Parallel Computing for Science & Engineering Spring 2013: MPI datatypes and communicators

Instructors:

Victor Eijkhout, Research Scientist, TACC

Kent Milfeld, Research Associate, TACC

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

MPI Data Types

- MPI data types are used in data communication operation.
- MPI has many different predefined data types
  – Defined to match C/Fortran data types
- MPI handles endianness conversion (though a mixed architecture system is rare)
- Packed/opaque types– User Defined Types can be made to handle C/F90 structures

MPI Predefined Data Types in C

C MPI Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long 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 Predefined Data Types in F90

MPI Parameter	F90 type
MPI_INTEGER	Integer
MPI_REAL	Real
MPI_DOUBLE_PRECISION	Double Precision
MPI_COMPLEX	Complex
MPI_LOGICAL	Logical
MPI_CHARACTER	Character
MPI_BYTE	Raw Byte (no conversion)
MPI_PACKED	MPI calls pack/unpack

Derived types

- MPI Predefined Data Types identify data types of the language.
- User 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 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 (C)

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

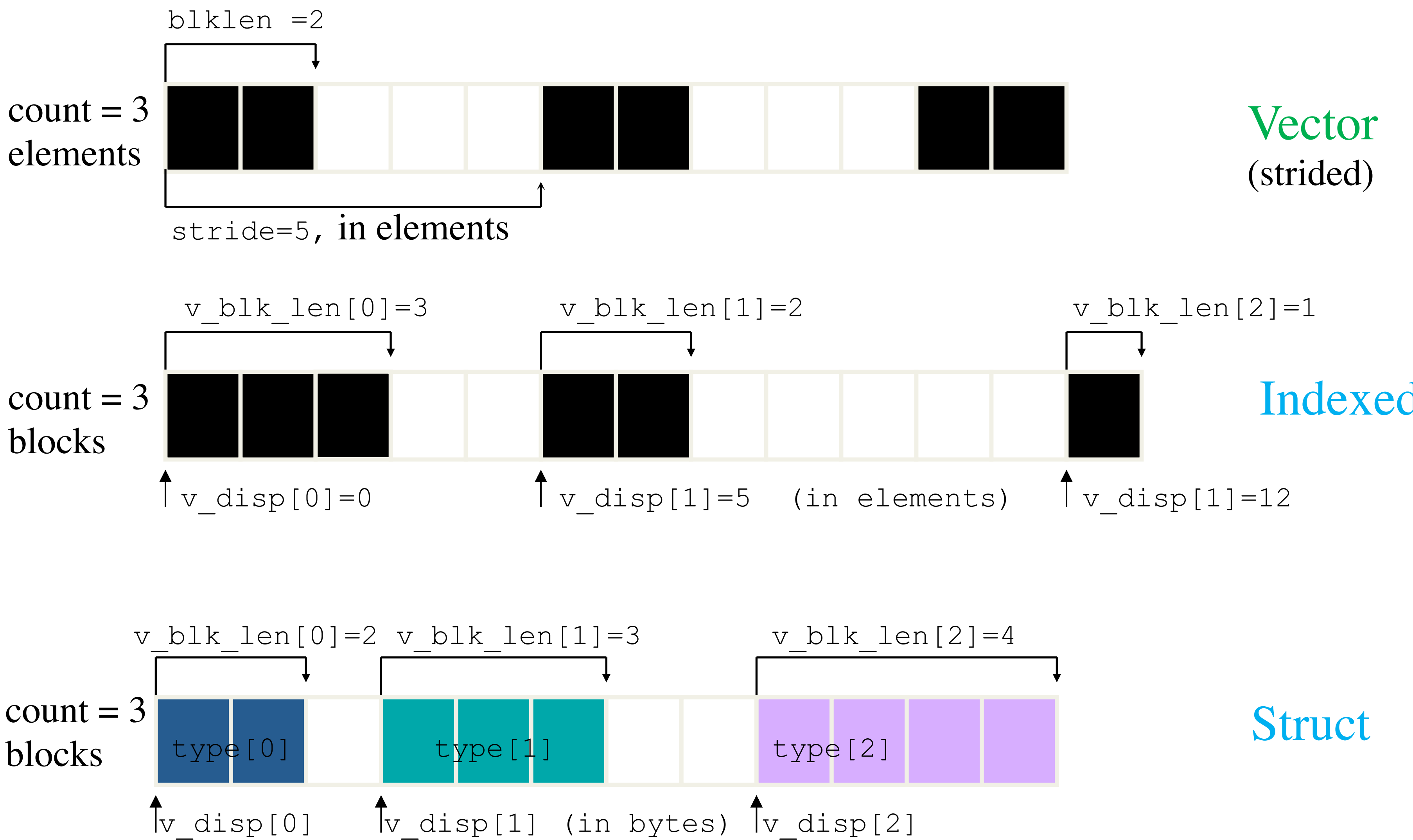
```
double a[N][N];
MPI_Datatype row_type;
MPI_Comm mycomm=MPI_COMM_WORLD;
int irow, ier;
...
ier= MPI_Type_contiguous(N, MPI_DOUBLE, &row_type);
ier= MPI_Type_commit(&row_type);
ier= MPI_Send(&a[irow][0],1,row_type,  1,9,mycomm);
...
ier= MPI_Type_free(&row_type);
```

