Parallel Computing for Science & Engineering
Introduction to MPI, Derived Types, Communicators, Groups, and Topologies
Spring 2018

Instructors:
Charlie Dey TACC
Lars Koesterke, TACC

Sample Code

- Check out:

  http://www.mcs.anl.gov/research/projects/mpi/usingmpi

  https://www.msi.umn.edu/sites/default/files/mpi.pdf

Derived types

- MPI *Predefined Data Types* identify data types of the language.
- *Derived Types* identify structures within data storage (contiguous/noncontiguous and pure/mixed types).
- Derived Types are composed of predefined and/or Derived Types
  - Types can be created hierarchically at run-time
  - Avoids manually packing into a data array to send as MPI_BYTE
    - Eliminates packing operations (it takes time to pack)
    - Avoid using extra memory (packing requires packing array)
    - Avoids non-standard, user coded packing (packing can be error-prone)
  - better to create new types that match the data
    - New types can be used anywhere a predefined type can be used
- Packing and unpacking is automatic

Derived types
Three main classifications

- Contiguous Arrays (easy to use)
  - send contiguous blocks of the same datatype
- Noncontiguous Vectors (relatively easy to use)
  - send noncontiguous blocks of the same datatype
- Abstract (struct and indexed) types (more difficult)
  - send C or Fortran 90 structures

Derived types

- Elementary: MPI names for language types
- Contiguous: Array with stride of one
- Vector: Array separated by constant stride
- Hvector: Vector, with stride in bytes
- Indexed: Array of indices (like gatherv)
- Hindexed: Indexed, with displacements in bytes
- Struct: General mixed types (C structs etc.)
- Pack and Unpack

Derived types, how to use them

- Three step process
  - Define the type (e.g.)

    `MPI_Type_contiguous` for contiguous arrays
    `MPI_Type_vector` for noncontiguous arrays
    `MPI_Type_struct` for structures

  - Commit the type Tells MPI when to compile an internal representation

    `MPI_Type_commit(... my_type...)`
  - Use in normal communication calls

    `MPI_Send( data, count, my_type, dest, tag, comm …)`
- Free space when done:

  `MPI_Type_free`

Contiguous type

- `MPI_Type_contiguous`: creates a contiguous array of elementary or derived data types

```
real*8 a(5,5);
MPI_Datatype col_type;
integer mycomm=MPI_COMM_WORLD;
integer icol;
...
call MPI_Type_contiguous(5, MPI_REAL8,  col_type, ier);
call MPI_Type_commit( col_type, ier);
call MPI_Send( a(1,5), 2,col_type,  1,9,mycomm, ier);
...
call MPI_Type_free(col_type, ier);
```

