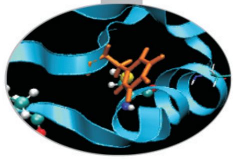
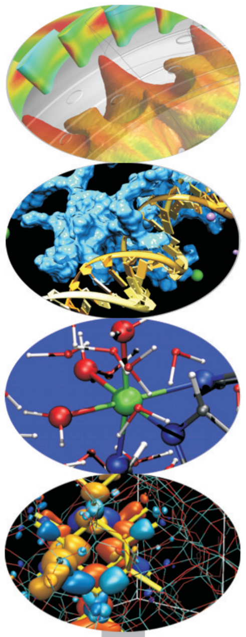
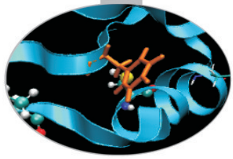
SuperComputing Applications and Innovation



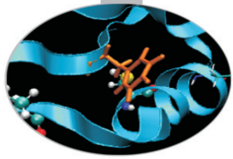
INTRODUCTION TO MPI – COLLECTIVE COMMUNICATIONS AND COMMUNICATORS



Introduction to Parallel Computing with MPI and OpenMP



Part I:
Collective communications



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:

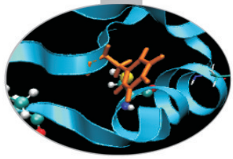
- Synchronization collectives
- Message passing collectives
- Reduction collectives

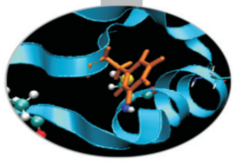
SuperComputing Applications and Innovation

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)

It'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?

SuperComputing Applications and Innovation

POINT-TO-POINT SOLUTION

```fortran
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
```

```c
#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 *:*

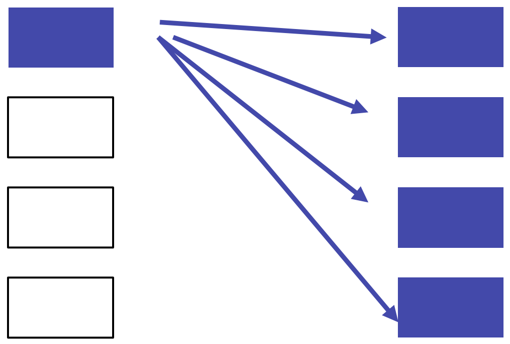
*int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

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

```fortran
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
```

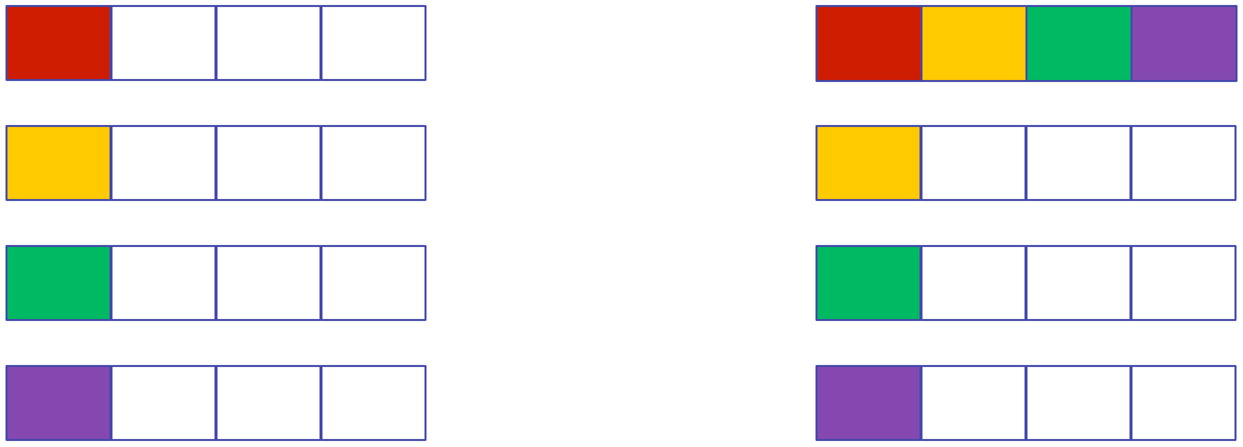
```c
#include <mpi.h>
#include <stdio.h>

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

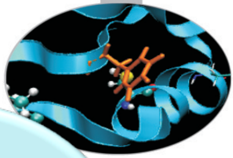
C :
*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)*

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.

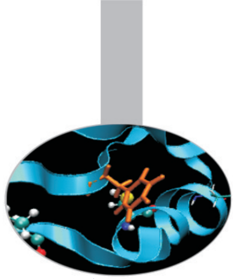


GATHER EXAMPLE (C)



```
#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



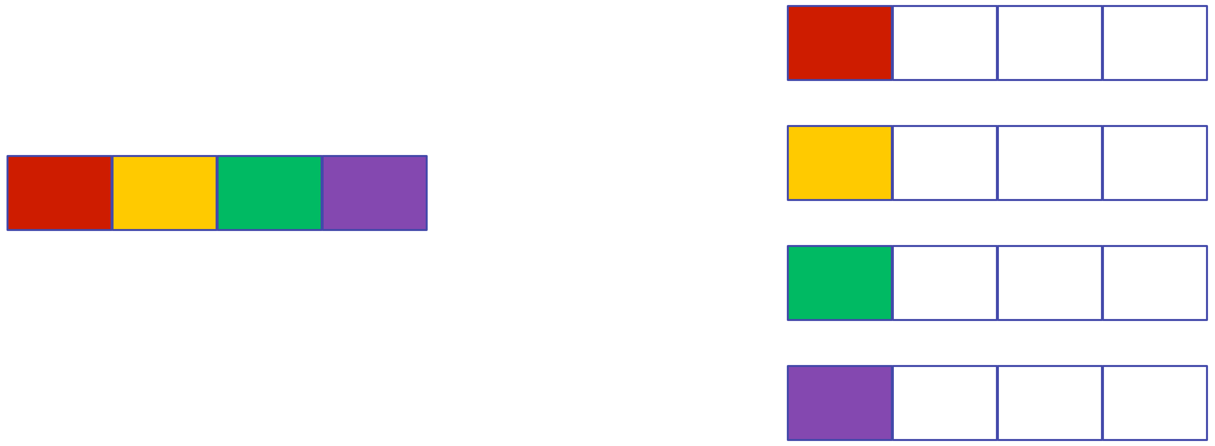
C :

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

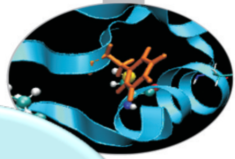
Fortran :

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.

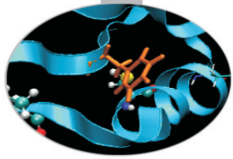


SCATTER EXAMPLE (FORTRAN)



```
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 :

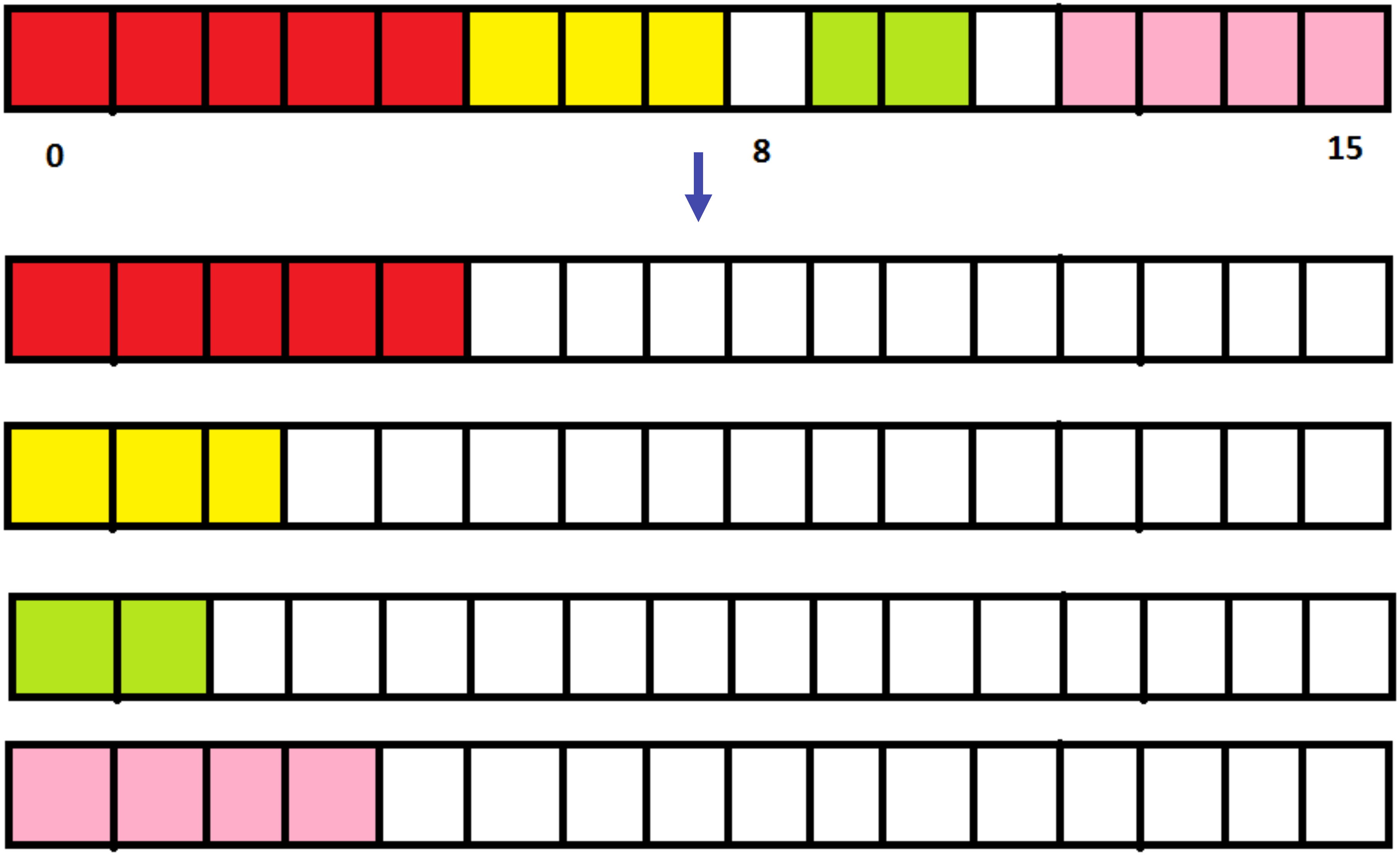
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, IERROR

