Introduction to MPI: Part III

Pawel Pomorski, University of Waterloo, SHARCNET
ppomorsk@sharcnet.ca

August 3, 2016

Summary of Part I:

To write working MPI (Message Passing Interface) parallel programs you only need:

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
- MPI_Finalize

See SHARCNET YouTube channel for recording of part I

Summary of Part II:

More advanced features:

- Collective communications (broadcast, reduce etc.)
- Non-blocking communications (overlap communication with computation)

See SHARCNET YouTube channel for recording of part II

Outline

In this talk will introduce two more advanced MPI concepts:

- user defined data types
- user defined communicators and topologies

Communication overhead

- There is always a fixed, non-zero cost to send any message, no matter how small
- To avoid this cost, it is better to combine multiple smaller messages into one larger one

Simplest grouping

- Use count parameter, as discussed in previous lectures, in conventional MPI_Send and MPI_Recv.
- This will work fine for contiguous, identical, and conventional items of data (integer, float etc.)

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

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Derived Types

- Want to send in one communication real numbers a and b, and integer n. Let's try:

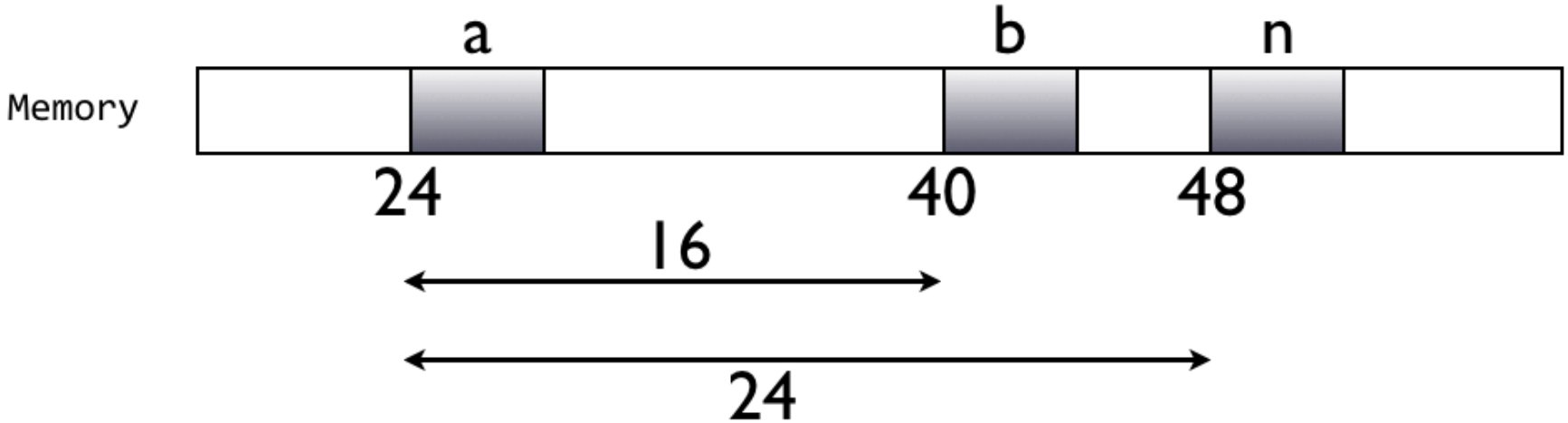
```
typedef struct {
float a;
float b;
int   n;
} INDATA_T
/* ... */
INDATA_T indata;
/* ... */
MPI_Bcast(&indata, 1, INDATA_T,0,MPI_COMM_WORLD);
```

- Will not work. Arguments to functions must be variables, not defined types.
- Need to define a type that can be used as a function argument, i.e. a type that can be stored in a variable. MPI provides just such a type: MPI_Datatype