Contiguous type (F)

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

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

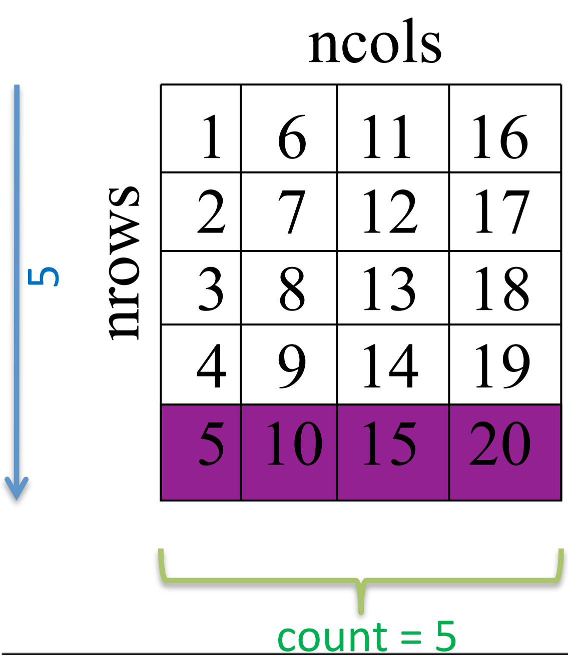
Derived types (arguments)



Vector Types

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

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



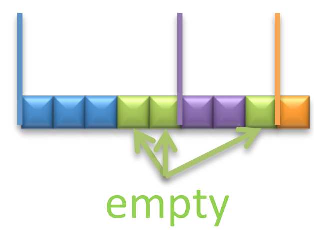
cnt blksz stride

```
integer row_type

...

call MPI_Type_vector(ncols, 1, nrows,
             MPI_REAL8, row_type, ierr)

call MPI_Type_commit(row_type, ierr)
```

Indexed Types

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

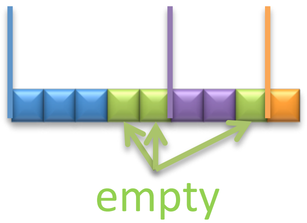
```
MPI_Type_indexed(count,vblklen,vdispl, oldtype,&newtype)

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_create_struct`: heterogeneous elements & arbitrary locations

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

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"

Communicators and libraries

- Sharing communicator between main and library:
- Library can receive messages from the main program. Oops.