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

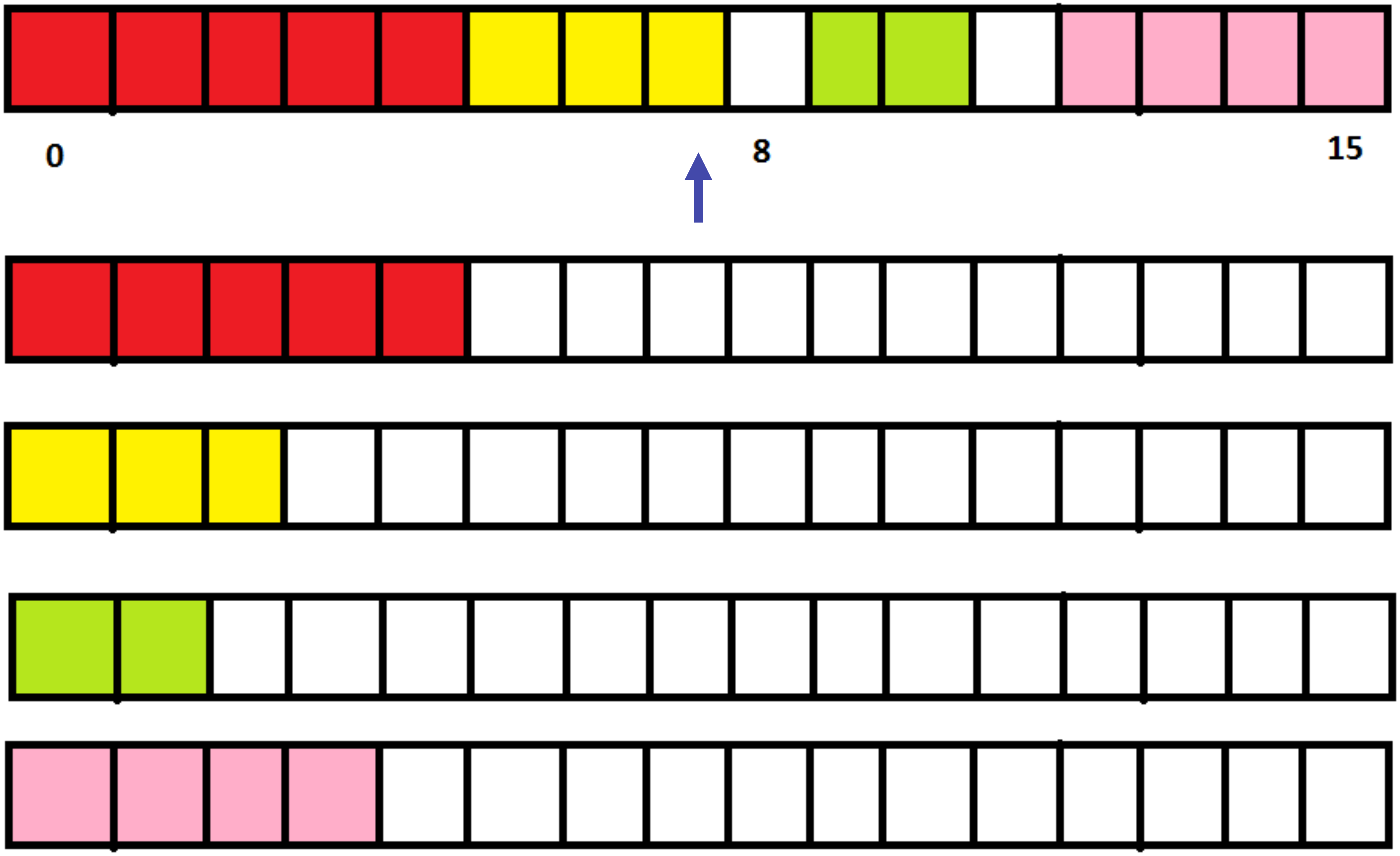
SCATTERV & GATHERV

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

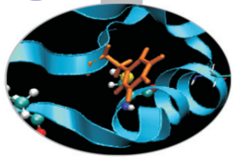


SCATTERV & GATHERV

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



COLLECTIVE COMBINATIONS



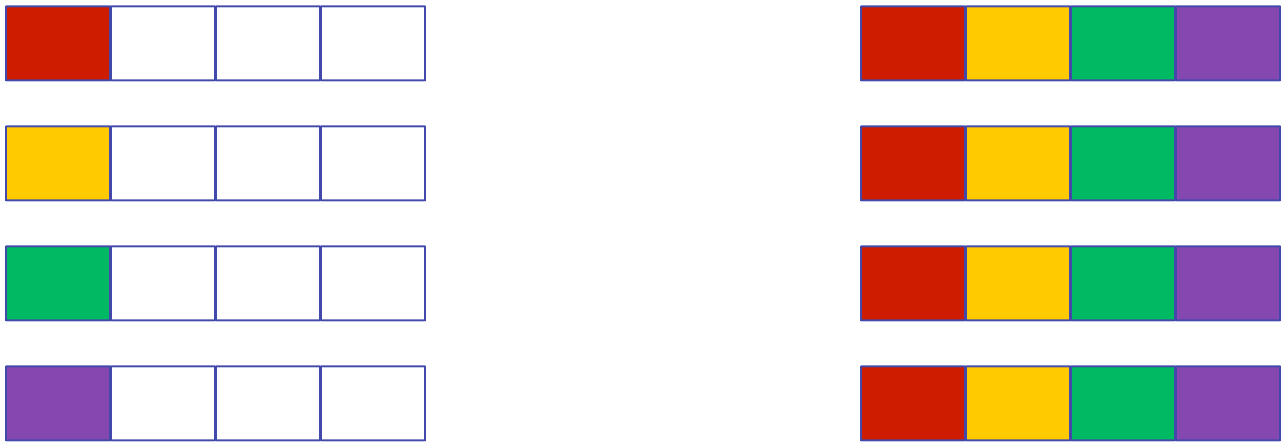
There are functions that combine the effects of two collective functions!
For example, **MPI Allgather** is a combination of a gather + a broadcast

C :

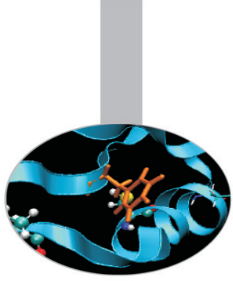
*int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

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 *:*

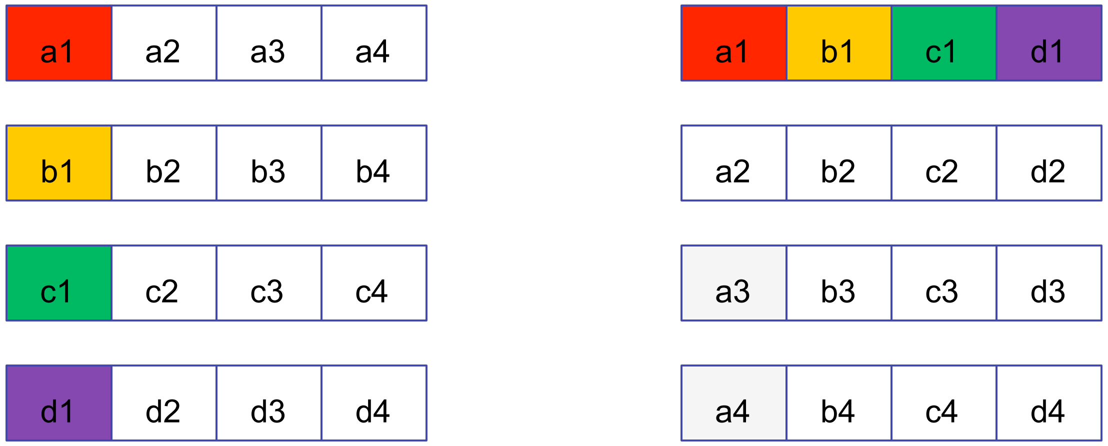
*int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

FORTRAN *:*

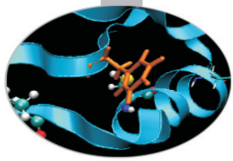
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

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

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



REDUCTION OPERATIONS



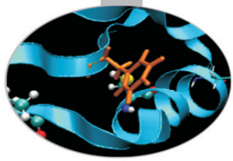
Reduction operations permit to:

- 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)*

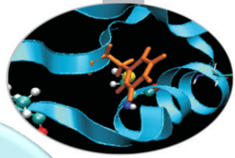
FORTRAN *:*
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

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)

