# CME 213

SPRING 2017

Eric Darve

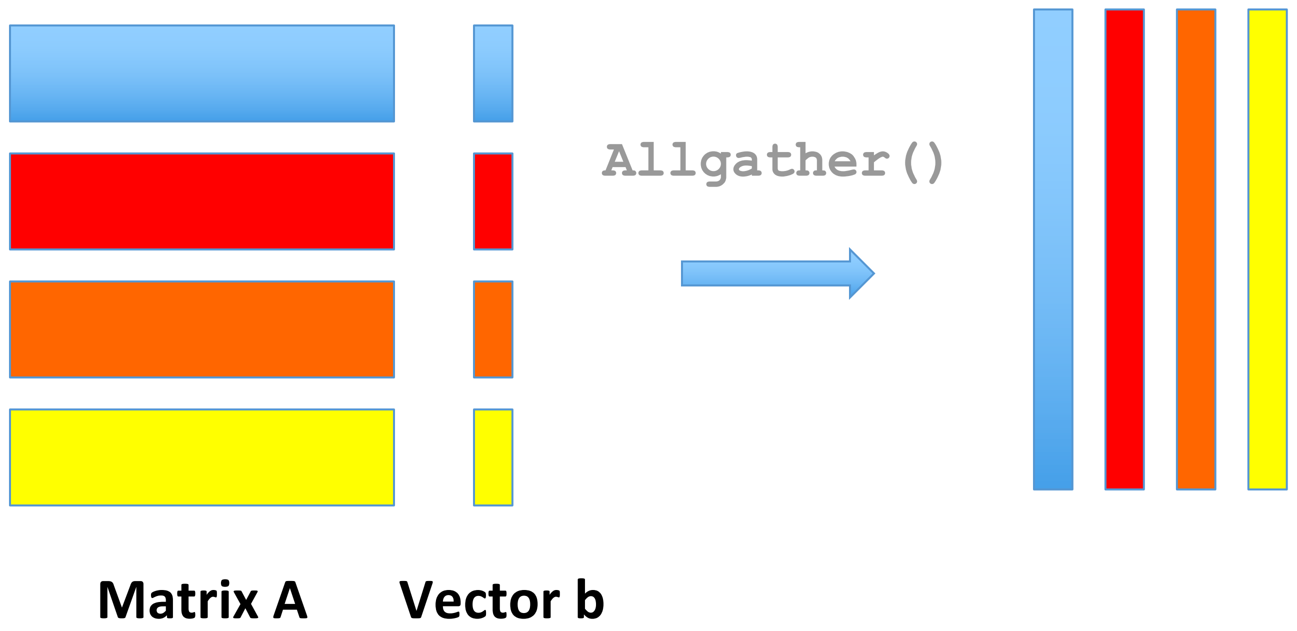
## Linear algebra
## Matrix-vector products

## Application example: matrix-vector product

- We are going to use that example to illustrate additional MPI functionalities.
- This will lead us to process groups and topologies.
- First, we go over two implementations that use the functionalities we have already covered.
- Two simple approaches:
  - Row partitioning of the matrix, or
  - Column partitioning

## Row partitioning

This is the most natural.



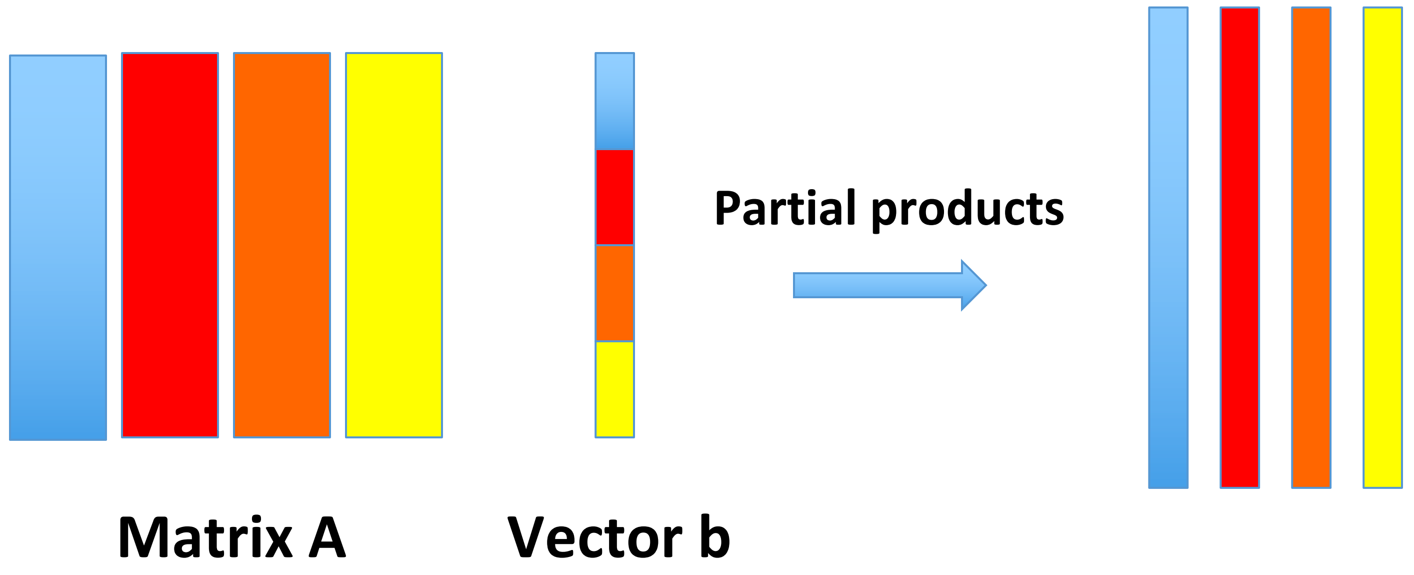
Step 1: replicate b on each process: `MPI_Allgather()`

