Lecture 10

Topologies

Outline

- Introduction
- Cartesian topology
- Some Cartesian topology functions
- Some graph topology functions
- Example

Introduction

Additional information can be associated, or cached, with a communicator (i.e. not just group and context).

Topology is a mechanism for associating different addressing schemes with processes.

A topology can be added to an intra-communicator, but not to inter-communicator.

A topology

- can provide a convenient naming mechanism for processes

- may assist the runtime system in mapping processes onto hardware

There are virtual process topology and topology of the underlying hardware.

The virtual topology can be exploited by the system in assigning of processes to processors.

Two types:

- Cartesian topology

- graph topology

Cartesian 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)

Some Cartesian topology functions

```
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. Collective operation.

comm_old - 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. Collective operation.

comm - communicator with Cartesian structure.

free_coords - an array which specifies which dimensions are free (true) and which are not free (false; they have thickness = 1).

Free dimensions are allowed to vary, i.e. we travel over that index to create a new communicator.

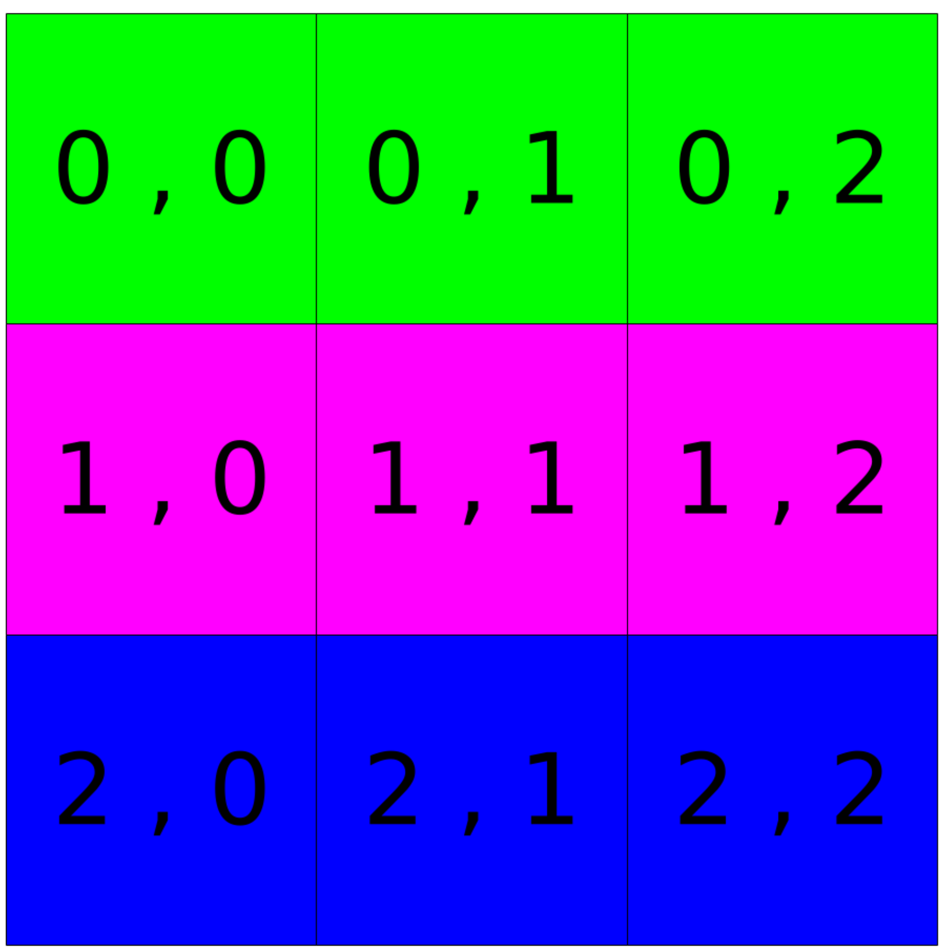
newcomm - communicator containing the subgrid that includes the calling process.

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);
```