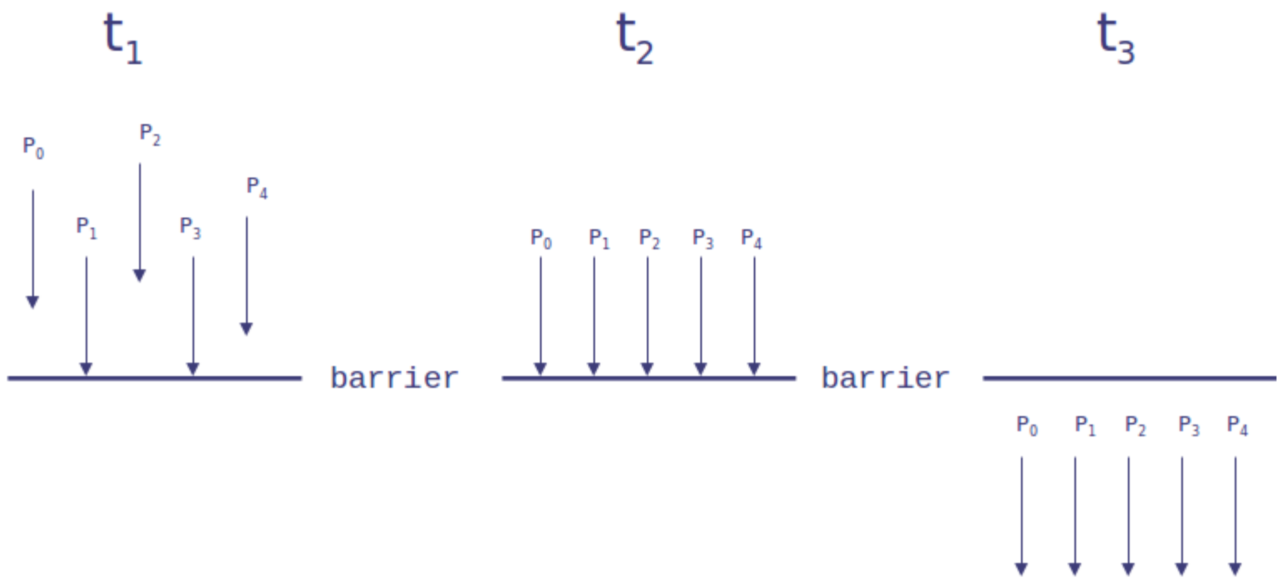


```
PROGRAM reduce
INCLUDE ’mpif.h’
INTEGER ierr, myid, nproc, root
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
a(1) = 2.0
a(2) = 4.0
CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root, &
MPI_COMM_WORLD, 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
```

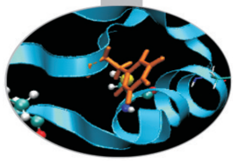
SYNCHRONIZATION COMMAND: MPI BARRIER

It stops all processes within a communicator until they are synchronized

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

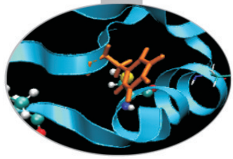


SuperComputing Applications and Innovation



Part II:
MPI communicators and groups

WHAT ARE COMMUNICATORS?

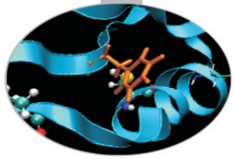


Many users are familiar with the mostly used communicator:
MPI_COMM_WORLD

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:

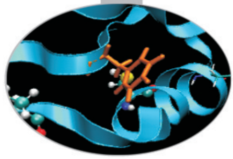
- **MPI_GROUP_SIZE**

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

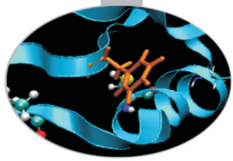


Group constructors are used to create new groups from existing ones (initially from the group associated with MPI_COMM_WORLD; you can use mpi_comm_group to get this).

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

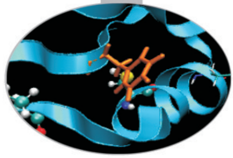
- **MPI_GROUP_INTERSECTION**(group_a, group_b, newgroup, ierr)

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]

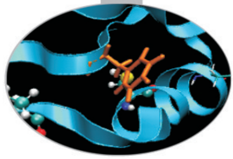
Example:
group = {a,b,c,d,e,f,g,h,i,j}
n = 5
ranks = {0,3,8,6,2}
newgroup = {a,d,i,g,c}

- **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

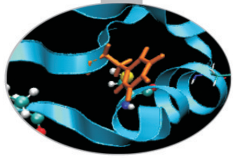


Communicator access operations are local, not requiring interprocess communication

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)

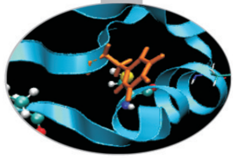
Returns the rank of the calling process within the group associated with the comm

- **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)

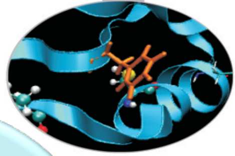
This returns a communicator newcomm identical to the communicator comm

- **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!

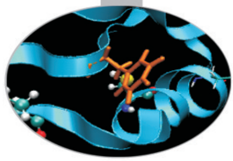
EXAMPLE (C)



```
#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}

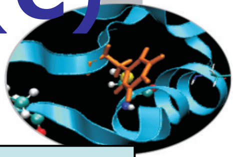
{k, b, e, g, h}

{f}

MPI COMM SPLIT – EXAMPLE (C)

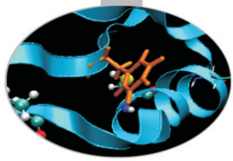
```
if(myid%2==0){
        color=1;
}else{
        color=2;
}
MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,&subcomm);
MPI_COMM_RANK(subcomm,mynewid);
printf("rank in MPICOMM_WORLD %d",myid,"rank in Subcomm %d",mynewid);
```