Step 2: perform product

See MPI code: `matvecrow/`

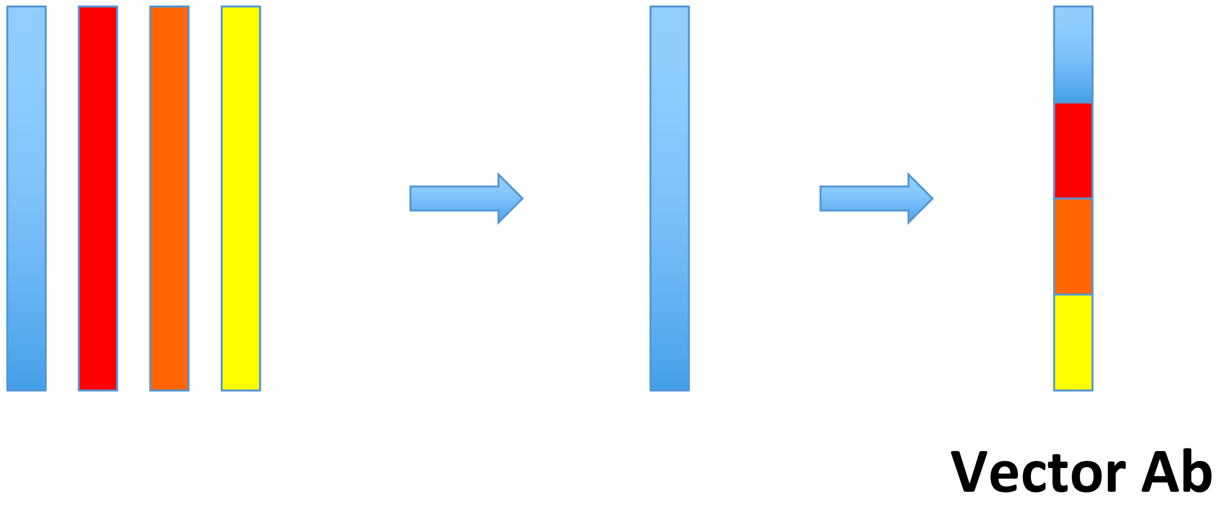
```
/* Gather entire vector b on each processor using Allgather */
MPI_Allgather(&bloc[0], nlocal, MPI_FLOAT, &b[0], nlocal, MPI_FLOAT,
              MPI_COMM_WORLD);
// sending nlocal and receiving nlocal from any other process

/* Perform the matrix-vector multiplication involving the
 locally stored submatrix. */
vector<float> x(nlocal);

for(int i=0; i<nlocal; i++) {
  x[i] = 0.0;

  for(int j=0; j<n; j++) {
    x[i] += a[i*n+j]*b[j];
  }
}
```

## Column partitioning



Step 1: calculate partial products with each process

# Column partitioning (cont’d)



- Step 2: reduce all partial results: `MPI_Reduce()`
- Step 3: send sub-blocks to all processes: `MPI_Scatter()`
- Steps are very similar to row partitioning.

