CINECA
consorzio interuniversitario

Introduction to Message Passing Interface

Fabio AFFINITO

Collective communications

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

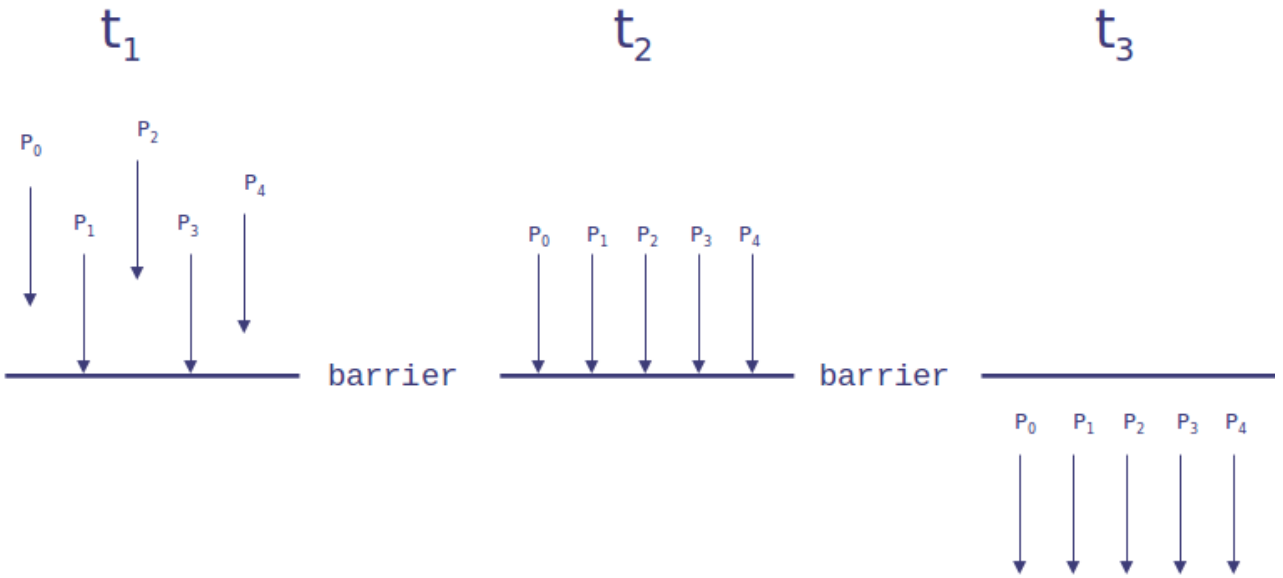
- Barrier synchronization
- Broadcast
- Gather/scatter
- Reduction

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

It’s a safe communication mode

MPI Barrier

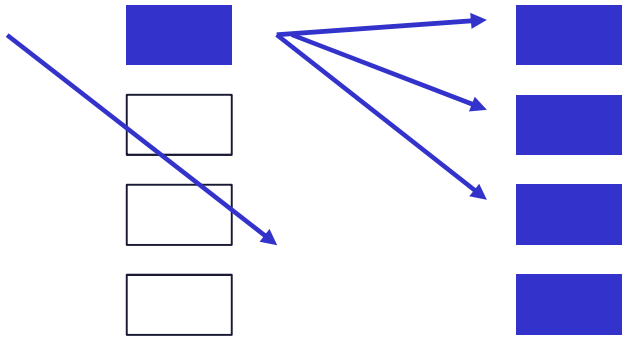
It stops all processes within a communicator until they are synchronized