MPI_REAL8 = MPI_DOUBLE_PRECISION

Contiguous type

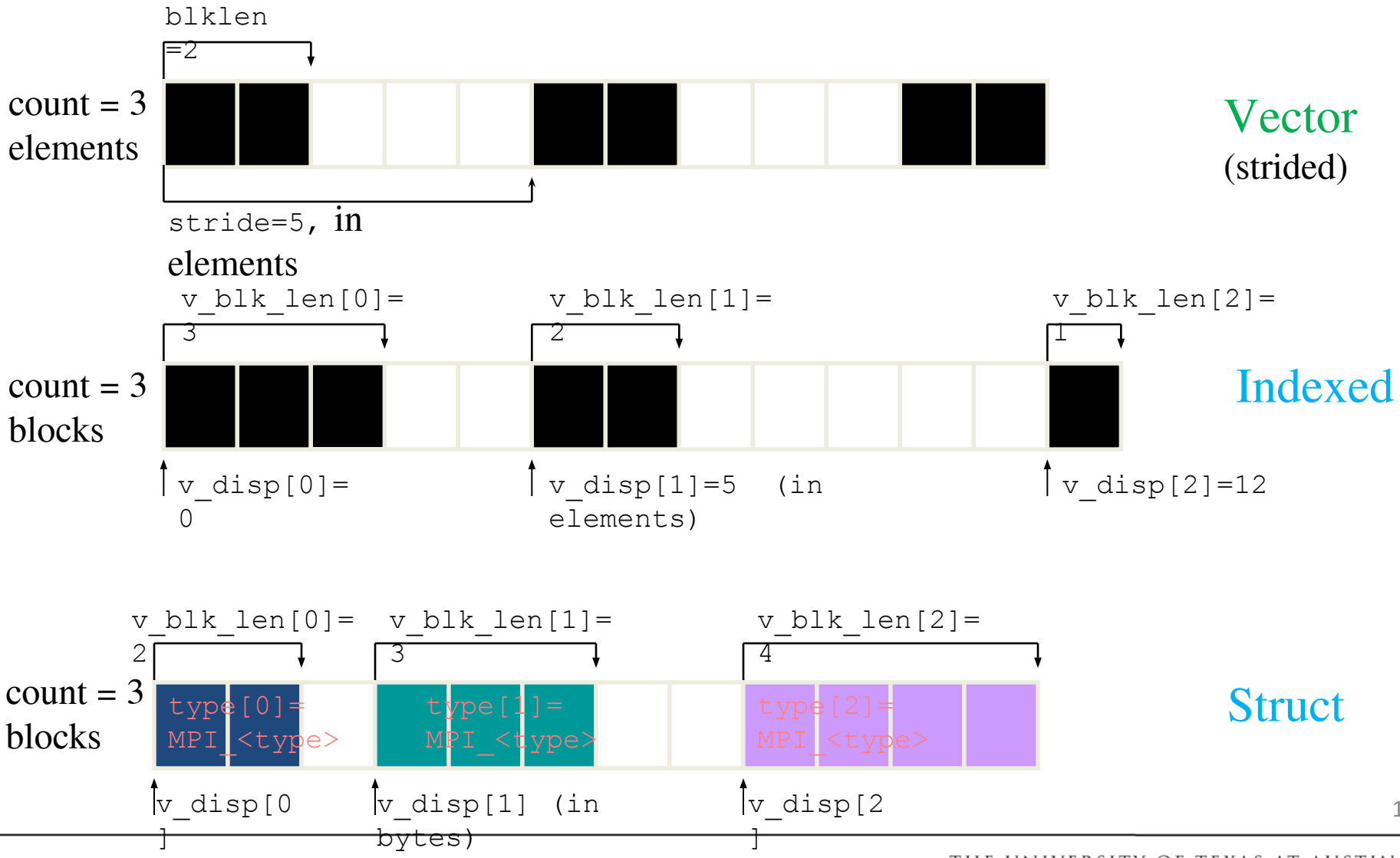
- `MPI_Type_contiguous`: creates a contiguous array of elementary or derived data types

```
double a[N][N];
MPI_Datatype row_type;
MPI_Comm mycomm=MPI_COMM_WORLD;
int irow, ier;
...
ier= MPI_Type_contiguous(N, MPI_DOUBLE, &row_type);
ier= MPI_Type_commit(&row_type);
ier= MPI_Send(&a[irow][0],1,row_type,  1,9,mycomm);
...
ier= MPI_Type_free(&row_type);
```

Contiguous type

```
int main(int argc, char *argv[])
{
    int myrank;
    MPI_Status status;
    MPI_Datatype type;
    int buffer[100];

    MPI_Init(&argc, &argv);

    MPI_Type_contiguous( 100, MPI_CHAR, &type );
    MPI_Type_commit(&type);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0)
    {
        MPI_Send(buffer, 1, type, 1, 123, MPI_COMM_WORLD);
    }
    else if (myrank == 1)
    {
        MPI_Recv(buffer, 1, type, 0, 123, MPI_COMM_WORLD, &status);
    }

    MPI_Finalize();
    return 0;
}
```