```
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.
```



SuperComputing Applications and Innovation

DESTRUCTORS

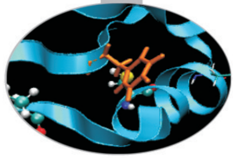


The communicators and groups from a process' viewpoint are just handles.
Like all handles, there is a limited number available: you could (in principle) run out!

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

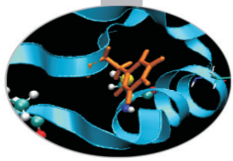


Intercommunicators are associated with 2 groups of disjoint processes.

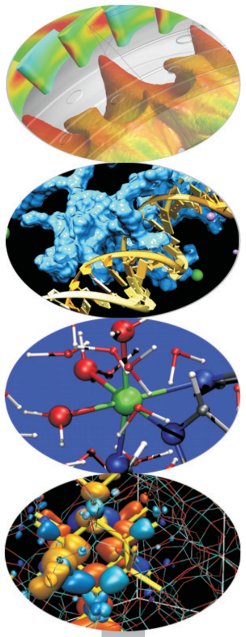
Intercommunicators are associated with a remote group and a local group

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

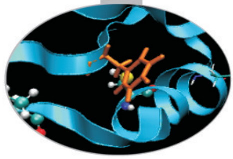


INTRODUCTION TO MPI – VIRTUAL TOPOLOGIES



Introduction to Parallel Computing with MPI and OpenMP

VIRTUAL TOPOLOGY



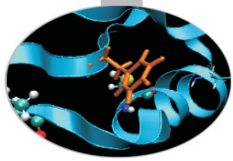
Topology:

- extra, optional attribute that can be given to an intra-communicator; topologies cannot be added to inter-communicators.
- can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

A process group in MPI is a collection of n processes:

- each process in the group is assigned a rank between 0 and n-1.
- in many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used).

VIRTUAL TOPOLOGY



Virtual topology:

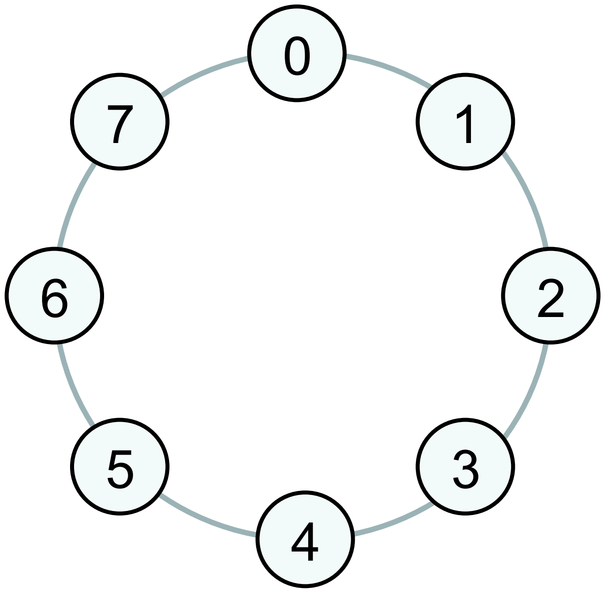
- logical process arrangement in topological patterns such as 2D or 3D grid; more generally, the logical process arrangement is described by a graph.

Virtual process topology .vs. topology of the underlying, physical hardware:

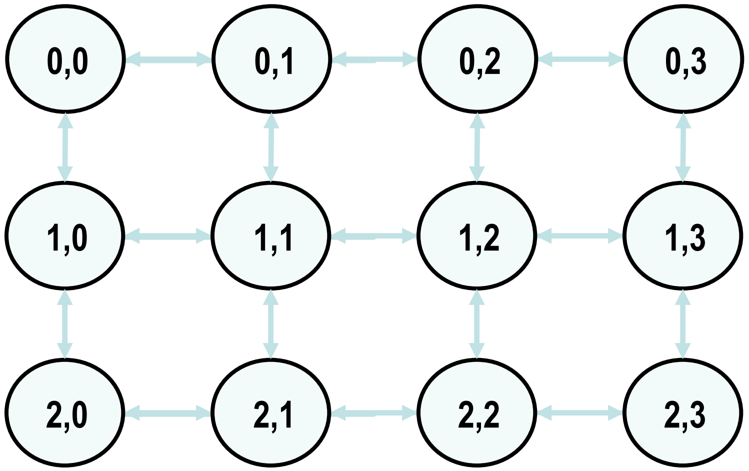
-virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine.

-the description of the virtual topology depends only on the application, and is machine-independent.

EXAMPLES



RING

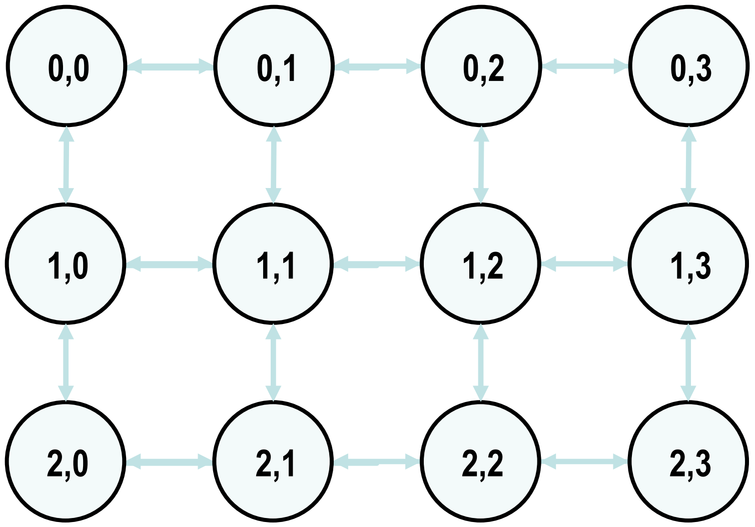


2D-GRID

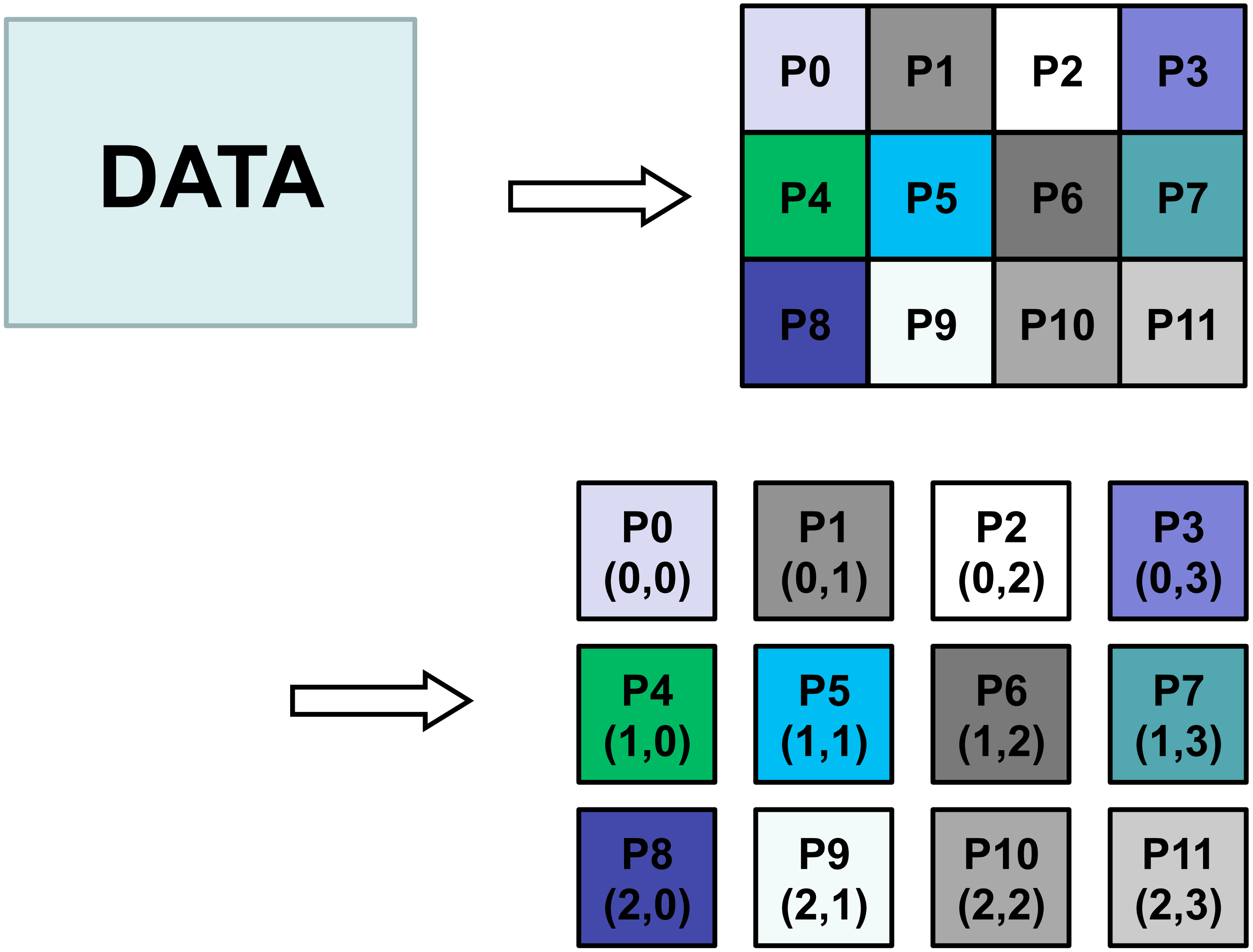
CARTESIAN TOPOLOGY

A grid of processes is easily described with a cartesian topology:

- each process can be identified by cartesian coordinates
- periodicity can be selected for each direction
- communications are performed along grid dimensions only



EXAMPLE: 2D DOMAIN DECOMPOSITION

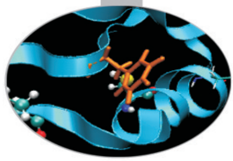


CARTESIAN CONSTRUCTOR

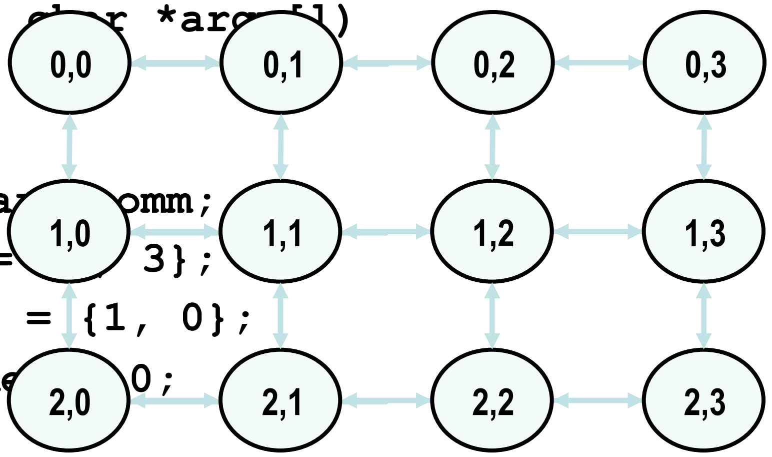
```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
                          comm_cart)
IN comm_old:   input communicator (handle)
IN ndims: number of dimensions of Cartesian grid (integer)
IN dims: integer array of size ndims specifying the number of
processes in each dimension
IN periods: logical array of size ndims specifying whether the      grid is
periodic (true) or not (false) in each dimension
IN reorder: ranking may be reordered (true) or not (false)
OUT comm_cart: communicator with new Cartesian topology (handle)
```

- Returns a handle to a new communicator to which the Cartesian topology information is attached.
- Reorder:
  - false: the rank of each process in the new group is identical to its rank in the old group.
  - True: the processes may be reordered, possibly so as to choose a good embedding of the virtual topology onto physical machine.
- If cart has less processes than starting communicator, left over processes have MPI_COMM_NULL as return

EXAMPLE (C)

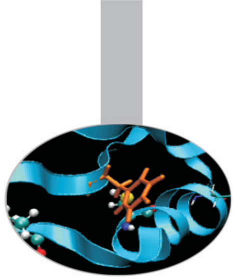


```
#include <mpi.h>

int main(int argc, [illegible] *arg[illegible])
{

    MPI_Comm ca[illegible]omm;
    int dim[] =[illegible] 3};
    int period[] = {1, 0};
    int reorde[illegible]0;

    MPI_Init(&argc, &argv);
```



```
MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &cart_comm);
...
}
```

CARTESIAN TOPOLOGY UTILITIES



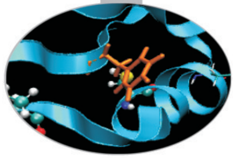
- **MPI_Dims_Create:**
  - compute optimal balanced distribution of processes per coordinate direction with respect to:
    - a given dimensionality
    - the number of processes in a group
    - optional constraints

- **MPI_Cart_coords:**
  - given a rank, returns process's coordinates

- **MPI_Cart_rank:**
  - given process's coordinates, returns the rank

- **MPI_Cart_shift:**
  - get source and destination rank ids in SendRecv operations

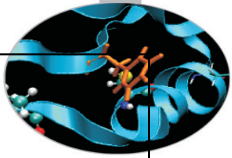
MPI DIMS CREATE



```
MPI_DIMS_CREATE(nnodes, ndims, dims)
IN nnodes: number of nodes in a grid (integer)
IN ndims: number of Cartesian dimensions (integer)
IN/OUT dims: integer array of size ndims specifying the number of
nodes in each dimension
```

- Help user to select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user
- if `dims[i]` is set to a positive number, the routine will not modify the number of nodes in that i dimension
- negative value of `dims[i]` are erroneous

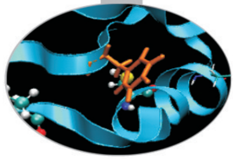
IN/OUT OF "DIMS"



```
MPI_DIMS_CREATE(nnodes, ndims, dims)
IN nnodes: number of nodes in a grid (integer)
IN ndims: number of Cartesian dimensions (integer)
IN/OUT dims: integer array of size ndims specifying the number of
nodes in each dimension
```

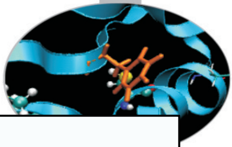
dims before call	Function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 2, dims)	erroneous call

USING MPI_DIMS_CREATE (FORTRAN)



```
integer :: dim(3),period(3),reorder, cube_comm, ierr

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs,ierr)

dim(1) = 0 ! let MPI arrange
dim(2) = 0 ! let MPI arrange
dim(3) = 3 ! I want exactly 3 planes