MPI Broadcast

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

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



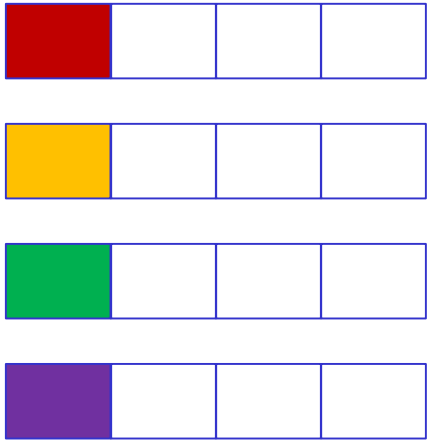
```
PROGRAM broad_cast
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF( myid .EQ. 0 ) THEN
a(1) = 2.0
a(2) = 4.0
END IF
CALL MPI_BCAST(a,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

MPI Gather

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

MPI Scatter

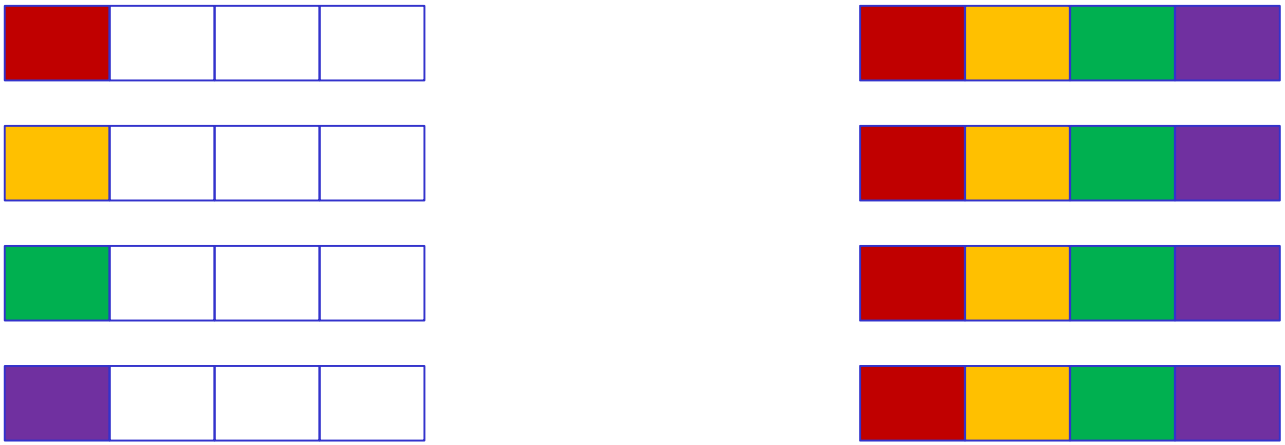
The root sends a message. The message is split into n equal segments, the i-th segment is sent to the i-th process in the group and each process receives this message.



There are possible combinations of collective functions.
For example,

MPI Allgather

It is a combination of a gather + a broadcast

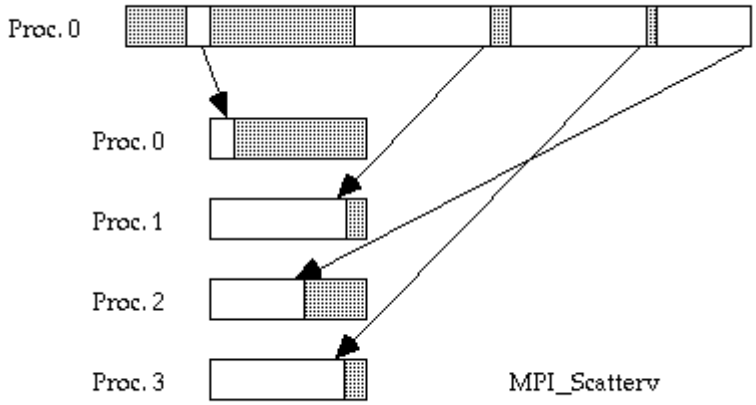


For many collective functions there are extended functionalities.

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

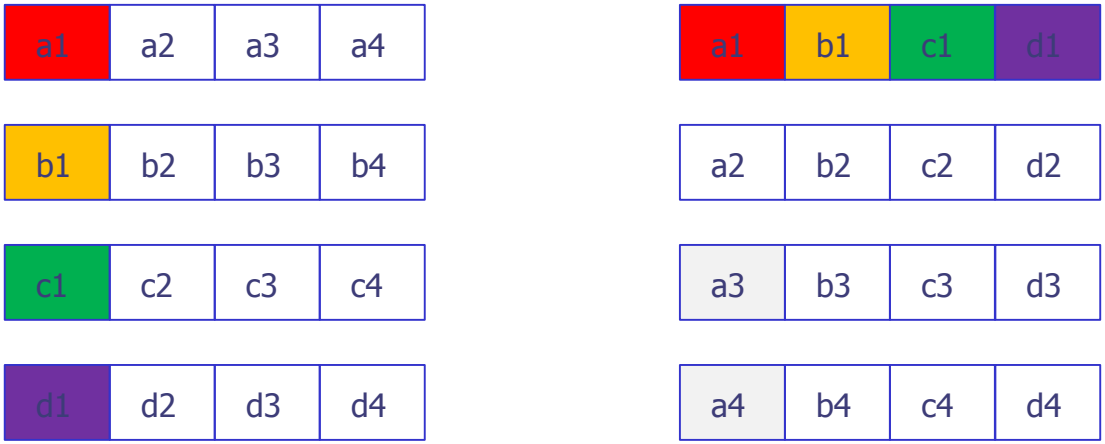
MPI_Scatterv

MPI_Gatherv



MPI All to all

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



Reduction

Reduction operations permits to

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root process (MPI_Reduce) or
- Store the result on all processes (MPI_Allreduce)

Predefined reduction operations

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

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

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

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

MPI 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

Intracommunicators

are used for communications within a single group

Intercommunicators

are used for communications between two disjoint groups

Group management:

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

Group accessors:

- **MPI_GROUP_SIZE**

This routine returns the number of processes in the group

- **MPI_GROUP_RANK**

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

Group constructors

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

Group creation is a local operation: no communication is needed

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

- **MPI_COMM_GROUP(**comm,group,ierr)

This routine returns the group associated with the communicator comm

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

This returns the ensemble union of group_a and group_b

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

This returns the ensemble intersection of group_a and group_b

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

This returns in newgroup all processes in group_a that 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, giving only some notions about intercommunicators.

Communicator accessors

- **MPI_COMM_SIZE**(comm,size,ierr)

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

- **MPI_COMM_RANK**(comm,rank,ierr)

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

- **MPI_COMM_COMPARE**(comm1,comm2,result,ierr)

Returns:

- MPI_IDENT if comm1 and comm2 are the same handle
- MPI_CONGRUENT if comm1 and comm2 have the same group attribute
- MPI_SIMILAR if the groups associated with comm1 and comm2 have the same members but in different rank order
- MPI_UNEQUAL otherwise

Communicator constructors

- **MPI_COMM_DUP**(comm, newcomm,ierr)

This returns a communicator newcomm identical to the communicator comm

- **MPI_COMM_CREATE**(comm, group, newcomm,ierr)

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

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

A practical example:

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

CALL MPI_COMM_GROUP (MPI_COMM_WORLD,wgroup,ierr)

define something..

CALL MPI_COMM_GROUP_EXCL(wgroup....., newgroup...)
CALL MPI_COMM_CREATE(MPI_COMM_WORLD,newgroup,newcomm,ierr)
```