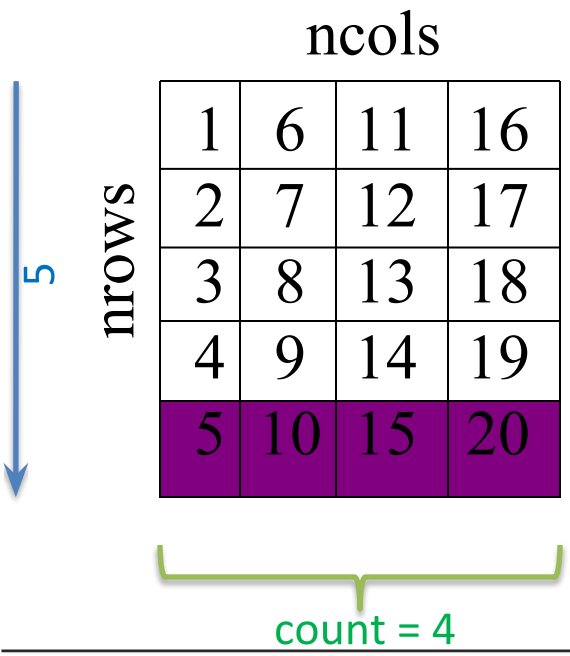
Derived types (arguments)



Vector Types

- `MPI_Type_vector`: create a type for non-contiguous vectors with constant stride

```
MPI_Type_vector(count,blklen,stride, oldtype,newtype, ierr)
```




```
integer row_type
...                                    cnt   blksz  stride
call MPI_Type_vector(ncols, 1, nrows,
                     MPI_REAL8, row_type, ierr)
call MPI_Type_commit(row_type, ierr)
```

call MPI_Send(a(5,1), 1, row_type…)

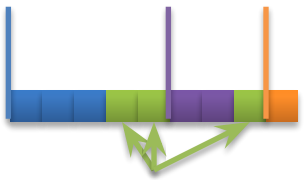
Vector type

```
 MPI_Datatype type, type2;

 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 MPI_Type_contiguous(3, MPI_INT, &type2);
 MPI_Type_commit(&type2);
 MPI_Type_vector(3, 2, 3, type2, &type);
 MPI_Type_commit(&type);

if (rank == 0)
{
    for (i=0; i<24; i++)
        buffer[i] = i;
    MPI_Send(buffer, 1, type, 1, 123, MPI_COMM_WORLD);
}

if (rank == 1)
{
    for (i=0; i<24; i++)
        buffer[i] = -1;
    MPI_Recv(buffer, 1, type, 0, 123, MPI_COMM_WORLD, &status);
    for (i=0; i<24; i++)
        printf("buffer[%d] = %d\n", i, buffer[i]);
    fflush(stdout);
}
```

Indexed Types

- `MPI_Type_indexed`: creates non-contiguous types with variable block sizes and displacements

```
MPI_Type_indexed(count,vblklen,vdispl, oldtype,&newtype)

MPI_Datatype newtype;
int          vblklen[3]  = {3,2,1};
int           vdispl[3]  = {0,5,8};
MPI_Type_indexed(3,vblklen,vdispl, MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
```

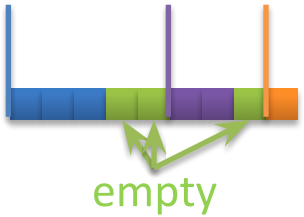


Indexed Types

- `MPI_Type_indexed`: creates non-contiguous types with variable block sizes and displacements

```
MPI_Type_indexed(count,vblklen,vdispl, oldtype,newtype,ier)
```

```
integer ::   newtype;
integer :: vblklen(3) = (/3,2,1/);
integer ::  vdispl(3) = (/0,5,8/);
call MPI_Type_indexed(3,vblklen,vdispl, MPI_REAL8, newtype, ier);
call MPI_Type_commit(newtype, ier);
```



Struct Types

- `MPI_Type_struct`: heterogeneous elements & arbitrary locations

```
MPI_Type_struct(count,vblklen,vdispl,vtypes, newtype)
MPI_Type_commit(newtype)
```

```
typedef struct {double val; int i,j;} xyz;

int                 vblklen[2] =  {1,2};
MPI_Aint             vdispl[2] =  {0,sizeof(double)};
MPI_Datatype          vtype[2] =  {MPI_DOUBLE, MPI_INT};

    MPI_Type_struct(2,vblklen,vdispl,vtype,&newtype);
    MPI_Type_commit(&newtype);
```