Communicating non-contiguous data

Need to send data preserving arrangement in memory

Variable	Address	Contents
a	24	0.0
b	40	1.0
n	48	1024



Information needed to send the a,b,n data from the example

- 1. There are three elements to be transmitted
- 2. Their types are float, float and integer.
- 3. Their displacements are 0, 16 and 24 from beginning of messages.
- 4. The beginning of the message has address &a
- New datatype will store information from section 1,2 and 3.
- Each of the receiving processes can determine exactly where the data should be received.
- The principle behind MPI's derived types is to provide all the information except the address of the beginning of message in a new MPI datatype.
- Then when program calls MPI_Send, MPI_Recv etc, it simply provides the address of the first element and the communications system can determine exactly what needs to be sent and received.

MPI_Type_struct

```
int MPI_Type_struct(int count, int *blocklens,
        MPI_Aint *indices, MPI_Datatype *old_types,
        MPI_Datatype *newtype )
```

- count - number of elements (or blocks, each containing an array of contiguous elements)
- blocklens - array with number of elements in each block
- indices - byte displacements of each block. MPI_Aint is a special type for storing addresses. Can use MPI_Address function to obtain this.
- old_types - types of elements in each block. These can be datatypes created by previous MPI_Type_struct calls
- newtype - new datatype. Must call MPI_Type_commit on it to activate it.

In our previous example, all blocklens entries would be 1, even though the first two had the same type, since all three elements were not contiguous in memory.

Build derived data type

```
void Build_derived_type(
        float*          a_ptr           /* in  */,
        float*          b_ptr           /* in  */,
        int*            n_ptr           /* in  */,
        MPI_Datatype*   mesg_mpi_t_ptr  /* out */) {

    int block_lengths[3];
    MPI_Aint displacements[3];
    MPI_Datatype typelist[3];

    MPI_Aint start_address;
    MPI_Aint address;
    block_lengths[0]=block_lengths[1]=block_lengths[2]=1;
```

Build derived data type cont

```
typelist[0] = MPI_FLOAT;
typelist[1] = MPI_FLOAT;
typelist[2] = MPI_INT;
/* First element, a, is at displacement 0      */
displacements[0] = 0;
/* Calculate displacements relative to a */
MPI_Address(a_ptr, &start_address);

MPI_Address(b_ptr, &address);
displacements[1] = address - start_address;

MPI_Address(n_ptr, &address);
displacements[2] = address - start_address;

MPI_Type_struct(3, block_lengths, displacements,
    typelist, mesg_mpi_t_ptr);
MPI_Type_commit(mesg_mpi_t_ptr);}          }
```

Get_data routine

```
void Get_data3(
          float*  a_ptr     /* out */,
          float*  b_ptr     /* out */,
          int*    n_ptr     /* out */,
          int     my_rank   /* in  */) {
    MPI_Datatype  mesg_mpi_t; /* MPI type corresponding */
                              /* to 2 floats and an int */

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }

    Build_derived_type(a_ptr, b_ptr, n_ptr, &mesg_mpi_t);
    MPI_Bcast(a_ptr, 1, mesg_mpi_t, 0, MPI_COMM_WORLD);
}
```

Other derived datatype constructors

MPI_Type_struct is the most general datatype constructor in MPI

- MPI_Type_contiguous - builds a derived datatype whose elements are contiguous entries in an array
- MPI_Type_vector - does this for equally spaced entries in an array
- MPI_Type_indexed - does this for arbirary entries of an array

Matrix Example

C language stores two dimensional arrays in row-major order. This means A[2][3] is preceded by A[2][2] and followed by A[2][4].