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

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

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

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

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

Both process a and j are returned MPI_COMM_NULL

3 new groups are created

{i, c, d}

{k, b, e, g, h}

{f}

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

MPI_comm_split, for example, creates a communicator...

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

```
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

Destructors

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

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

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

Intercommunicators

Intercommunicators are associated with 2 groups of disjoint processes.

Intercommunicators are associated with a remote group and a local group

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

A communicator is either intra or inter, never both

MPI topologies

Virtual topologies

- Virtual topologies
- MPI supported topologies
- How to create a cartesian topology
- Cartesian mapping functions
- Cartesian partitioning

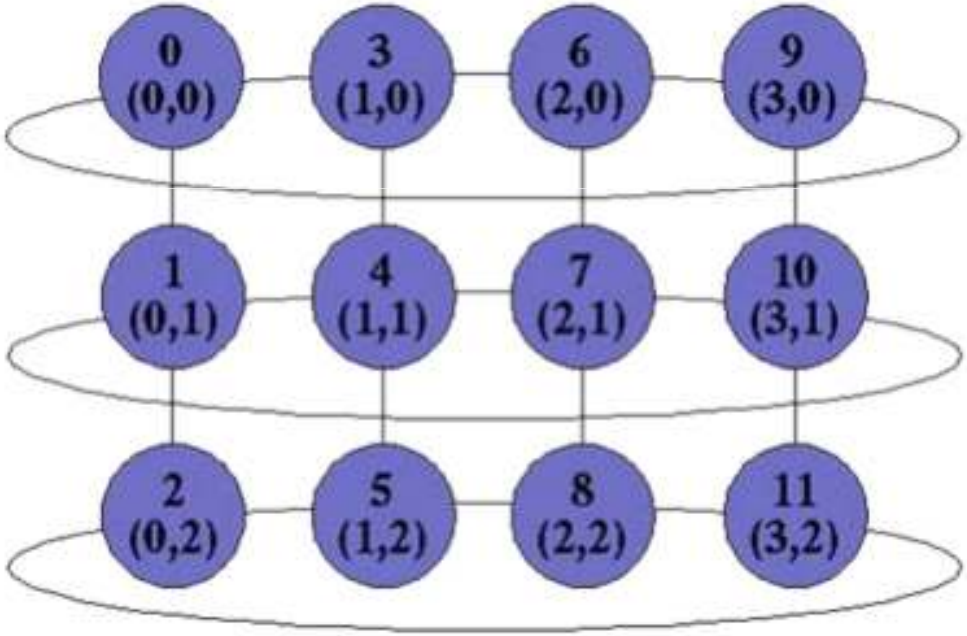
Why a virtual topology can be useful?

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies the writing of the code
- Can allow MPI to optimize communications

How to use a virtual topology?

- A new topology = a new communicator
- MPI provides some “mapping functions” to manage virtual topologies
- Mapping functions compute processor ranks, based on the topology name scheme

Cartesian topology on a 2D torus



MPI supports...

- Cartesian topologies
  - each process is connected to its neighbours in a virtual grid
  - Boundaries can be cyclic
  - Processes can be identified by cartesian coords
- Graph topologies

MPI_Cart_Create