Struct Types

- `MPI_Type_struct`: heterogeneous elements & arbitrary locations

```
MPI_Type_struct(count,vblklen,vdispl,vtypes, newtype, ier)
MPI_Type_commit(newtype, ier)
```

```
type xyz
   sequence; real*8 val; integer i, j;
end type xyz
integer                  :: vblklen(2) = (/1,2/);
integer(MPI_ADDRESS_KIND) :: vdispl(2) = (/0,8/);
integer                  :: vtype(2) = (/MPI_REAL8, MPI_INTEGER/);

call MPI_Type_struct(2,vblklen,vdispl,vtype, newtype,ier);
call MPI_Type_commit( newtype, ier);
```

Struct Types

- `MPI_Type_struct`: heterogeneous elements & arbitrary locations

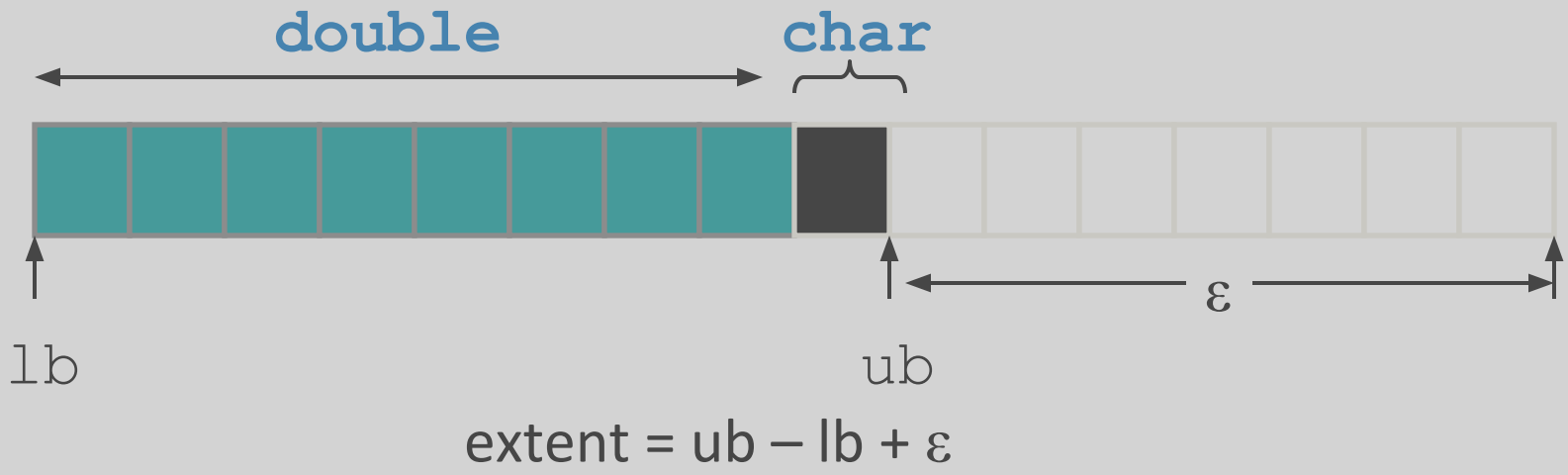
```
MPI_Type_struct(count,vblklen,vdispl,vtypes, newtype, ier)
MPI_Type_commit(newtype, ier)
```

```
type xyz
   sequence; real*8 val; integer i, j;
end type xyz
integer                :: vblklen(2) = (/1,2/);
integer                :: vdispl(2) = (/0,8/);
integer                :: vtype(2) = (/MPI_REAL8, MPI_INTEGER/);

call MPI_Type_struct(2,vblklen,vdispl,vtype, newtype,ier);
call MPI_Type_commit( newtype, ier);
```