New communicator row_comm on processes 0,0 0,1 0,2

New communicator row_comm on processes 1,0 1,1 1,2

New communicator row_comm on processes 2,0 2,1 2,2

Example

Code adapted from P. Pacheco, PP with MPI

```
/* top_fcns.c -- test basic topology functions
 *
 * Algorithm:
 *     1.  Build a 2-dimensional Cartesian communicator from
 *         MPI_Comm_world
 *     2.  Print topology information for each process
 *     3.  Use MPI_Cart_sub to build a communicator for each
 *         row of the Cartesian communicator
 *     4.  Carry out a broadcast across each row communicator
 *     5.  Print results of broadcast
 *     6.  Use MPI_Cart_sub to build a communicator for each
 *         column of the Cartesian communicator
 *     7.  Carry out a broadcast across each column
 *         communicator
 *     8.  Print results of broadcast
 *
 * Note: Assumes the number of processes, p, is a
 * perfect square
 */
```

~syam/ces745/mpi/topologies/top_fcns.c

```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

Communicator defined with periodic boundary conditions

```
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

Communicator NOT defined with periodic boundary conditions

```
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

Communicator NOT defined with periodic boundary conditions

Graph topology

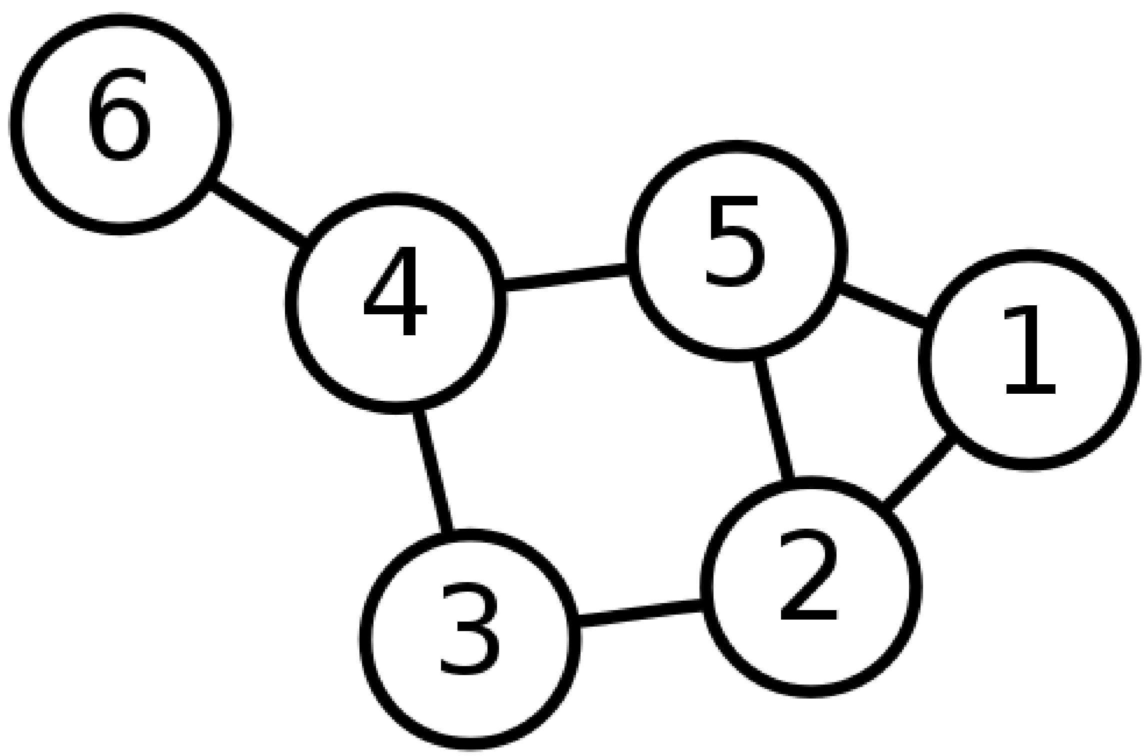
Graph - abstract representation of a set of objects (vertices, nodes, points) where some pairs are connected by links (edges).

The degree of a vertex is the number of edges that connect to it.

In MPI topology, vertices usually stand for processes and edges for communications links.

However, it is still possible to send a message between any two vertices, they do not have to be connected.

Edges may represent optimal (fast) communications links.



MPI_Graph_create