```
MPI_Comm vu;
int dim[2], period[2], reorder;
dim[0]=4; dim[1]=3;
period[0]=TRUE; period[1]=FALSE;
reorder=TRUE;

MPI_Cart_create(MPI_COMM_WORLD,
2,dim,period,reorder,&vu)
```

Useful functions

Grid coords  ranks

MPI_Cart_rank

ranks  Grid coords

MPI_Cart_coords

Moving upwards, downwords, leftside, rightside...

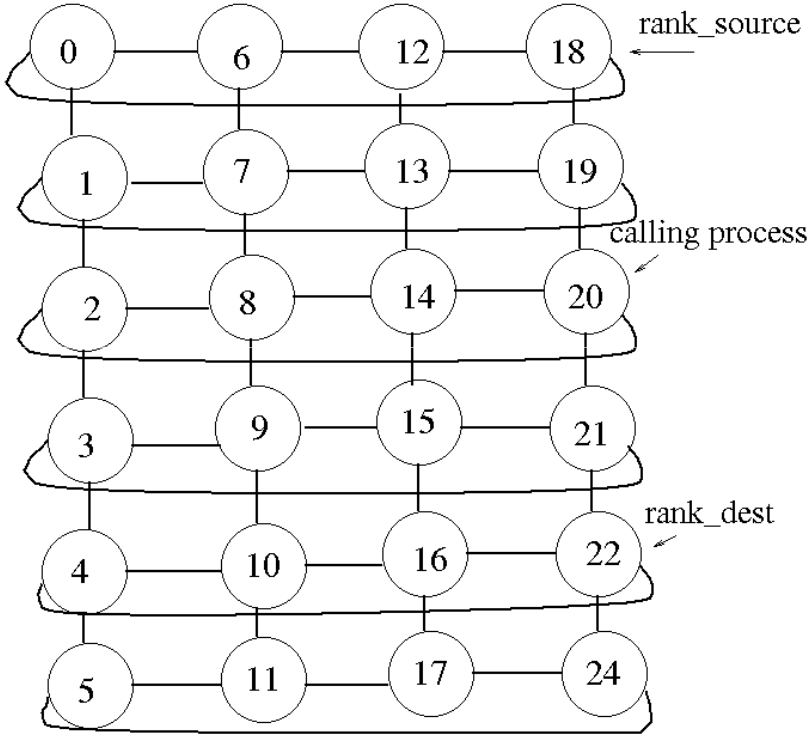
MPI_Cart _shift