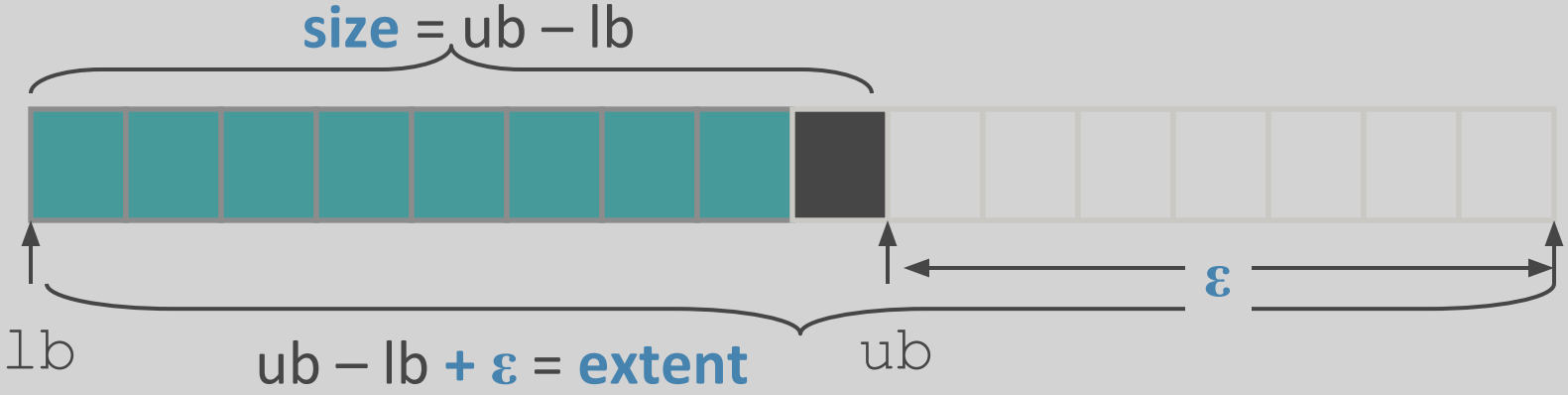
Alignment in Derived Structures

```
int          v_blk_len[2] = {1,1};
MPI_Aint       v_disp[2] = {0,8};
MPI_Datatype v_types[2] = {MPI_DOUBLE, MPI_CHAR};
MPI_Datatype newtype;

MPI_Type_struct(2, v_blk_len, v_disp, v_types, &newtype);
```

But, may need an array of these!



Alignment in Derived Structures




```
C   MPI_Type_extent(datatype, &extent);
F90 MPI_Type_extent(datatype, iextent,
ierr)

C   MPI_Type_ub(datatype, &displ);
F90 MPI_Type_ub(datatype, idispl, ierr)
C   MPI_Type_lb(datatype, &displ);
F90 MPI_Type_lb(datatype, idispl, ierr)

C   MPI_Type_size(datatype, &bytes);
F90 MPI_Type_size(datatype, ibytes, ierr)
```

Communicators

- A communicator is a "context" for communicating only among a group of tasks.
- `MPI_COMM_WORLD` is the default communicator and consists of all tasks.
- Communication is isolated to context of the group– i.e. no messages from other contexts are "seen".

Why Communicators?

- Isolate communication to a small number of processors
- Useful for creating libraries
- Collective communication between subgroups (in lieu of all tasks) can drastically reduce communication costs if only some need to participate
- Useful for communicating with "nearest neighbors"

Groups

A new communication group can only be created from a previously defined group. A group must also have a context for communication and, therefore, must have a communicator created for it. The basic steps to form a group are:

- Obtain a complete set of task IDs from a communicator
  `MPI_Comm_group`.
- Create a group as a subset of the complete set by
  `MPI_Group_excl`, `MPI_Group_incl`, `...`
- Create the new communicator for group (subset) using
  `MPI_Comm_create`.

Communicators

Routine	Function
MPI_Comm_group	returns group reference of a communicator
MPI_Group_incl	forms new group from inclusion list
MPI_Group_excl	forms new group from exclusion list
MPI_Group_{union, intersection, difference}	Forms new group from union, intersection, or difference of 2 groups.
MPI_Comm_create	creates communicator from a group reference

Creating Communicators for Groups

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXEVEN 128
main(int argc, char **argv){
    int npes, irank,   ierr;
    int neven,iegid,   iogid,   i,  iranks[MAXEVEN];

    MPI_Group iegroup, iogroup, iwgroup;
    MPI_Comm  iecomm,  iocomm;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &npes);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &irank);

                        /* Extract group from World Comm. */

    ierr =  MPI_Comm_group(MPI_COMM_WORLD, &iwgroup);
```

Creating Communicators for Groups

```
                        /* Make list of even ranks. */

neven = (npes+1)/2;
if(neven > MAXEVEN) exit(1);
for(i=0; i < npes; i+=2) iranks[i/2] = i;