Say we want to send, for example, third row of A from process 0 to process 1, this is easy

```
float A[10][10]; /* define 10 by 10 matrix */
if (myrank == 0){
MPI_Send(&(A[2][0]),10, MPI_FLOAT,1,0,MPI_COMM_WORLD)
} else { /*my_rank = 1 */
MPI_Recv(&(A[2][0]), 10, MPI_FLOAT,0 ,0 ,MPI_COMM_WORLD,
 &status);
}
```

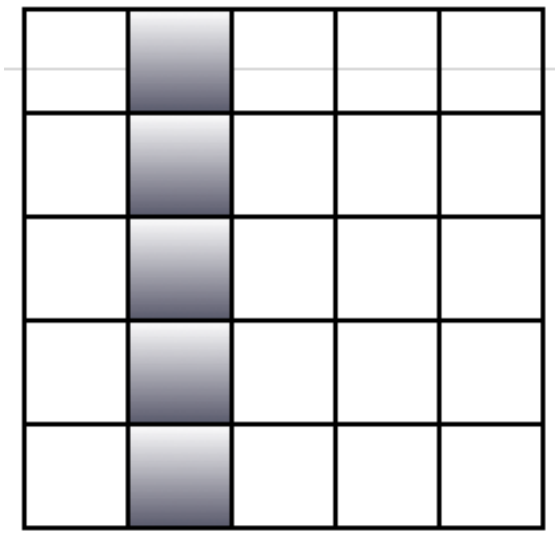
Could have used MPI_Type_Contiguous, but in this case MPI_Send and MPI_Recv are sufficient.

BUT, if we want to send the third column of A, then those entries no longer form a continuous block in memory, hence a single MPI_Send and MPI_Recv pair is no longer sufficient.

MPI_Type_vector

```
int MPI_Type_vector( int count, int blocklen, int stride,
        MPI_Datatype old_type, MPI_Datatype *newtype )
```

- count - number of blocks
- blocklen - number of elements in each block
- stride - number of elements between start of each block
- old_type - old datatype
- new_type - new datatype



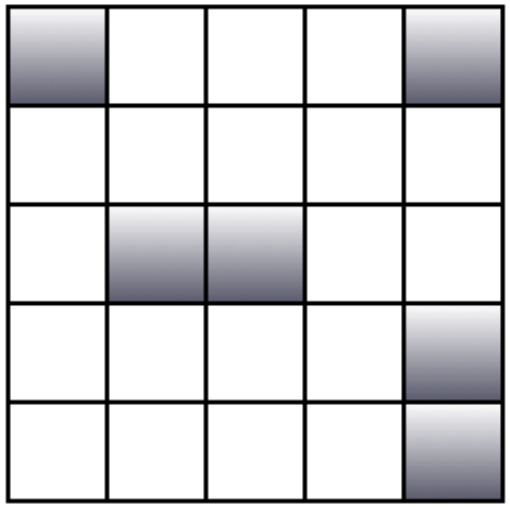
Matrix example continued

```
/* column_mpi_t is declared to have type MPI_Datatype */

MPI_Type_vector(10,1,10,MPI_FLOAT, &column_mpi_t);
MPI_Type_commit(&column_mpi_t);
if (my_rank==0)
   MPI_Send(&(A[0][2]),1,column_mpi_t,1,0,MPI_COMM_WORLD)
else
   MPI_Recv(&(A[0][2]),1,column_mpi_1,0,0,MPI_COMM_WORLD,
   &status);

/* column_mpi_t can be used to send any column in any
10 by 10 matrix of floats */
```

MPI_Type_indexed

```
int MPI_Type_indexed( int count, int *blocklens,
        int *indices, MPI_Datatype old_type,
        MPI_Datatype *newtype )
```

- count - number of blocks, also number of entries in blocklens and in indices
- blocklens - array with number of elements in each block
- indices - displacements of each block in multiples of old type
- old_type - old type
- newtype - new type



A pair of MPI_Send/MPI_Recv can have different types

For example, a type containing N floats will be compatible with type containing N floats, even if displacements between elements are different.

```
float A[10][10];
if (my_rank == 0 )
  MPI_Send(&(A[0][0]),1,column_mpi_t,1,0,MPI_COMM_WORLD);
else if (my_rank == 1)
  MPI_Recv(&(A[0][0],10,MPI_FLOAT,0,0,MPI_COMM_WORLD,
  &status);
```

- This will send the first column of the matrix A on process 0 to the first row of matrix A on process 1
- In contrast, collective communications (eg. MPI_Bcast) must use the same datatype across all processes when called)
- Receiving type must be compatible with sending type

Alternative: pack/unpack

- MPI_Pack - allows one to explicitly store noncontiguous data in contiguous memory locations.
- MPI_Unpack - can be used to copy data from a contiguous buffer into noncontiguous memory locations.