```
int MPI_Graph_create (MPI_Comm comm_old, int nnodes, int *index ,
int *edges , int reorder , MPI_Comm *comm_graph)
```

Creates a communicator with a graph topology attached. Collective operation.

comm_old - input communicator without topology

nnodes - number of nodes in graph

index - array of integers describing node degrees

edges - array of integers describing graph edges

reorder - ranking may be reordered (true) or not (false) (logical)

comm_graph - communicator with graph topology added

The *i*-th entry of *index* stores the total number of neighbours of the first *i* graph nodes (i.e. *index* is cumulative).

The list of neighbours of nodes 0, 1, . . . , *nnodes*-1 are stored in consecutive locations in array edges (each edge counted twice since bi-directional communication assumed).

Example:

Assume 4 processes such that

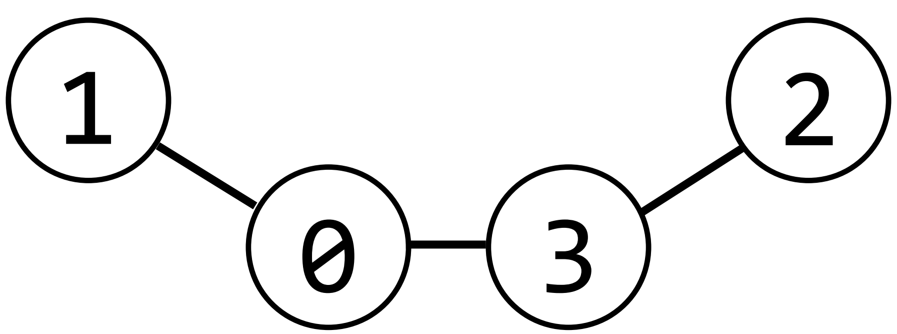
The input should be

nnodes = 4

index = (2,3,4,6) : cumulative

edges = (1,3,0,3,0,2)

process	neighbours
0	1 , 3
1	0
2	3
3	0 , 2



Some graph topology functions

MPI_Graphdims_get - returns number of nodes and edges in a graph.

MPI_Graph_get - returns index and edges as supplied to MPI_Graph_create.

MPI_Graph_neighbours_count - returns the number of neighbours of a given process.

MPI_Graph_neighbours - returns the edges associated with given process.

MPI_Graph_map - returns a graph topology recommended by the MPI system.

Overall, graph topology is more general that Cartesian, but not widely used as it is less convenient for many problems.

If the computational problem cannot be represented in a Cartesian grid topology, then most likely load balancing becomes a serious issue to be addressed.

Communicators and Topologies: Matrix Multiplication Example

Fox's algorithm

A and B are n x n matrices.

Compute $C = AB$ in parallel.