CALL MPI_DIMS_CREATE(nprocs, 3, dim, ierr)

if (dim(1)*dim(2)*dim(3) .LE. nprocs) then
print *,"WARNING: some processes are not in use!"
endif

period = (1, 1, 0)
reorder = 0

CALL MPI_CART_CREATE(MPI_COMM_WORLD, 3, dim, period, reorder, &
cube_comm,ierr)
```

FROM COORDINATE TO RANK

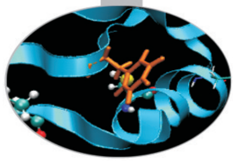


`MPI_CART_RANK(comm, coords, rank)`

```
IN comm: communicator with Cartesian structure
IN coords: integer array (of size ndims) specifying the Cartesian
coordinates of a process
OUT rank: rank of specified process
```

- translation of the logical process coordinates to process ranks as they are used by the point-to-point routines
- if `dimension i` is periodic, when i-th coordinate is out of range, it is shifted back to the interval `0<coords(i)<dims(i)` automatically
- out-of-range coordinates are erroneous for non-periodic dimensions

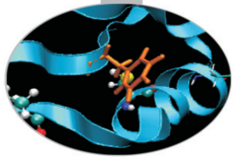
FROM RANK TO COORDINATE



```
MPI_CART_COORDS(comm, rank, maxdim, coords)
IN comm: communicator with Cartesian structure
IN rank: rank of a process within group of comm
IN maxdims: length of vector coords in the calling program
OUT coords: integer array (of size ndims) containing the Cartesain
coordinates of specified process
```

- For each MPI process in Cartesian communicator, the coordinate whitin the cartesian topology are returned

MAPPING OF COORDINATES (C)

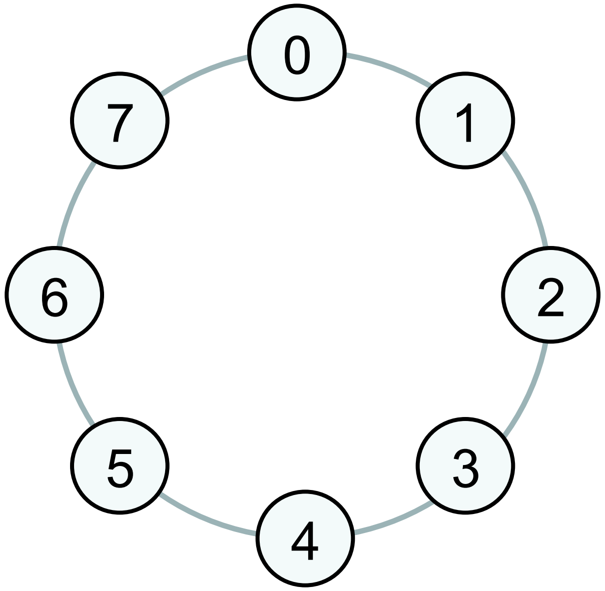


```
int cart_rank;
MPI_Comm_rank(cart_comm, &cart_rank);

int coords[2];
MPI_Cart_coords(cart_comm, cart_rank, 2, coords);

// set linear boundary values on bottom/left-hand domain
if (coords[0] == 0 || coords[1] == 0) {
SetBoundary( linear(min, max), domain);
}

// set sinusoidal boundary values along upper domain
if (coords[0] == dim[0]) {
SetBoundary( sinusoid(), domain);
}

// set polynomial boundary values along right-hand of domain
if (coords[1] == dim[1]) {
SetBoundary( polynomial(order, params), domain);
}
```

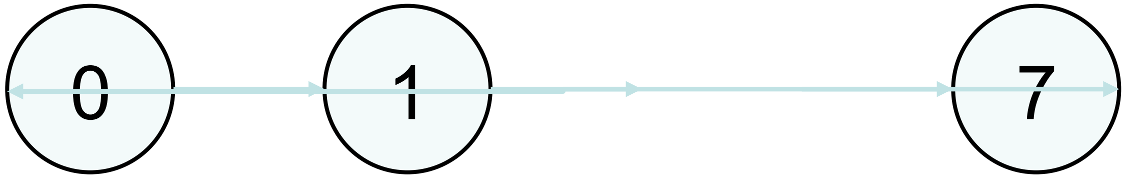
CARTESIAN SHIFT: A 1D CARTESIAN TOPOLOGY

Circular shift is another tipical MPI communication pattern:

- each process communicate only with its neighbors along one direction
- periodic boundary conditions can be set for letting first and last processes partecipate in the communication



such a pattern is nothing more than a 1D cartesian grid topology with optional periodicity

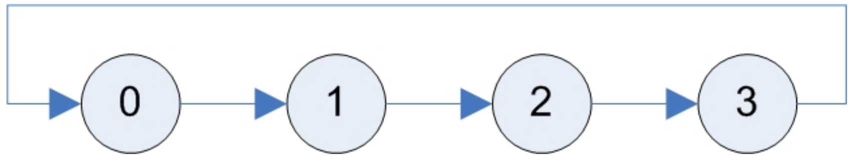


MPI CART SHIFT

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
IN comm: communicator with Cartesian structure
IN direction: coordinate dimension of shift
IN disp: displacement (>0: upwards shift; <0: downwards shift
OUT rank_source: rank of source process
OUT rank_dest: rank of destination process
```

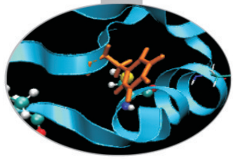
- Depending on the periodicity of the Cartesian group in the specied coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-o shift.
- In the case of an end-o shift, the value **MPI_PROC_NULL** may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.
- provides the calling process the ranks of source and destination processes for an MPI_SENDRECV with respect to a specified coordinate direction and step size of the shift

EXAMPLE (FORTRAN)



```
...
integer ::  dim = nprocs
integer ::  period = 1
integer ::  source, dest, ring_comm, status(MPI_STATUS_SIZE),ierr

CALL MPI_CART_CREATE(MPI_COMM_WORLD, 1, dim, period, 0,ring_comm,ierr)

CALL MPI_CART_SHIFT(ring_comm, 0, 1, source, dest, ierr)

CALL MPI_SENDRECV(right_bounday, n, MPI_INT, dest, rtag, left_boundary,
      n, MPI_INT, source, ltag, ring_comm, status, ierr)

...
```

PARTITIONING OF CARTESIAN STRUCTURES



- It is often useful to partition a cartesian communicator into subgroups that form lower dimensional cartesian subgrids

  - new communicators are derived

  - lower dimensional communicators cannot communicate among them (unless inter-communicators are used)

MPI CART SUB

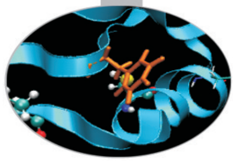
MPI_CART_SUB(comm, remain_dims, newcomm)

IN comm: communicator with Cartesian structure

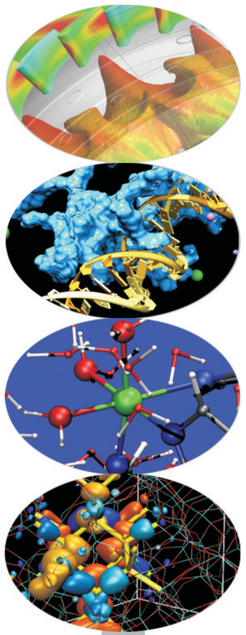
IN remain_dims: the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)

OUT newcomm: communicator containing the subgrid that includes the calling process

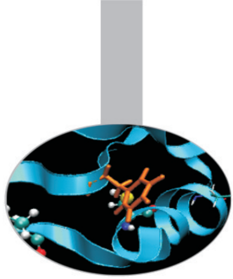
```
int dim[] = {2, 3, 4};

int remain_dims[] = {1, 0, 1}; // 3 comm with 2x4 processes 2D grid
...
int remain_dims[] = {0, 0, 1}; // 6 comm with 4 processes 1D topology
```



INTRODUCTION TO MPI – MPI DATATYPES



Introduction to Parallel Computing with MPI and OpenMP



DERIVED DATATYPE

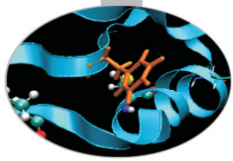
What are?

Derived datatypes are datatypes that are built from the basic MPI datatypes (e.g. MPI_INT, MPI_REAL, …)

Why datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lenghts of elementary datatypes (heterogeneous communication)
- Specifying application-oriented layout of data in memory
  - -can reduce memory-to memory copies in the implementaion
  - -allows the use of special hardware (scatter/gather) when available
- Specifying application-oriented layout of data on a file can reduce systems calls and physical disk I/O

DERIVED DATATYPE



You may need to send messages that contain:

1. non-contiguous data of a single type (e.g. a sub-block of a matrix)
2. contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers)
3. non-contiguous data of mixed types

Possible solutions:

1. make multiple MPI calls to send and receive each data element

   → If advantegeous, copy data to a buffer before sending it

2. use MPI_pack/MPI_Unpack to pack data and send packed data (datatype MPI_PACKED)
3. use MPI_BYTE to get around the datatype-matching rules. Like MPI_PACKED, MPI_BYTE can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte.

Additional latency costs due to multiple calls

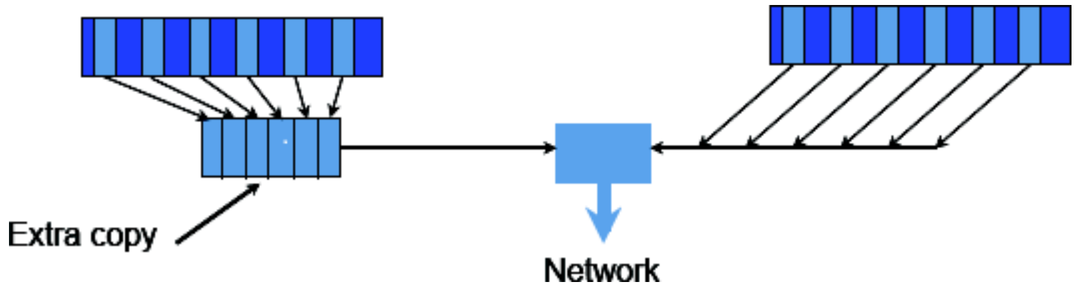
Additional latency costs due to memory copy

Not portable to a heterogeneous system using MPI_BYTE or

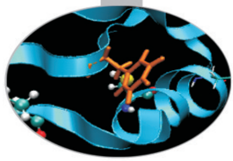
DERIVED DATATYPE

Datatype solution:

1. The idea of MPI derived datatypes is to provide a simple, portable, elegant and efficient way of communicating non-contiguous or mixed types in a message.
   - During the communication, the datatype tells MPI system where to take the data when sending or where to put data when receiving.
2. The actual performances depend on the MPI implementation
3. **Derived datatypes are also needed for getting the most out of MPI-I/O.**



DEFINITION



A **general datatype** is an **opaque object** able to describe a buffer layout in memory by specifing:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements.