Communicators, groups, contexts

Processes can be collected into groups

A group is an ordered set of processes - Each process has a unique rank in the group - Ranks are from 0 to p − 1, where p is the number of processes in the group

A communicator consists of a:

- group
- context, a system-defined object that uniquely identifies a communicator

- A process is identified by its rank in the group associated with a communicator
- MPI_COMM_WORLD is a default communicator, whose group contains all initial processes

When to create a new communicator

- To achieve modularity; e.g. a library can exchange messages in one context, while an application can work within another context
- Use of tags is not sufficient, as we need to know the tags in other modules
- To restrict a collective communication to a subset of processes
- To create a virtual topology that fits the communication pattern better

Some group and communicator operations

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```

- Returns a handle to the group associated with comm
- Obtain group for existing communicator: MPI_Comm -> MPI_Group

MPI_Group_incl

```
int MPI_Group_incl(MPI_Group original_group, int n,
        int *ranks, MPI_Group *new_group)
```

- creates a new group from a list of processes in the original_group
- n is the number of processes in the new group
- ranks array lists processes to be included, using their index in original_group
- Process i in new_group has rank *rank[i]* in original_group

MPI_Comm_create

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group new_group ,
                    MPI_Comm *new_comm)
```

- Associates a context with new group and creates new_comm
- All the processes in new group belong to the group underlying comm
- This is a collective operation
- All process in comm must call MPI_Comm_create, so all processes choose a single context for the new communicator

Example

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NPROCS 8

int main(int argc, char *argv[]) {
  int rank, new_rank,
    sendbuf, recvbuf, numtasks;
  int ranks1[4]={0,1,2,3};
  int ranks2[4]={4,5,6,7};

  MPI_Group  orig_group, new_group;
  MPI_Comm   new_comm;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
if (numtasks != NPROCS && rank==0) {
      printf("Must specify MP_PROCS = %d.\n", NPROCS);
      MPI_Finalize(); exit(0); }

  /* store the global rank in sendbuf */
  sendbuf = rank;

  /* Extract the original group handle */
  MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

  /* Divide tasks into two distinct groups based upon rank
  if (rank < numtasks/2)
    /* if rank = 0,1,2,3, put original processes 0,1,2,3
       into new_group */
    MPI_Group_incl(orig_group, 4, ranks1, &new_group);
  else
    /* if rank = 4,5,6,7, put original processes 4,5,6,7
       into new_group */
    MPI_Group_incl(orig_group, 4, ranks2, &new_group);
```

```
/* Create new new communicator and then perform collective
   communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

/* new_comm contains a group with processes 0,1,2,3
   on processes 0,1,2,3 */
/* new_comm contains a group with processes 4,5,6,7
   on processes 4,5,6,7 */
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
              MPI_SUM, new_comm);

/* new_rank is the rank of my processs in the new group
MPI_Group_rank (new_group, &new_rank);

printf("rank= %d newrank= %d recvbuf= %d\n",
       rank,new_rank,recvbuf);

MPI_Finalize();
return 0; }
```