```
#include<mpi.h>
int main(int argc, char *argv[]) {
int rank;
MPI_Comm vu;
int dim[2],period[2],reorder;
int coord[2],id;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
dim[0]=4; dim[1]=3;
period[0]=TRUE; period[1]=FALSE;
reorder=TRUE;
MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,
reorder,&vu);
if(rank==5){
MPI_Cart_coords(vu,rank,2,coord);
printf("P:%d My coordinates are %d %d\n",rank,
coord[0],coord[1]);
}
if(rank==0){
coord[0]=3; coord[1]=1;
MPI_Cart_rank(vu,coord,&id);
printf("The processor at position (%d, %d) has
rank %d\n",coord[0],coord[1],id);
}
```

MPI_Cart_shift

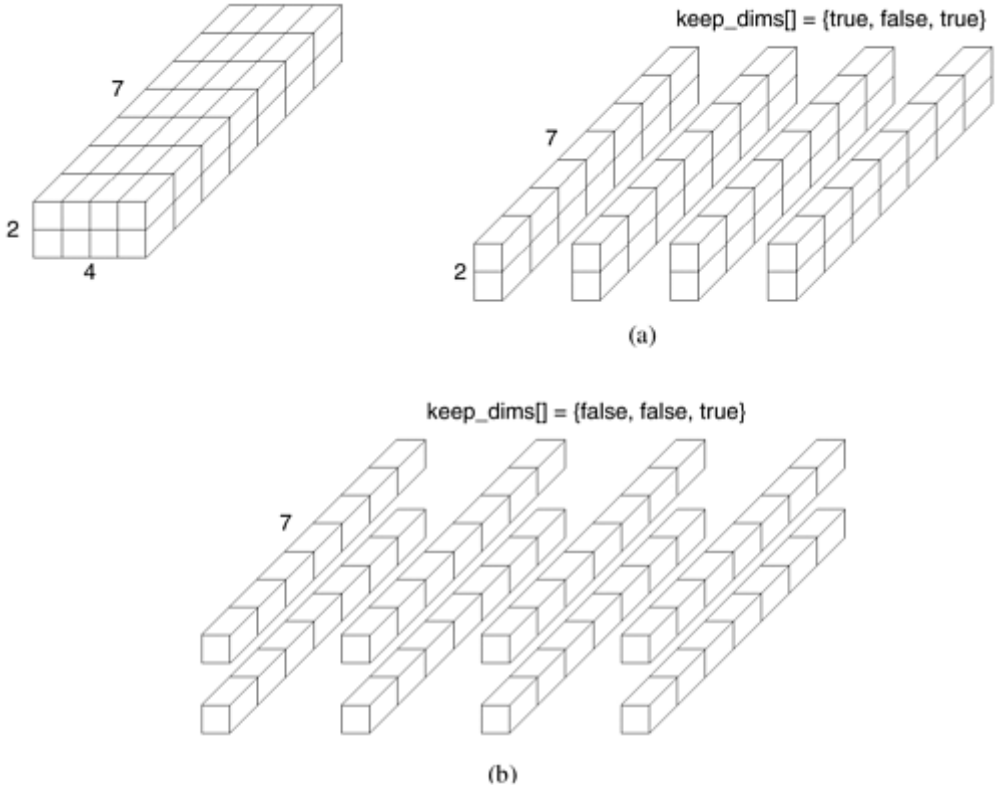
• It doesn't shift data actually: it returns the correct ranks for a shift that can be used in the subsequent communication call

• Arguments:
  - Direction: in which direction the shift should be made
  - disp: length of the shift
  - rank_source: where the calling process should receive a message from during the shift
  - rank_dest: where the calling process should send a message to during the shift


0
6
12
18
rank_source
1
7
13
19
calling process
2
8
14
20
3
9
15
21
rank_dest
4
10
16
22
5
11
17
24

Cartesian partitioning

- Often we want to do an operation on only a part of an existing cartesian topology
- Cut a grid up into “slices”
- A new communicator (i.e. a new cart. topology) is produced for each slide
- Each slice can perform its own collective communications




```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
                 MPI_Comm *newcomm)
```

MPI derived datatypes

We start from here...

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI derived datatypes

- Constructed from existing types
- Used in MPI communications to transfer high-level, extensive data entities

Examples:
- Sub arrays of “natural” array memory striding (row and columns..)
- C structures and Fortran common blocks
- Large set of general variables

Alternative to repeated sends of basic types
- Slow, clumsy and error prone

Life cycle of a derived datatype


```
MPI_Type_contiguous, MPI_Type_vector,
MPI_Type_struct, MPI_Type_indexed,
MPI_Type_hvector
```