```
main (){
  if (me==0) MPI_Send( ..to 1.. ,
                 MPI_COMM_WORLD )
  library_call()
  if (me==1) MPI_Recv( .. from 0 .. ,
                 MPI_COMM_WORLD )
}

void library_call() {
  other = me-1;
  MPI_Recv( .. from other .. ,
                 MPI_COMM_WORLD )
}
```

Duplicate communicators

- Duplicate communicator with MPI_Comm_dup:
- Same group of processors, but different context: no confusion possible.

```
main (){
  if (me==0) MPI_Send( ..to 1..,
                 MPI_COMM_WORLD )
  library_call(MPI_COMM_WORLD)
  if (me==1) MPI_Recv( .. from 0 .. ,
                 MPI_COMM_WORLD )
}

void library_call(comm) {
  MPI_Comm my_comm = // copy of comm
  other = me-1;
  MPI_Recv( .. from other ..,
            my_comm )
}
```

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   ! 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
```

One group

0	0	9	2	0
1	1	8	2	0
2	2	7	2	0
3	0	6	1	0
4	1	5	1	0
5	2	4	1	0
6	0	3	0	9
7	1	2	0	12
8	2	1	0	15

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

Topologies

- Use the MPI library for common grid topologies (local functions)
- A *topology* maps process-ranks onto a set of N-tuples.
- E.g. {0, 1, 2, 3}->{(0,0), (0,1), (1,0), (1,1)} (row-major in ranks)
- Cartesian Maps (arbitrary number of dimensions):

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 Nth neighbor's coords.

- **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.

(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 ***programmed*** **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.
- Can be used within an intra-communicator; cannot be added to inter-communicators.

Topologies

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

Topologies (Shift)

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 = dimension for end-off/circular shift (see lperiod of MPI_Cart_create)
rank_src = rank of source process
rank_dest = rank of destination process

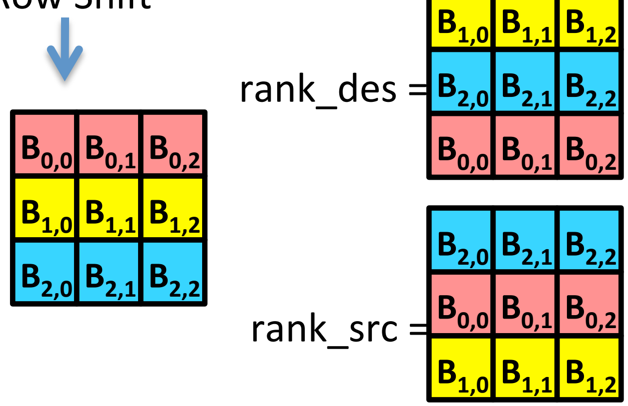
Topology Illustrations

Rank map onto 2-D Cartesion Topology

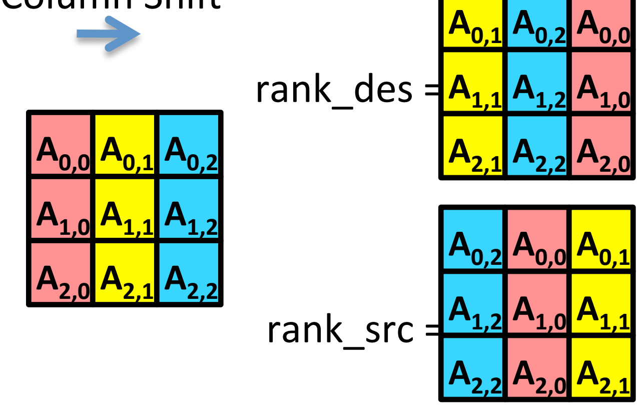
0 (0,0)	1 (0,1)	2 (0,2)
3 (1,0)	4 (1,1)	5 (1,2)
6 (2,0)	7 (2,1)	8 (2,2)

Column/Row Shift (reference)

Periodic Displacement of 1 in Dimension “0”
Row Shift



Periodic Displacement of 1 in Dimension “1”
Column Shift



C Example

changed

```
#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);
```

changed

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)

 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
```

changed

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   )
```