MPI_Comm_split

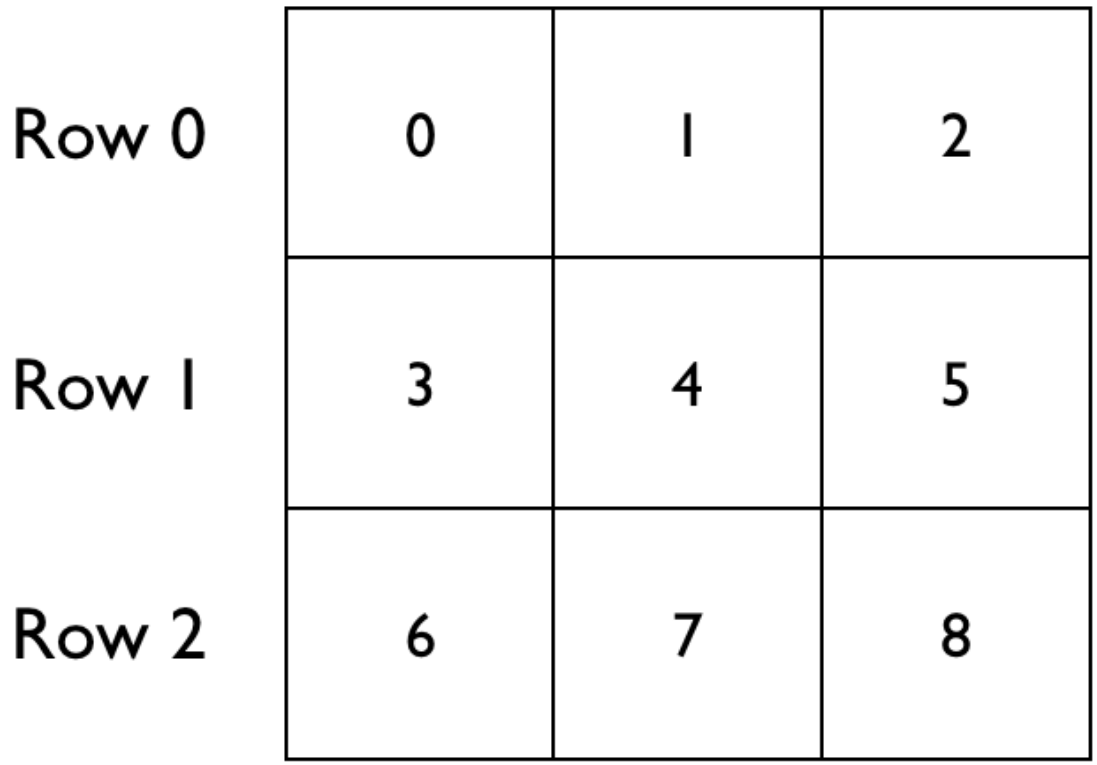
```
int MPI_Comm_split(MPI_Comm comm,int color ,int key,
    MPI_Comm *comm_out)
```

- partitions the group associated with comm into disjoint subgroups, one for each value of color
- each subgroup contains all processes marked with the same color
- within each subgroup, processes are ranked in order defined by the value of key
- ties are broken according to their rank in the old group
- a new communicator is created for each subgroup and returned in comm_out
- although a collective operation, each process is allowed to provide different values for color and key
- the value of color must be greater than or equal to 0

N=q*q processes arranged on a q by q grid (q=3 in this example)

Need to define communicator for each row of processes
i.e. processes {0,1,2}, {3,4,5} and {6,7,8}, for easy communication between them

MPI_Comm_split provides an easy way to do this, more convenient than using MPI_Group_include and MPI_Comm_create



```
/* comm_split.c -- build a collection of q
   communicators using MPI_Comm_split
                the q communicators.
 * Note:  Assumes the number of processes, p = q^2
 */
#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(int argc, char* argv[])
{
  int        p, my_rank;
  MPI_Comm   my_row_comm;
  int        my_row, my_rank_in_row;
  int        q, test;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
q = (int) sqrt((double) p);
/* my_rank is rank in MPI_COMM_WORLD.
   q*q = p */
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
               &my_row_comm);

/* Test the new communicators */
MPI_Comm_rank(my_row_comm, &my_rank_in_row);
if (my_rank_in_row == 0) test = my_row;
else test = 0;

MPI_Bcast(&test, 1, MPI_INT, 0, my_row_comm);

printf("Process %d > my_row = %d,"
       "my_rank_in_row = %d, test = %d\n",
       my_rank, my_row, my_rank_in_row, test);
MPI_Finalize();
return 0; }
```