```
MPI_Type_commit
```

```
MPI_Send(a,1,newtype....
```

```
MPI_Type_free
```

• A datatype is specified by its type map, like a stencil laid over memory.

• Displacements are offsets in bytes from the starting memory address of the desidered data

• MPI provides tools (MPI_data_extent) that can be used to get the size (in bytes) of datatypes

Contiguous datatypes (Fortran)

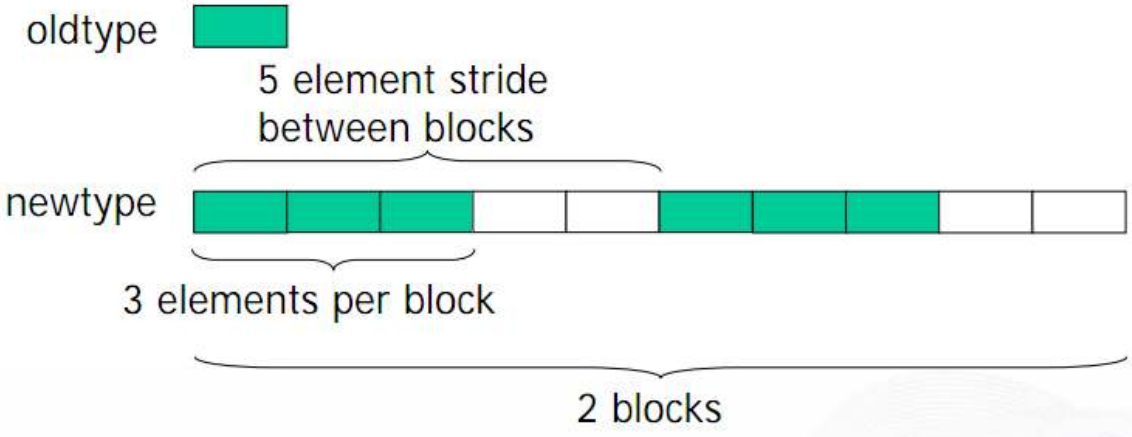
```
PROGRAM contiguous
C Run with four processes
INCLUDE 'mpif.h'
INTEGER err, rank, size
integer status(MPI_STATUS_SIZE)
integer x,y,z
common/point/x,y,z
integer ptype
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
CALL MPI_TYPE_CONTIGUOUS(3,MPI_INTEGER,ptype,err)
call MPI_TYPE_COMMIT(ptype,err)
print *,rank,size
if(rank.eq.3) then
x=15
y=23
z=6
CALL MPI_SEND(x,1,ptype,1,30,MPI_COMM_WORLD,err)
else if(rank.eq.1)then
CALL MPI_RECV(x,1,ptype,3,30,MPI_COMM_WORLD,status,err)
print *,'P:',rank,' coords are ',x,y,z
end if
CALL MPI_FINALIZE(err)
END
```

Contiguous datatypes (C)

```
#include <stdio.h>
#include<mpi.h>
/* Run with four processes */
int main(int argc, char *argv[]){
int rank;
MPI_Status status;
struct {
int x;
int y;
int z;
} point;
MPI_Datatype ptype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Type_contiguous(3,MPI_INT,&ptype);
MPI_Type_commit(&ptype);
if(rank==3){
point.x=15; point.y=23; point.z=6;
MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
} else if(rank==1) {
MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
printf("P:%d received coords are (%d,%d,%d)
\n",rank,point.x,point.y,point.z);
}
MPI_Finalize();
```

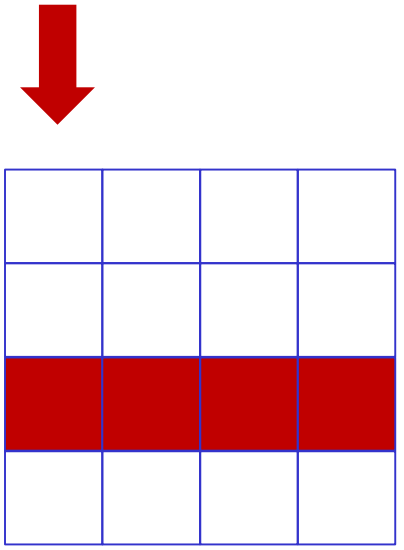
Vector datatypes