                        /* Form even and odd groups. */

ierr = MPI_Group_incl(iwgroup,neven,iranks, &iegroup);
ierr = MPI_Group_excl(iwgroup,neven,iranks, &iogroup);

ierr = MPI_Comm_create(MPI_COMM_WORLD,iegroup,&iecomm);
ierr = MPI_Comm_create(MPI_COMM_WORLD,iogroup,&iocomm);
```

Creating Communicators for Groups

```
  ierr = MPI_Group_rank(iegroup, &iegid);

     if(iegid != MPI_UNDEFINED) {
       printf("PE: %d, id %d of even group.\n", irank,iegid);
     }
     else {
       ierr =  MPI_Group_rank(iogroup, &iogid);
       printf("PE: %d, id %d of odd  group.\n", irank,iogid);
     }
  MPI_Comm_free( iecomm ); MPI_Comm_free( iocomm );
  MPI_Group_free(iegroup); MPI_Group_free(iogroup);

  ierr = MPI_Finalize();
}
```

MPI_Comm_split

- Provides a short cut method to create a collection of communicators
- All processors with the "same color" will be in the same communicator
- Index controls relative rank in group
- Fortran

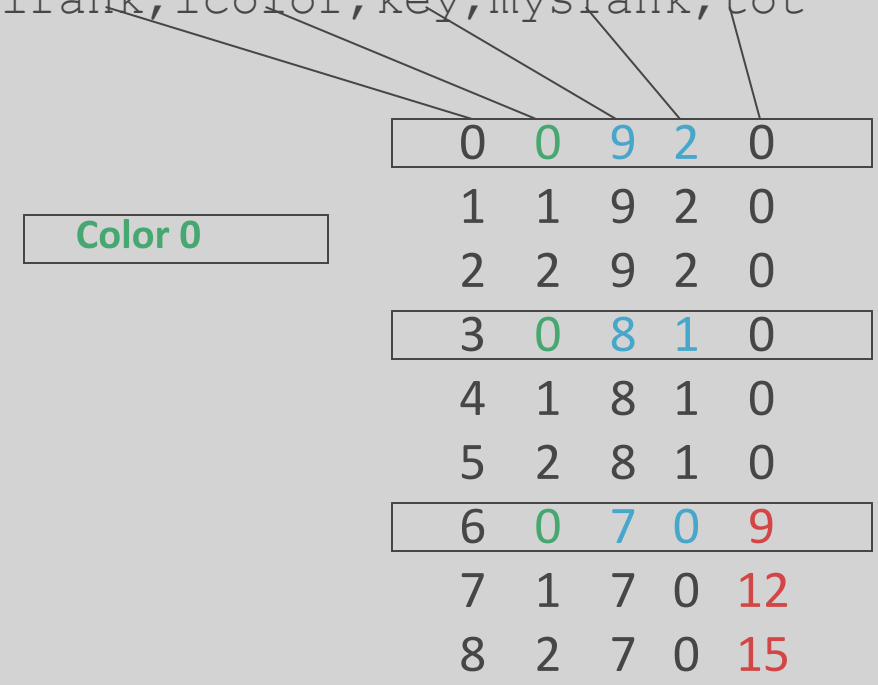
```
MPI_Comm_split(OLD_COMM, color, index,  NEW_COMM, ierr)
```

- C

```
MPI_Comm_split(OLD_COMM, color, index, &NEW_COMM)
```

MPI_Comm_split

```
call MPI_Comm_rank(MPI_COMM_WORLD,irank,ierr)
icolor = modulo(irank,3)
key    = npes - irank/3   ! reverse the ordering

call MPI_Comm_split(MPI_COMM_WORLD, icolor, key, newcom, ierr)
call MPI_Comm_rank(newcom,mysrank,ierr)