Cartesian topology

- convenient way to handle Cartesian grid topology for processors
- saves programmer work
- MPI implementation may assign processors optimally to fit the topology

Process coordinates begin with 0

Row-major numbering

Example: 12 processes arranged on a 3 x 4 grid

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

MPI_Cart_create

```
int MPI_Cart_create(MPI_Comm comm_old,int ndims,int *dims,
        int *periods , int reorder , MPI_Comm *comm_cart)
```

- Creates a new communicator with Cartesian topology of arbitrary dimension
- comm - old input communicator
- ndims - number of dimensions of Cartesian grid
- dims - array of size ndims specifying the number of processes in each dimension
- periods - logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
- reorder - ranking of initial processes may be reordered (true) or not (false)
- comm_cart - communicator with new Cartesian topology

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
            int *coords)
```

- Rank-to-coordinates translator
- comm - communicator with Cartesian structure
- rank - rank of a process within group of comm
- maxdims - length of vector coords in the calling program
- coords -array containing the Cartesian coordinates of specified process

```
int MPI_Cart_rank(MPI_Comm comm, int *coords , int *rank)
```

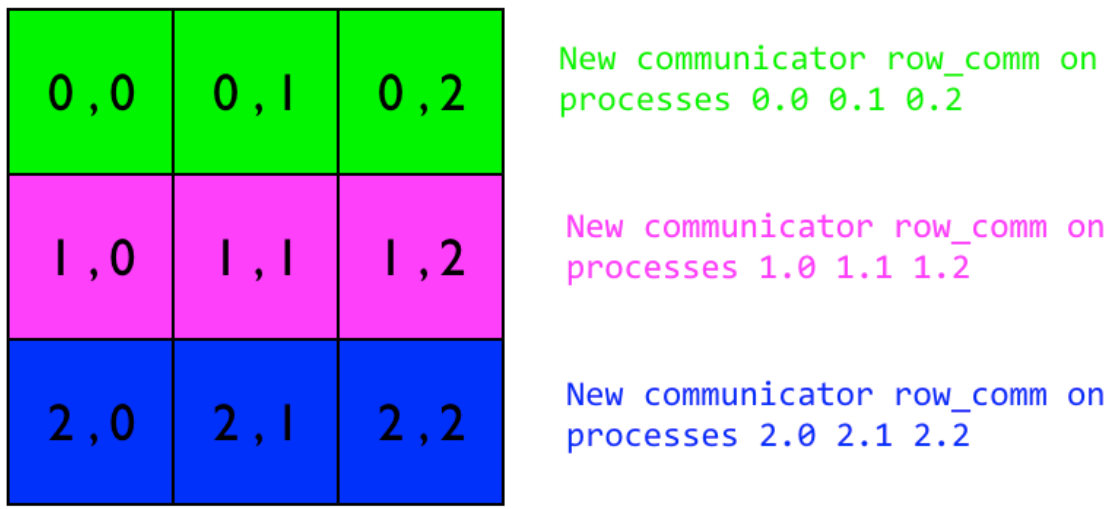
Coordinates-to-rank translator

```
int MPI_Cart_sub(MPI_Comm comm, int *free_coords ,
            MPI Comm *newcomm)
```

- Partitions a communicator into subgroups which form lower-dimensional Cartesian subgrids
- comm communicator with Cartesian structure
- free_coords - an array which specifies which dimensions are free (true) and which are not free (false)
- Free dimensions are allowed to vary, i.e. we sum over that index to create a new communicator
- newcomm - communicator containing the subgrid that includes the calling process
- In general this call creates multiple new communicators, though only one on each process

Illustration with code