Typemap = {(type 0, displ 0), … (type n-1, displ n-1)}

– pairs of basic types and displacements (in byte)

Type signature = {type 0, type 1, … type n-1}

– list of types in the typemap

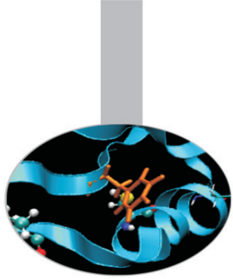
– gives size of each elements

– tells MPI how to interpret the bits it sends and received

Displacement:

– tells MPI where to get (when sending) or put (when receiving)

TYPEMAP



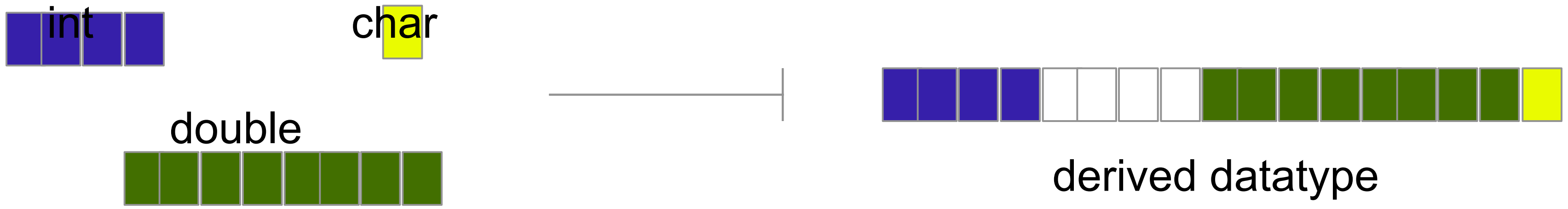
Example:

Basic datatype are particular cases of a general datatype, and are predefined:

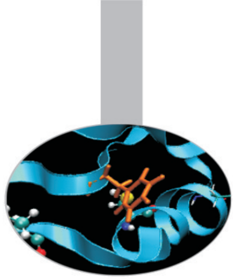
MPI_INT = {(int, 0)}

General datatype with typemap

Typemap = {(int,0), (double,8), (char,16)}



HOW TO USE



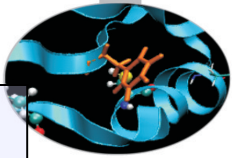
General datatypes (differently from C or Fortran) are created (and destroyed) at run-time through calls to MPI library routines.

Implementation steps are:

1. Creation of the datatype from existing ones with a **datatype constructor**.
2. Allocation (**committing**) of the datatype before using it.
3. **Usage of the derived datatype** for MPI communications and/or for MPI-I/O
4. Deallocation (**freeing**) of the datatype after that it is no longer needed.

SuperComputing Applications and Innovation

COMMITTING AND FREEING



MPI_TYPE_COMMIT (datatype)

INOUT datatype: datatype that is committed (handle)

- Before it can be used in a communication or I/O call, each derived datatype has to be committed

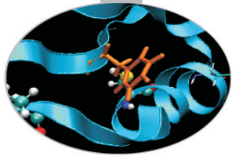
MPI_TYPE_FREE (datatype)

INOUT datatype: datatype that is freed (handle)

Mark a datatype for deallocation

Datatype will be deallocated when all pending operations are finished

MPI TYPE CONTIGUOUS



MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)

IN count: replication count (non-negative integer)

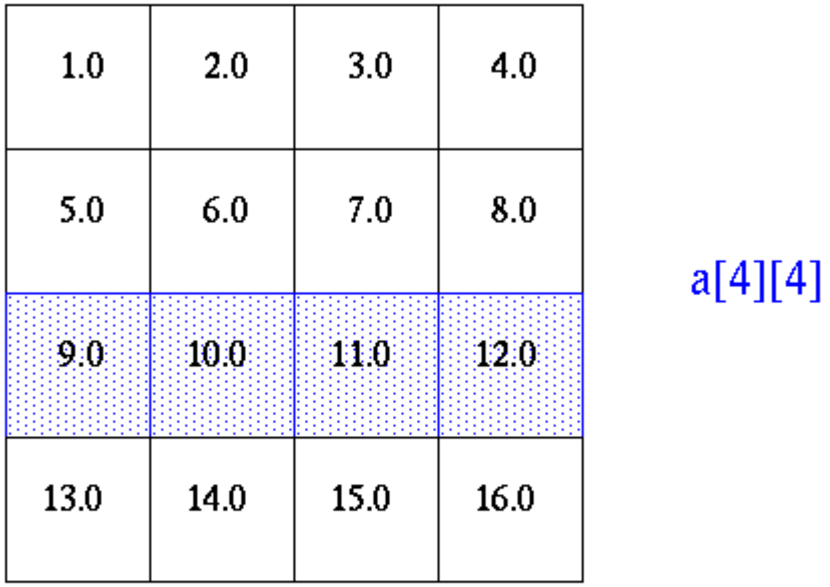
IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- MPI_TYPE_CONTIGOUS constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations.
- newtype is the datatype obtained by concatenating count copies of oldtype.

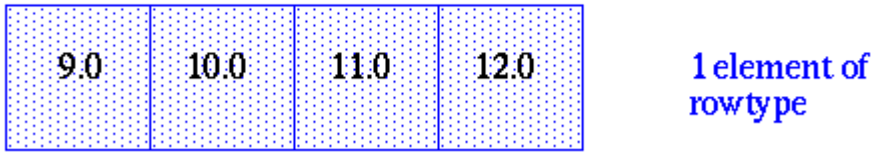
MPI TYPE CONTIGUOUS

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```



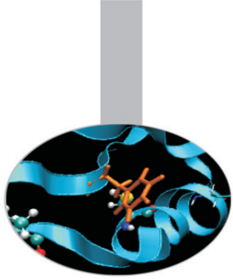
1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```



9.0	10.0	11.0	12.0

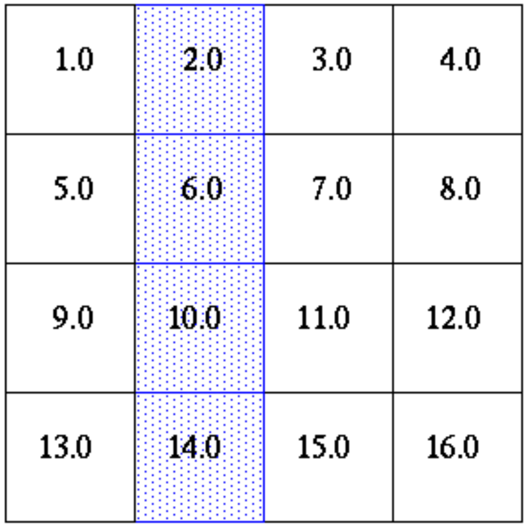
MPI TYPE VECTOR



MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
IN count: Number of blocks (non-negative integer)
IN blocklen: Number of elements in each block
(non-negative integer)
IN stride: Number of elements (NOT bytes) between start of
each block (integer)
IN oldtype: Old datatype (handle)
OUT newtype: New datatype (handle)

- Consists of a number of elements of the same datatype repeated with a certain stride

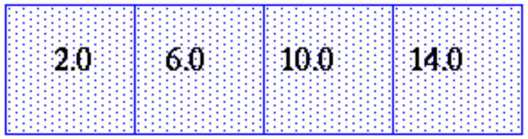
```
count = 4;   blocklength = 1;   stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
&columntype);
```