Let q = sqrt(p) be an integer such that it divides n, where p is the number of processes.

Create a Cartesian topology with processes (i,j), $i,j = 0,...,q-1$.

Denote $nb = n/q$.

Distribute A and B by blocks on p processes such that $A_{i,j}$ and $B_{i,j}$ are nb x nb blocks stored on process (i,j).

Algorithm goal

Perform the operation with as few communications as possible.

Have a clear communication scheme that results in effective code.

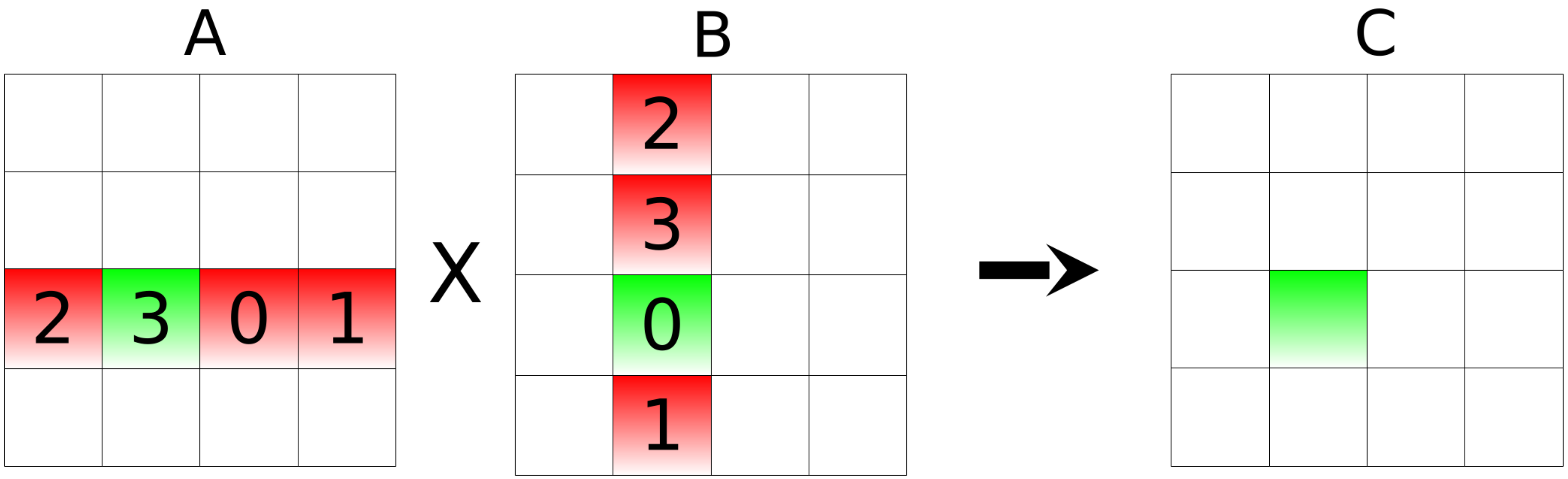
On process (i,j), we want to compute

$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots + A_{i,q-1}B_{q-1,j}$

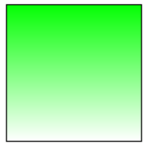
Rewrite this as

stage 0	$C_{i,j} = A_{i,i}B_{i,j}$
stage 1	$C_{i,j} = C_{i,j} + A_{i,i+1}B_{i+1,j}$
...	...
stage	$C_{i,j} = C_{i,j} + A_{i,q-1}B_{q-1,j}$
stage	$C_{i,j} = C_{i,j} + A_{i,0}B_{0,j}$
stage	$C_{i,j} = C_{i,j} + A_{i,1}B_{1,j}$
...	...
stage q-1	$C_{i,j} = C_{i,j} + A_{i,i-1}B_{i-1,j}$

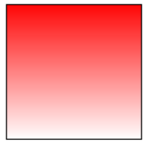
Example: 4 x 4 block matrix



Highlighted blocks need to compute result block for process.



Available to process at start.



Has to be received from other processors.

Numbers indicate order in which operations are done.

Initially, assume $C_{i,j} = 0$, the zero block, on each (i,j).

Process (i,j) at stage 0:

- (i,j) has $B_{i,j}$
- (i,j) needs $A_{i,i}$ which it has to get from process (i,i)
- (i,i) broadcasts $A_{i,i}$ across processor row i
- $C_{i,j} = C_{i,j} + A_{i,i}B_{i,j}$

Process (i,j) at stage 1:

- (i,j) has $B_{i,j}$, but needs $B_{i+1,j}$
  Shift the j-th block column of B by one block up
  Block 0 goes to block q-1
- (i,j) needs $A_{i,i+1}$ which it has to get from process (i,i+1)
- (i,i+1) broadcasts $A_{i,i+1}$ across processor row i
- $C_{i,j} = C_{i,j} + A_{i,i+1}B_{i+1,j}$

Similarly on next stages.

Implementation (just essential parts)