psum  = irank
call MPI_Reduce(psum,tot,1,MPI_INTEGER,MPI_SUM,0,newcom,ierr)
print*, irank,icolor,key,mysrank,tot
```



Colors are 0, 1 and 2
Keys are 9, 8 and 7
Lowest keys are roots

(Virtual) Topologies

- In terms of MPI, a virtual topology describes a **mapping** and **ordering** of MPI processes into a geometric shape.
- The two main types of topology supported by MPI are **Cartesian**(grid) and Graph.
- MPI topologies are **virtual** – there may be no relation between the physical structure of parallel machine and the process topology.
- Virtual topologies are **built upon MPI communicator and groups**.
- Must be ***built* by the application developer**.
- Useful for applications with **specific communication pattern**.
- A particular **implementation may optimize process mapping** based on the physical characteristics of a given parallel machine.

Topologies

- Use the MPI library for common grid topologies (local functions)
- A *topology* maps each rank onto a set of N-tuples (N dimensions)

  E.g. For a 2x2 Cartesian grid:

  Ranks:{0, 1, 2, 3}->{(0,0), (0,1), (1,0), (1,1)} tuples

  (row-major in ranks)
- Cartesian Map operations:

MPI_Cart_create	Creates map (ranks → coordinates).
MPI_Cart_get	Returns info created in MPI_Cart_create.
MPI_Cart_coords	Returns coordinates from rank.
MPI_Cart_rank	Returns rank from coordinates.
MPI_Cart_shift	Returns rank of neighbors in nth dimension.

- **graph** constructors go beyond the *N*-dimensional rectilinear mapping of the Cartesian topology (MPI_Graph_create)

Note: the virtual topology does not necessarily map the hardware processor grid to the process grid in the most efficient manner.

Topologies

Use Case:

1. Built 2-/3-D Cartesian Grid
2. Use shift operator to get neighbor rank
   for destination and source
   in x, y and z directions

Topologies

```
MPI_Cart_create( icomm,     idim, ishape, lperiod, lreorder , icartcom)
MPI_Cart_rank   ( icartcom, icoords,  irank)
MPI_Cart_coords( icartcom, irank, idim,  icoords)
MPI_Cart_get    ( icartcom, idim,  ishape, lperiod, icoords)
```

icomm	idim, ishape	lperiod	lreorder	icartcom
communicator	number of dims cart. grid shape	periodic? (array)	allowed to reorder (logical)	returns new communicator
icartcom	**icoords**	**irank**		
cartesian communicator	coordinate array for rank	returns rank		
icartcom	**irank**	**idim**	**icoords**	
cartesian communicator	rank	dimension of topology	returns coordinates	
icartcom	**idim**	**ishape**	**lperiod**	**icoords**
communicator	dimension of topology	returns shape of topology	returns periodicity	returns coordinates

Topologies (Shift)

Assume 3x3 grid

```
MPI_Cart_create( MPI_COMM_WORLD, 2, shape, true, 0, cartcom)
```

(3,3)

C

```
MPI_Cart_Shift(cartcomm, direct, disp, &rank_src, &rank_dst)
```

Fortran

```
MPI_Cart_Shift(cartcomm, direct, disp,  rank_src,  rank_dst,ierr)
```

Parameters

cartcom = communicator with Cartesian structure
direct = coordinate dimension of shift
disp = >0 = upwards shift, <0 = downwards shift
end-off/circular shift (see lperiod of MPI_Cart_create)
rank_src = output: rank of source process
rank_dst = output: rank of destination process

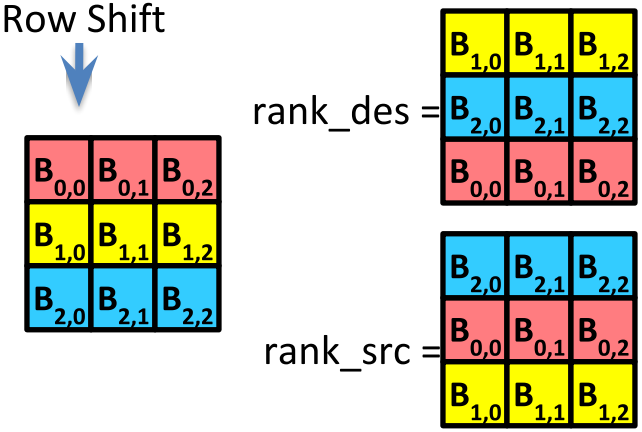
Topology Illustrations

Rank map onto 2-D Cartesion Topology

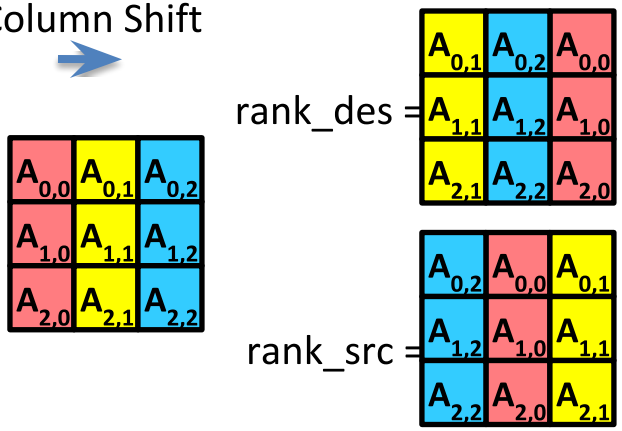
0 (0,0)	1 (0,1)	2 (0,2)
3 (1,0)	4 (1,1)	5 (1,2)
6 (2,0)	7 (2,1)	8 (2,2)

Column/Row Shift (reference)

Periodic Displacement of 1 in Dimension "0"
Row Shift



Periodic Displacement of 1 in Dimension "1"
Column Shift



C Example

