needed, it is always a good practice (like with allocatable arrays)

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