1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

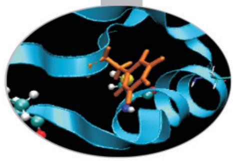
```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```



2.0	6.0	10.0	14.0

1 element of columntype

MPI TYPE HVECTOR



MPI_TYPE_CREATE_HVECTOR (count, blocklength, stride, oldtype, newtype)

IN count: Number of blocks (non-negative integer)
IN blocklen: Number of elements in each block (non-negative integer)
IN stride: Number of bytes between start of each block (integer)
IN oldtype: Old datatype (handle)
OUT newtype: New datatype (handle)

- It’s identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements
- “H” stands for heterogeneous

MPI TYPE INDEXED

MPI_TYPE_INDEXED (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN count: number of blocks – also number of entries in array_of_blocklenghts and array_of_displacements (non-negative integer)

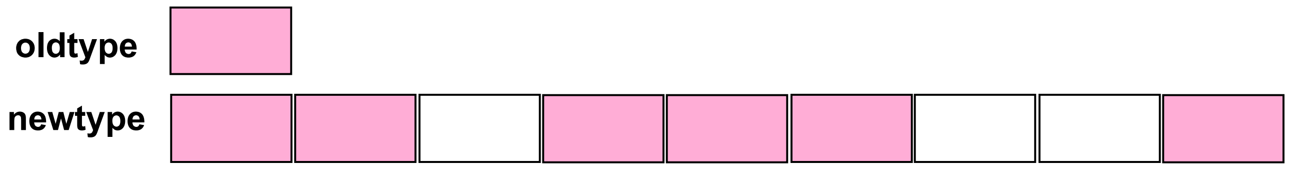
IN array_of_blocklengths: number of elements per block (array of non-negative integers)

IN array_of_displacements: displacement for each block, in multiples of oldtype extent (array of integer)

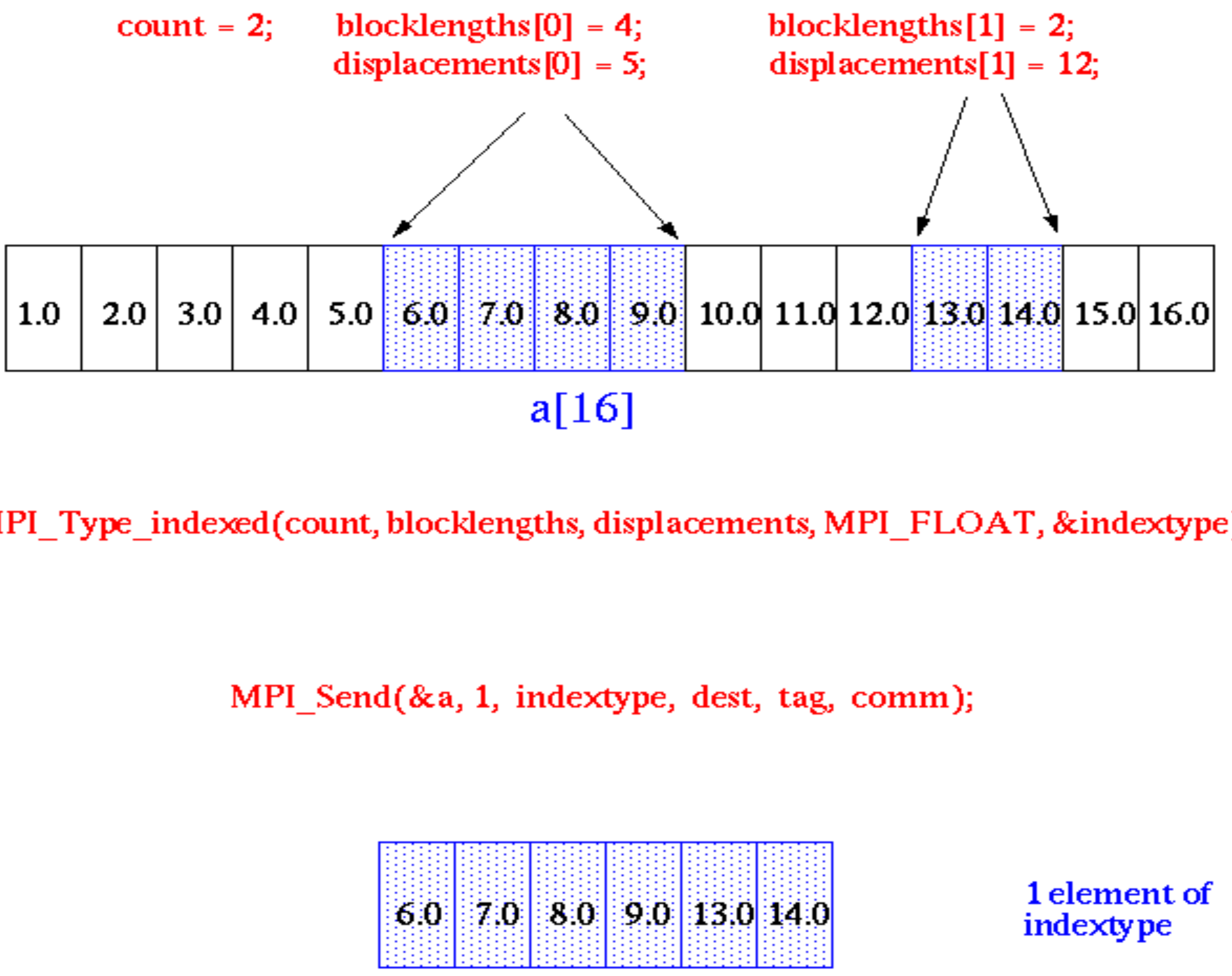
IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- Creates a new type from blocks comprising identical elements
  - The size and displacements of the blocks can vary

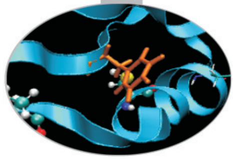


count=3, array_of_blocklenghths=(/2,3,1/), array_of_displacements=(/0,3,8/)

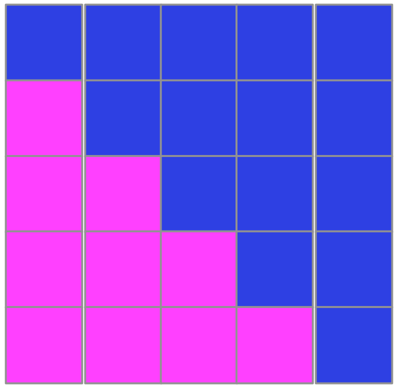


```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

MPI_Send(&a, 1, indextype, dest, tag, comm);
```

MPI TYPE INDEXED EXAMPLE (FORTRAN)



```
! upper triangular matrix
real, dimension(100,100) :: a
integer, dimension(100) :: displ, blocklen
integer :: i, upper, ierr

! compute start and size of the rows
do i=1,100
displ(i) = 100*i+i

blocklen(i) = 100-i

end do
```

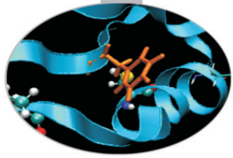


```
! create and commit a datatype for upper triangular matrix
```
CALL MPI_TYPE_INDEXED (100, blocklen, disp, MPI_DOUBLE, upper,ierr)

CALL MPI_TYPE_COMMIT (upper,ierr)

```
! … send it ...
CALL MPI_SEND (a, 1, upper, dest, tag, MPI_COMM_WORLD, ierr)
```
MPI_Type_free (upper, ierr)

MPI TYPE HINDEXED



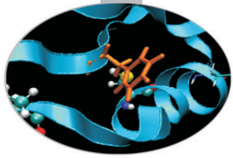
MPI_TYPE_CREATE_HINDEXED (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN count: number of blocks – also number of entries in array_of_blocklengths and array_of_displacements (non-negative integer)
IN array_of_blocklengths: number of elements in each block (array of non-negative integers)
IN array_of_displacements: byte displacement of each block (array of integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)

- This function is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather that in multiples of the oldtype extent

SuperComputing Applications and Innovation

MPI TYPE INDEXED BLOCK

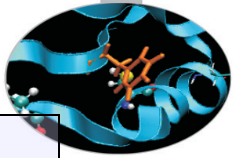


MPI_TYPE_CREATE_INDEXED_BLOCK (count, blocklengths, array_of_displacements, oldtype, newtype)

IN count: length of array of displacements (non-negative integer)
IN blocklengths: size of block (non-negative integer)
IN array_of_displacements: array of displacements (array of integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)

- Similar to MPI_TYPE_INDEXED, except that the block-length is the same for all blocks.
- There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). This function allows for constant blocksize and arbitrary displacements.

MPI TYPE SUBARRAY

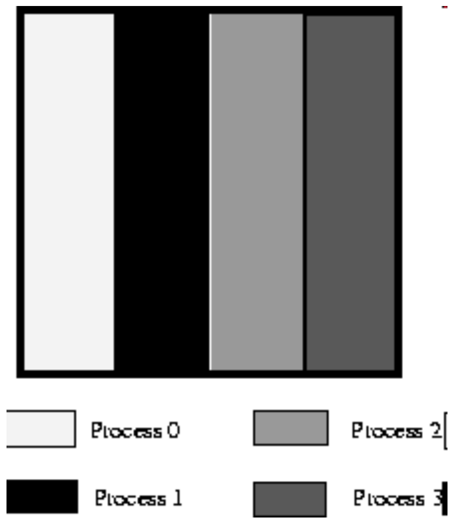


MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN ndims: number of array dimensions (positive integer)
IN array_of_sizes: number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN array_of_subsizes: number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN array_of_starts: starting coordinates of the subarray in each dimension (array of non-negative integers)
IN order: array storage order flag (state: MPI_ORDER_C or MPI_ORDER_FORTRAN)
IN oldtype: array element datatype (handle)
OUT newtype: new datatype (handle)

The subarray type constructor creates an MPI datatype describing an n dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

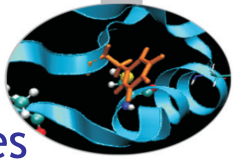
MPI TYPE SUBARRAY EXAMPLE (C)



MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts,
MPI_ORDER_C, MPI_DOUBLE, &filetype);

MPI_Type_commit(&filetype);
```

SIZE AND EXTENT

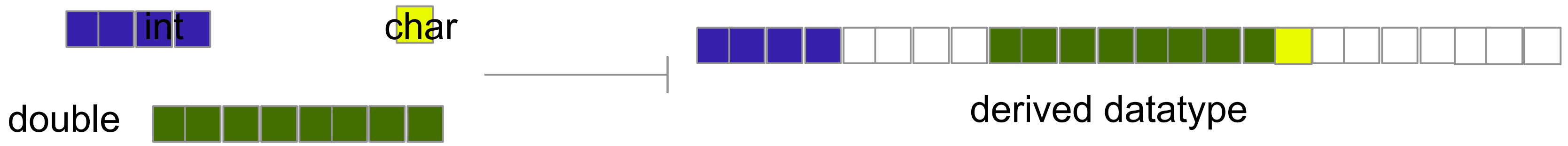


The MPI datatype for structures – MPI_TYPE_CREATE_STRUCT – requires dealing with memory addresses and further concepts:

Typemap: pairs of basic types and displacements

Extent: The **extent** of a datatype is the span from the lower to the upper bound **(including "holes")**

Size: The **size** of a datatype is the net number of bytes to be transferred **(without "holes")**