```c
/* setupgrid.c */
void Setup_grid(GRID_INFO_T*  grid)
{
  int old_rank;
  int dimensions[2];
  int wrap_around[2];
  int coordinates[2];
  int free_coords[2];

  /* Set up Global Grid Information */
  MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
  MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

  /* We assume p is a perfect square */
  grid->q = (int) sqrt((double) grid->p);
  dimensions[0] = dimensions[1] = grid->q;

  /* We want a circular shift in second dimension. */
  /* Don't care about first                        */
  wrap_around[0] = wrap_around[1] = 1;
```

```c
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
                wrap_around, 1, &(grid->comm));

MPI_Comm_rank(grid->comm, &(grid->my_rank));
MPI_Cart_coords(grid->comm, grid->my_rank, 2,
                coordinates);

grid->my_row = coordinates[0];
grid->my_col = coordinates[1];

/* Set up row communicators */
free_coords[0] = 0;
free_coords[1] = 1;
MPI_Cart_sub(grid->comm, free_coords, &(grid->row_comm));

/* Set up column communicators */
free_coords[0] = 1;
free_coords[1] = 0;
MPI_Cart_sub(grid->comm, free_coords, &(grid->col_comm));}
```

```
void Fox(int n,GRID_INFO_T* grid,
         LOCAL_MATRIX_T* local_A, LOCAL_MATRIX_T* local_B,
         LOCAL_MATRIX_T* local_C)
{
  LOCAL_MATRIX_T*  temp_A;
  int              stage;
  int              bcast_root;
  int              n_bar;  /* n/sqrt(p)              */
  int              source;
  int              dest;
  MPI_Status       status;

  n_bar = n/grid->q;
  Set_to_zero(local_C);

  /* Calculate addresses for circular shift of B */
  MPI_Cart_shift (grid->col_comm, 0, -1,  &source, &dest );

  /* Set aside storage for the broadcast block of A */
  temp_A = Local_matrix_allocate(n_bar);
```

```
  for (stage = 0; stage < grid->q; stage++)
    {
      bcast_root = (grid->my_row + stage) % grid->q;
      if (bcast_root == grid->my_col)
        {
          MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                    bcast_root, grid->row_comm);
          Local_matrix_multiply(local_A, local_B,
                                local_C);
        }
      else
        {
          MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                    bcast_root, grid->row_comm);
          Local_matrix_multiply(temp_A, local_B,
                                local_C);
        }
      if (stage < grid->q-1)
        MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
                             dest, 0, source, 0,
                             grid->col_comm, &status);
    }
}
```

Grouping Data for Communication

Introduction

In general, sending a message may be an expensive operation.

Rule of thumb: the fewer messages send, the better the performance of the program.

MPI provides three mechanisms for grouping individual data items into a single message:

- count parameter, which we have already seen before,

- derived datatypes,

- MPI_Pack/MPI_Unpack routines.

Consider send and receive.

Their execution can be divided into two phases:

- startup

- communication

Their cost varies among systems.

Denote the runtime of the startup phase by t_s, and the communication phase by t_c .

The cost of sending a message of k units is

$t_s + k*t_c$

Usually t_s is much larger than t_c .

Their values vary among systems.

It generally pays to group data before sending/receiving, in order to reduce the number of times that communication needs to be started.

This will be especially important for messages of small size (i.e. small k).

Simplest grouping

Use *count* parameter, as discussed in previous lectures.

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

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

Derived Types

Say we want to send in one communication real numbers *a* and *b*, and integer *n*.

Suppose we try:

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

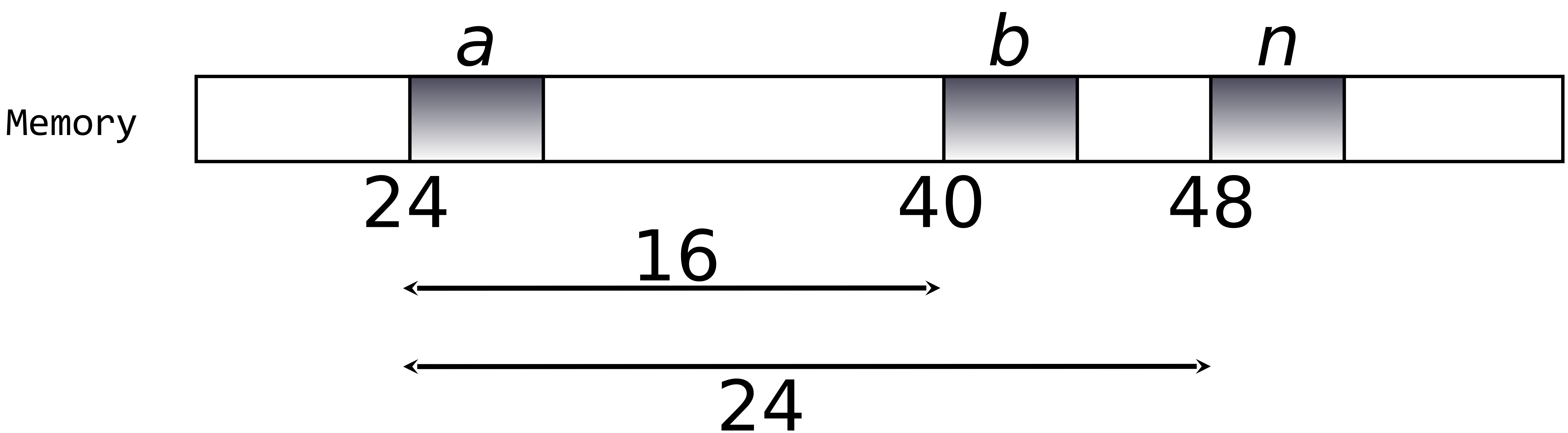
Will not work. Arguments to functions must be variables, not defined types.

Need a method of defining 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 .

We do not assume that a,b,n are contiguous in memory. For example, consider the case when they are defined and stored on process 0 as:

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



We would like to send this data in a single message while preserving the relative arrangement in memory, i.e. displacement between a and b should remain 16 bytes, and between a and n remain 24 bytes.

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

1. There are three elements to be transmitted

2. a. 1st element is a float
   b. 2nd element is a float
   c. 3rd is an int

3. a. 1st is displaced 0 bytes from the beginning of message
   b. 2nd element is displaced 16 bytes from beginning of message
   c. 3rd element is displaced 24 bytes from beginning of message

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[] - number of elements in each block.

indices[] - byte displacement of each block.
MPI_Aint is a special type for storing addresses, can handle addresses longer than can be stored in int.

old_types[] - type of elements in each block. These can be datatypes created by previous MPI_Type_struct calls.

newtype - new datatype.

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.

To be able to use the newly created datatype in communications, must call MPI_Type_commit .

Example from P. Pacheco, PP with MPI