```
#include <mpi.h>
#include <stdio.h>
#define NP 3

main(int argc, char **argv){
   int npes,   mype, ierr,  myrow, mycol;
   int isrca, isrcb, idesa, idesb;
   MPI_Comm IWCOMM = MPI_COMM_WORLD, igcomm;
/*                                   MPI Cartesian Grid information */
   int  ivdim[2] = {NP,NP}, ivper[2]={1,1};
   int ivdimx[2],           ivperx[2], mygrids[2];
...
/*  Create Cartesian Grid and extract information */

    ierr= MPI_Cart_create(IWCOMM,2,ivdim ,ivper, 0,&igcomm);
    ierr= MPI_Cart_get(   igcomm,2,ivdimx,ivperx, mygrids);
    ierr= MPI_Cart_shift( igcomm,1,1, &isrca,&idesa);
    ierr= MPI_Cart_shift( igcomm,0,1, &isrcb,&idesb);
```

Fortran Example

```
 integer,parameter :: NP=3
 logical, dimension(2)    :: lvper=(/.true.,.true./), lvperx
 integer, dimension(2)    :: ivdim=(/    NP,    NP/), ivdimx
 integer, dimension(2)    :: mygrid
...
 call mpi_cart_create(iwcomm,2,ivdim ,lvper, .false.,igcomm,ir)
 call mpi_cart_get(   igcomm,2,ivdimx,lvperx, mygrid, ir)
 call mpi_cart_shift( igcomm,1,1, isrca,idesa, ierr)
 call mpi_cart_shift( igcomm,0,1, isrcb,idesb, ierr)
myrow=mygrid(1);mycol=mygrid(2);mype=rank

 print*,'A:',isrca,')- ',mype,'[',myrow,',',mycol,'] ->',idesa,
& '      B:',isrcb,')- ',mype,'[',myrow,',',mycol,'] ->',idesb
```

Will receive from

Who I am

Will send to

column shift @[0,0]

row shift @[0,0]``

```
 A: 2)-  0 [ 0 , 0 ] -> 1        B: 6 )-  0 [ 0 , 0 ] -> 3
```

Generic Example: Send “a” blocks down/up, and “b” blocks right/left.

```
MPI_CART_SHIFT(cartcomm, 0, 1, UP,   DOWN   )
MPI_CART_SHIFT(cartcomm, 1, 1, LEFT, RIGHT  )

…

MPI_ISEND(a1, N,MPI_INTEGER, DOWN,  1,MPI_COMM_WORLD,reqs1  )
MPI_IRECV(a2, N,MPI_INTEGER, UP,    1,MPI_COMM_WORLD,reqa2  )

…

MPI_ISEND(b1, N,MPI_INTEGER, RIGHT, 2,MPI_COMM_WORLD,reqb1  )
MPI_IRECV(b2, N,MPI_INTEGER, LEFT,  2,MPI_COMM_WORLD,reqb2  )
```