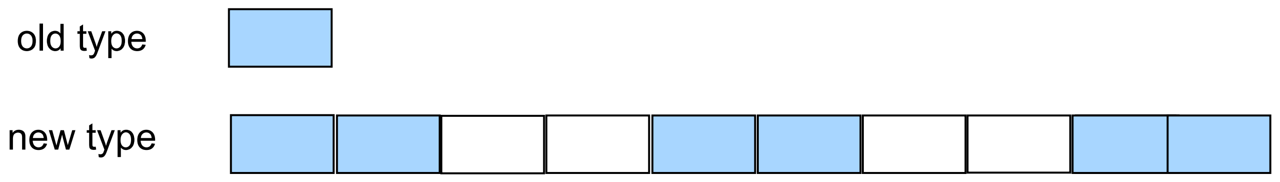
SIZE AND EXTENT

Basic datatypes:

- size = extent = number of bytes used by the compiler

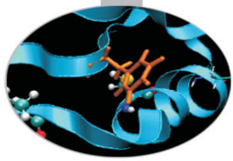
Derived datatypes:

- **extent include holes but...**
- **beware of the type vector: final holes are a figment of our imagination**



- size = 6 x size of “old type”
- extent = 10 x extent of “old type”

QUERY SIZE AND EXTENT OF DATATYPE



- Returns the total number of bytes of the entry datatype

MPI_TYPE_SIZE (datatype, size)

IN datatype: datatype (handle)

OUT size: datatype size (integer)

- Returns the lower bound and the extent of the entry datatype

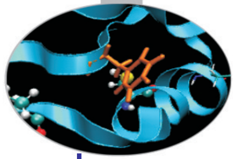
MPI_TYPE_GET_EXTENT (datatype, lb, extent)

IN datatype: datatype to get information on(handle)

OUT lb: lower bound of datatype (integer)

OUT extent: extent of datatype (integer)

EXTENT



- Extent controls how a datatype is used with the count field in a send and similar MPI operations
- Consider

```
call MPI_Send(buf,count,datatype,...)
```

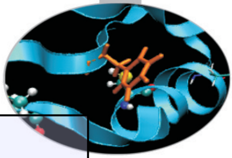
- What actually gets sent?

```
do i=0,count-1
call MPI_Send(bufb(1+i*extent(datatype)),1,datatype,...)
enddo
```

where *bufb* is a byte type like *integer*1*

- *extent* is used to decide where to send from (or where to receive to in MPI_Recv) for count>1
  - Normally, this is right after the last byte used for (i-1)

MPI TYPE STRUCT



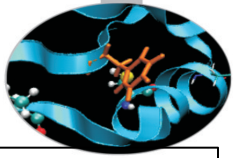
```
MPI_TYPE_CREATE_STRUCT (count, array_of_blocklengths,
                        array_of_displacements, array_of_oldtypes, newtype )
```

IN count: number of blocks (non-negative integer) -- also number of entries the following arrays
IN array_of_blocklenghts: number of elements in each block (array of non-negative integer)
IN array_of_displacements: byte displacement of each block (array of integer)
IN array_of_oldtypes: type of elements in each block (array of handles to datatype objects)
OUT newtype: new datatype (handle)

This subroutine returns a new datatype that represents count blocks. Each block is defined by an entry in array_of_blocklengths, array_of_displacements and array_of_types.

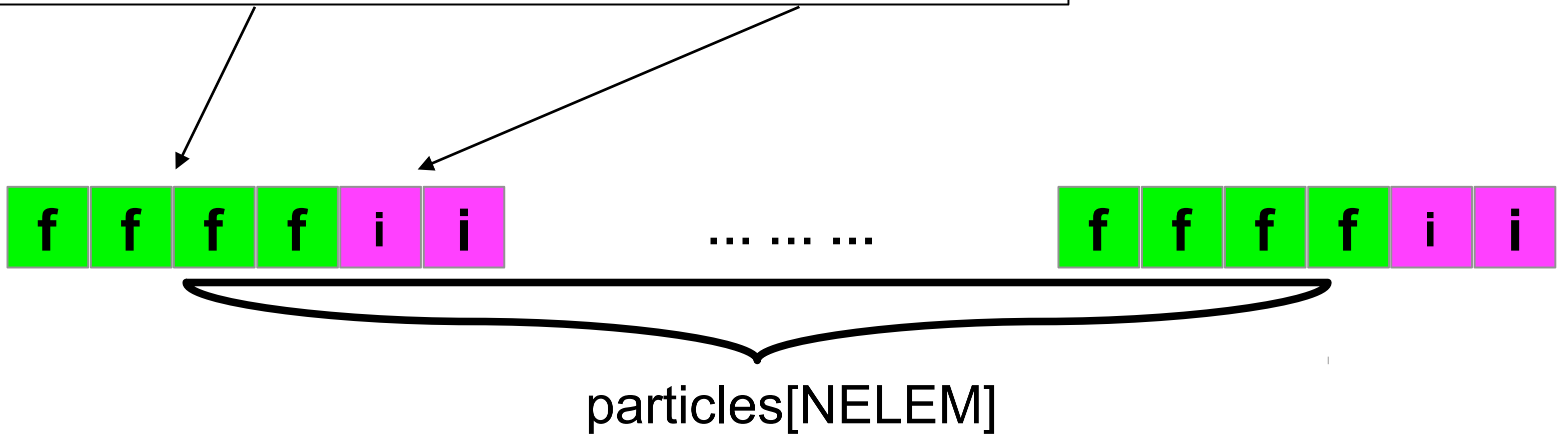
- Displacements are expressed in bytes (since the type can change!)
- To gather a mix of different datatypes scattered at many locations in space into one datatype that can be used for the communication.

USING EXTENT (NOT SAFE)



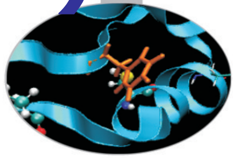
```
struct {
float x, y, z, velocity;
int n, type;
} Particle;

Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);

count = 2;
blockcounts[0] = 4;              blockcount[1] = 2;
oldtypes[0]= MPI_FLOAT;     oldtypes[1] = MPI_INT;
displ[0] = 0;                    displ[1] = 4*extent;
```



```
MPI_Type_struct (count, blockcounts, displ, oldtypes, &particletype);
MPI_Type_commit(&particletype);
```

USING EXTENT (NOT SAFE)



```
struct {
float x, y, z, velocity;
int n, type;
} Particle;

Particle particles[NELEM];
```

```
int count, blockcounts[2];
MPI_Aint displ[2];
MPI_Datatype particletype, oldtypes[2];

count = 2;
blockcounts[0] = 4; blockcount[1] = 2;
oldtypes[0]= MPI_FLOAT; oldtypes[1] = MPI_INT;

MPI_Type_extent(MPI_FLOAT, &extent);
displ[0] = 0; displ[1] = 4*extent;

MPI_Type_create_struct (count, blockcounts, displ, oldtypes,
                        &particletype);

MPI_Type_commit(&particletype);

MPI_Send (particles, NELEM, particletype, dest, tag,
          MPI_COMM_WORLD);

MPI_Free(&particletype);
```

USING EXTENT (NOT SAFE)

- C struct may be automatically padded by the compiler, e.g.

```
struct mystruct {
        char a;
        int b;
        char c;
    } x
```

→

```
struct mystruct {
        char a;
        char gap_0[3];
        int b;
        char c;
        char gap_1[3];
    } x
```

- Using extents to handle structs is not safe! Get the addresses

MPI_GET_ADDRESS (location, address)

IN location: location in caller memory (choice)

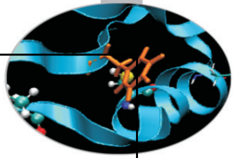
OUT address: address of location (integer)

- The address of the variable is returned, which can then be used to determine the correct relative dispacements
  - Using this function helps with portability

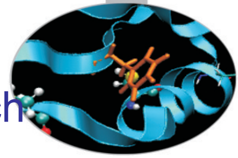
USING DISPLACEMENTS

```
struct PartStruct {
        char class;
      double d[6];
        int b[7];
} particle[100];
```

```
MPI_Datatype ParticleType;
int count = 3;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];

MPI_Get_address(&particle[0].class, &disp[0]);
MPI_Get_address(&particle[0].d, &disp[1]);
MPI_Get_address(&particle[0].b, &disp[2]);
/* Make displacements relative */
disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;

MPI_Type_create_struct (count, blocklen, disp, type,
&ParticleType);
MPI_Type_commit (&ParticleType);

MPI_Send(particle,100,ParticleType,dest,tag,comm);
MPI_Type_free (&ParticleType);
```



FORTRAN TYPES



- According to the standard the memory layout of Fortran derived data is much more liberal
  - An array of types, may be implemented as 5 arrays of scalars!

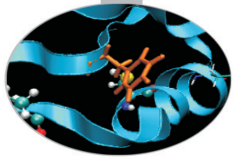
```
type particle
real :: x,y,z,velocity
integer :: n
end type particle
type(particle) :: particles(Np)
```

$\Rightarrow$

```
type particle
sequence
real :: x,y,z,velocity
integer :: n
end type particle
type(particle) :: particles(Np)
```

- The memory layout is guaranteed using sequence or bind(C) type attributes
  - Or by using the (old style) commons...
- With Fortran 2003, MPI_Type_create_struct may be applied to common blocks, sequence and bind(C) derived types
  - it is implementation dependent how the MPI implementation computes the alignments (sequence, bind(C) or other)
- The possibility of passing **particles** as a type depends on MPI implementation: try **particle%x** and/or study the MPI standard and Fortran 2008 constructs

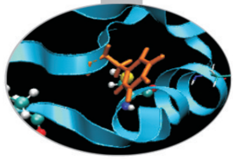
PERFORMANCE



Performance depends on the datatype – more general datatypes are often slower

- some MPI implementations can handle important special cases: e.g., constant stride, contiguous structures
- Overhead is potentially reduced by:
- Sending one long message instead of many small messages
- Avoiding the need to pack data in temporary buffers
- Some implementations are slow

Credits...



These slides have been written, checked and maintained by:

- Alessandro Marani (a.marani@cineca.it)
- Andy Emerson (a.emerson@cineca.it)
- Giusy Muscianisi (g.muscianisi@cineca.it)
- Luca Ferraro (l.ferraro@cineca.it)
- Fabio Affinito (f.affinito@cineca.it)