```
int free_coords[2];
MPI_Comm row_comm;

free_coords[0] = 0;
free_coords[1] = 1;
MPI_Cart_sub(grid_comm, free_coords, &row_comm);
```



Algorithm:

- Build a 2-dimensional Cartesian communicator from MPI_Comm_world
- Print topology information for each process
- Use MPI_Cart_sub to build a communicator for each row of the Cartesian communicator
- Carry out a broadcast across each row communicator
- Print results of broadcast
- Use MPI_Cart_sub to build a communicator for each column of the Cartesian communicator
- Carry out a broadcast across each column communicator
- Print results of broadcast

Note: Assumes the number of processes, p, is a perfect square

```c
#include <stdio.h>
#include "mpi.h"
#include <math.h>
int main(int argc, char* argv[]) {
  int p, my_rank, q;
  MPI_Comm  grid_comm;
  int dim_sizes[2];
  int wrap_around[2];
  int coordinates[2];
  int free_coords[2];
  int reorder = 1;
  int my_grid_rank, grid_rank;
  int       row_test, col_test;
  MPI_Comm  row_comm;
  MPI_Comm  col_comm;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
q = (int) sqrt((double) p);
dim_sizes[0] = dim_sizes[1] = q;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes,
                wrap_around, reorder, &grid_comm);

MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2,
                coordinates);

MPI_Cart_rank(grid_comm, coordinates, &grid_rank);

printf("Process %d > my_grid_rank = %d,"
       "coords = (%d,%d), grid_rank = %d\n",
       my_rank, my_grid_rank, coordinates[0],
       coordinates[1], grid_rank);
```

```
free_coords[0] = 0;
free_coords[1] = 1;

MPI_Cart_sub(grid_comm, free_coords, &row_comm);

if (coordinates[1] == 0)
  row_test = coordinates[0];
else
  row_test = -1;

MPI_Bcast(&row_test, 1, MPI_INT, 0, row_comm);
printf("Process %d > coords = (%d,%d), row_test = %d\n",
 my_rank, coordinates[0], coordinates[1], row_test);
```

```
free_coords[0] = 1;
free_coords[1] = 0;

MPI_Cart_sub(grid_comm, free_coords, &col_comm);

if (coordinates[0] == 0)
  col_test = coordinates[1];
else
  col_test = -1;

MPI_Bcast(&col_test, 1, MPI_INT, 0, col_comm);

printf("Process %d > coords = (%d,%d), col_test = %d\n",
 my_rank, coordinates[0], coordinates[1], col_test);
MPI_Finalize(); return 0; }
```

Sending and receiving in Cartesian topology

- There is no MPI_Cart_send or MPI_Cart_recv which would allow you to send a message to process (1,0) in your Cartesian topology, for example
- You must use standard communication functions
- There is a convenient way to obtain the rank of the desired destination/source process from your Cartesian coordinate grid
- Usually one needs to determine which are the adjacent processes in the grid and obtain their ranks in order to communicate

```
int MPI_Cart_shift ( MPI_Comm comm, int direction,
        int displ, int *source, int *dest )
```

- comm - communicator with cartesian structure
- direction - coordinate dimension of shift, in range [0,n-1] for an n-dimensional Cartesian grid
- displ - displacement (> 0: upwards shift, < 0: downwards shift), with periodic wraparound possible if communicator created with periodic boundary conditions turned on
- outputs are possible inputs to MPI_Sendrecv
- source - rank of process to receive data from, obtained by subtracting displ from coordinate determined by direction
- dest - rank of process to send data to, obtained by adding displ to coordinate determined by direction
- These may be undefined (i.e. $=$ MPI_PROC_NULL) if shift points outside grid structure and the periodic boundary conditions off

MPI_Cart_shift(comm, 1, 1, &source, &dest);

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

MPI_Cart_shift(comm, 1, 1, &source, &dest);

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

MPI_Cart_shift(comm, 0, -1, &source, &dest);

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