```
int MPI_Type_vector(  int count,
                      int blocklength,
                      int stride,
                      MPI_datatype oldtype,
                           MPI_datatype *newtype)
```



count = 2
stride = 5
blocklength = 3

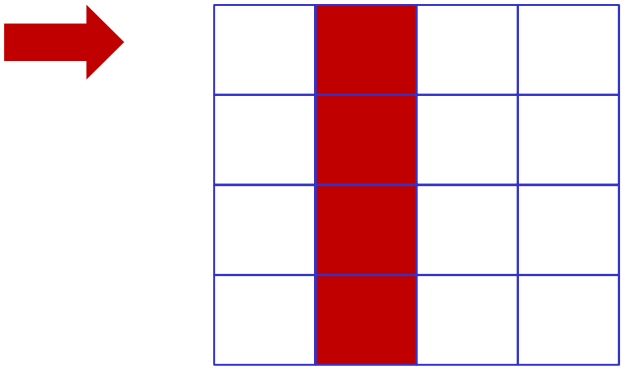
```
PROGRAM vector
C Run with four processes
INCLUDE 'mpif.h'
INTEGER err, rank, size
integer status(MPI_STATUS_SIZE)
real x(4,8)
integer rowtype
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
call MPI_TYPE_VECTOR(8,1,4,MPI_REAL,rowtype,err)
call MPI_TYPE_COMMIT(rowtype,err)
if(rank.eq.3) then
do i=1,4
do j=1,8
x(i,j)=10.0**i+j
end do
enddo
call MPI_SEND(x(2,1),1,rowtype,1,30,MPI_COMM_WORLD,err)
else if(rank.eq.1)then
call MPI_RECV(x(4,1),1,rowtype,3,30,MPI_COMM_WORLD,status,err)
print *,'P:',rank,' the 4th row of x is'
do i=1,8
print*,x(4,i)
end do
end if
CALL MPI_FINALIZE(ierr)
END
```



Vector datatypes example (Fortran)

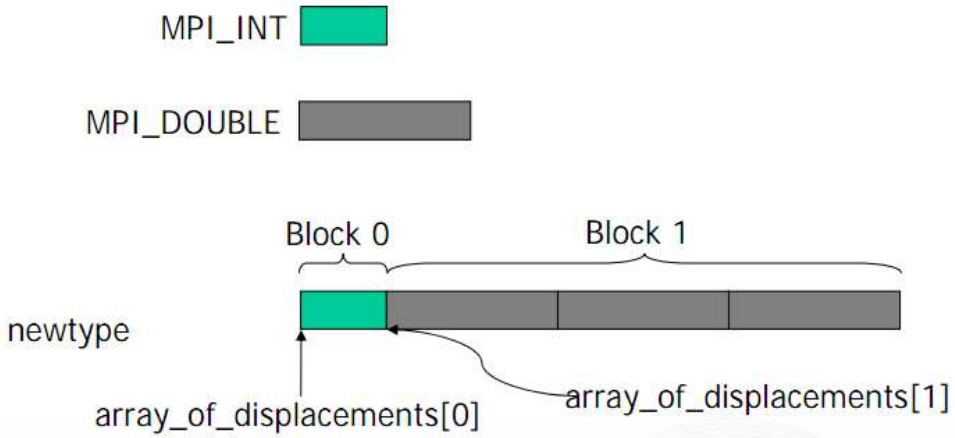
Vector datatypes example (C)

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
int rank,i,j;
MPI_Status status;
double x[4][8];
MPI_Datatype coltype;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
MPI_Type_commit(&coltype);
if(rank==3){
for(i=0;i<4;++i)
for(j=0;j<8;++j)
     x[i][j]=pow(10.0,i+1)+j;
     MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
} else if(rank==1) {
MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
for(i=0;i<4;++i)
     printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
}
MPI_Finalize();
}
```



Structured datatypes

```
int MPI_Type_struct(    int count,
                        int *array_of_blocklengths,
                        MPI_Aint *array_of_displs,
                        MPI_datatype *array_of types,
                        MPI_datatype *newtype)
```



count = 2
array of blocklength= {1,3}

array of types =
{MPI_INT, MPI_DOUBLE}

array of displs =
{0, extent[MPI_INT]}

Practical info

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

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

A.2 C Bindings

A.2.1 Point-to-Point Communication C Bindings

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

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

int MPI_Buffer_attach(void* buffer, int size)

int MPI_Buffer_detach(void* buffer_addr, int* size)

int MPI_Cancel(MPI_Request *request)

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

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

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

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

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

From C bindings to Fortran bindings

-In Fortran all function are transformed in subroutines and they don't return a type

-All functions have an addictional argument (ierror) of type integer

-All MPI datatypes in Fortran are defined as integers

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

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

Now we can seriously start to work...