```
void Build_derived_type(
        float*        a_ptr          /* in   */,
        float*        b_ptr          /* in   */,
        int*          n_ptr          /* in   */,
        /* pointer to new MPI type */
        MPI_Datatype*  mesg_mpi_t_ptr  /* out  */) {
    /* The number of elements in each "block" of the   */
    /*     new type.  For us, 1 each.                  */
    int block_lengths[3];

    /* Displacement of each element from start of new  */
    /*     type.  The "d_i's."                         */
    /* MPI_Aint ("address int") is an MPI defined C    */
    /*     type.  Usually an int.                      */
    MPI_Aint displacements[3];

    /* MPI types of the elements.  The "t_i's."        */
   MPI_Datatype typelist[3];

    /* Use for calculating displacements               */
    MPI_Aint start_address;
    MPI_Aint address;
    block_lengths[0] = block_lengths[1] = block_lengths[2] = 1;
```

```
/* Build a derived datatype consisting of  */
/* two floats and an int                   */
typelist[0] = MPI_FLOAT;
typelist[1] = MPI_FLOAT;
typelist[2] = MPI_INT;

/* First element, a, is at displacement 0      */
displacements[0] = 0;

/* Calculate other displacements relative to a */
MPI_Address(a_ptr, &start_address);

/* Find address of b and displacement from a   */
MPI_Address(b_ptr, &address);
displacements[1] = address - start_address;

/* Find address of n and displacement from a   */
MPI_Address(n_ptr, &address);
displacements[2] = address - start_address;

/* Build the derived datatype */
MPI_Type_struct(3, block_lengths, displacements,
    typelist, mesg_mpi_t_ptr);

/* Commit it -- tell system we'll be using it for communication*/
MPI_Type_commit(mesg_mpi_t_ptr);}         }
```

```
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.

The user must provide a complete description of each element of the type.

If we have a special case where the data to be transmitted consists of a subset of the entries in an array, we should not need to provide such detailed information since all elements have the same basic type.

MPI provides three derived datatype constructors for this situation:

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 arbitrary entries of an array.

MPI_Type_contiguous

```
int MPI_Type_contiguous ( int count, MPI_Datatype old_type,
          MPI_Datatype *newtype )
```

count - number of elements

old_type - old datatype

new_type - new datatype

Matrix Example

```
float A[10][10]; /* define 10 by 10 matrix */
```

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].

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

```
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);
}
```

We could have defined a new datatype to handle the row of the matrix with MPI_Type_Contiguous, but clearly in this case there is little need since using just MPI_Send and MPI_Recv is so easy (and more efficient).

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 )
```

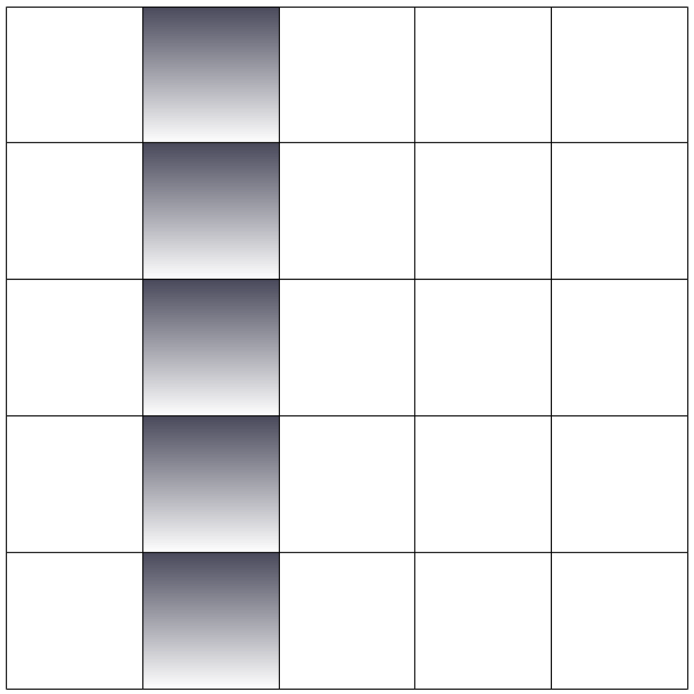
count - number of blocks

blocklen - number of elements in each block

stride - number of elements between start of each block

old_type - old datatype

new_type - new datatype



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 )
```

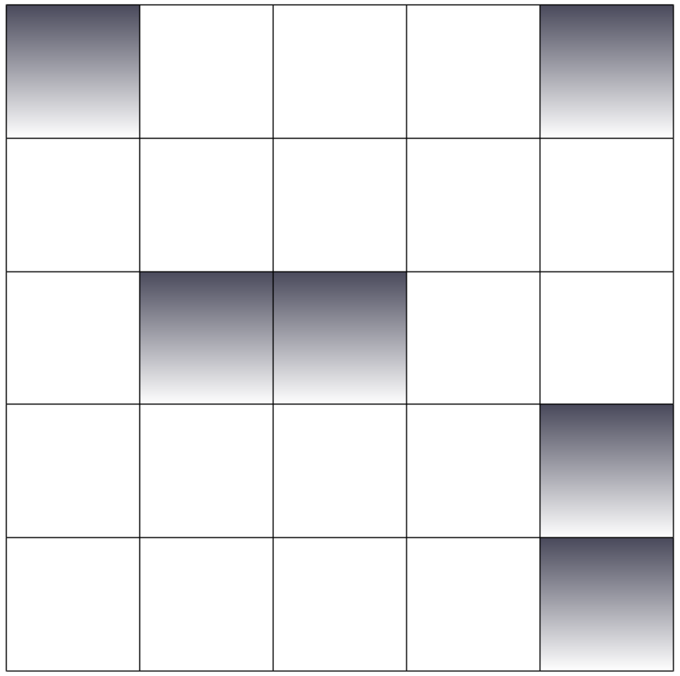
count - number of blocks, also number of entries in *blocklens[]* and in *indices[]*

blocklens[] - number of elements in each block

indices[] - displacement of each block in multiples of *old_type*

old_type - old type

newtype - new type



Can a pair of MPI_Send/MPI_Recv have different types?

Yes, as long as they are compatible.

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

For example:

```
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.