# ow and column partitioning sch are dual of each other

Yes **A**

No **B**

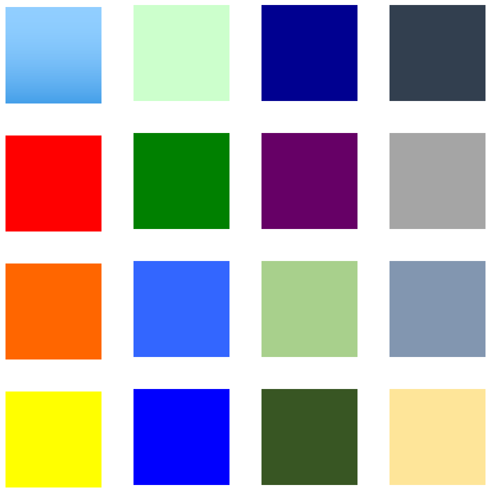
**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

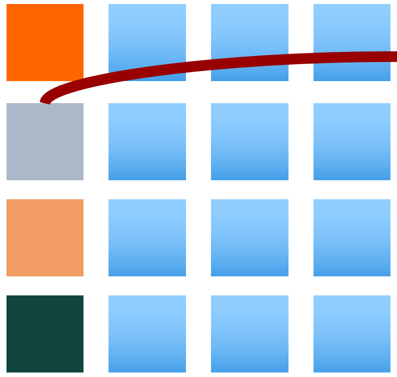
# A better partitioning

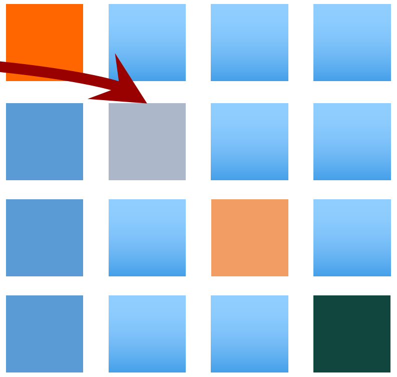
- If the number of processes becomes large compared to the matrix size, we need a 2D partitioning:

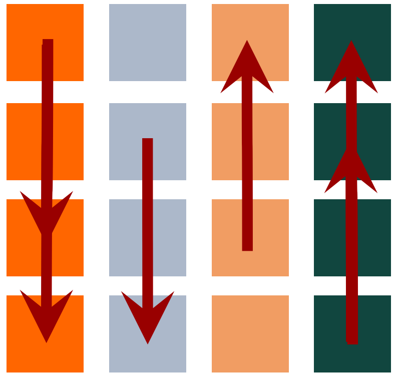


- Each colored square can be assigned to a process.
- This allows using more processes.
- In addition, a theoretical analysis (more on this later) shows that this scheme runs faster.

# Outline of algorithm: step 1


First column contains b

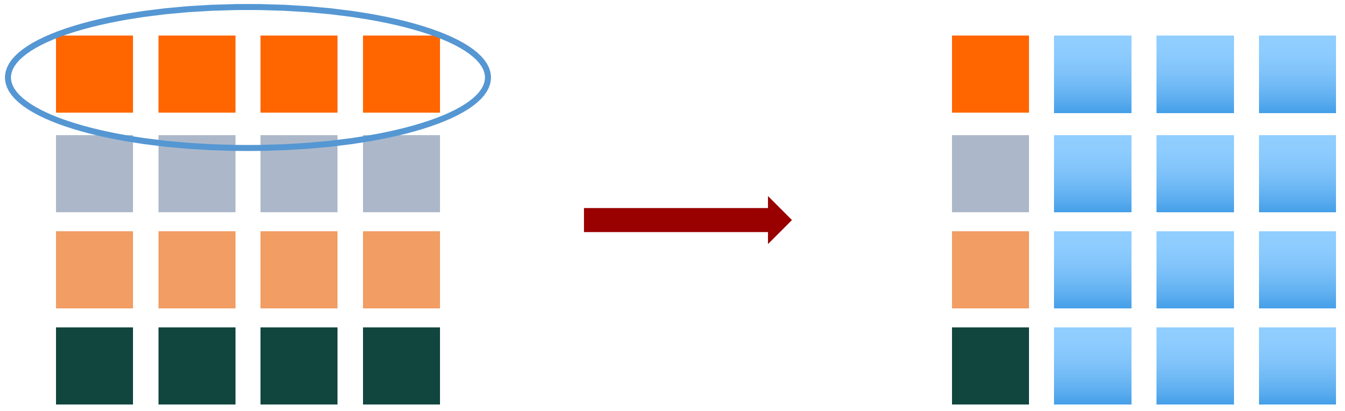

Send b to the diagonal processes


Send b down each column.

This is a broadcast operation.

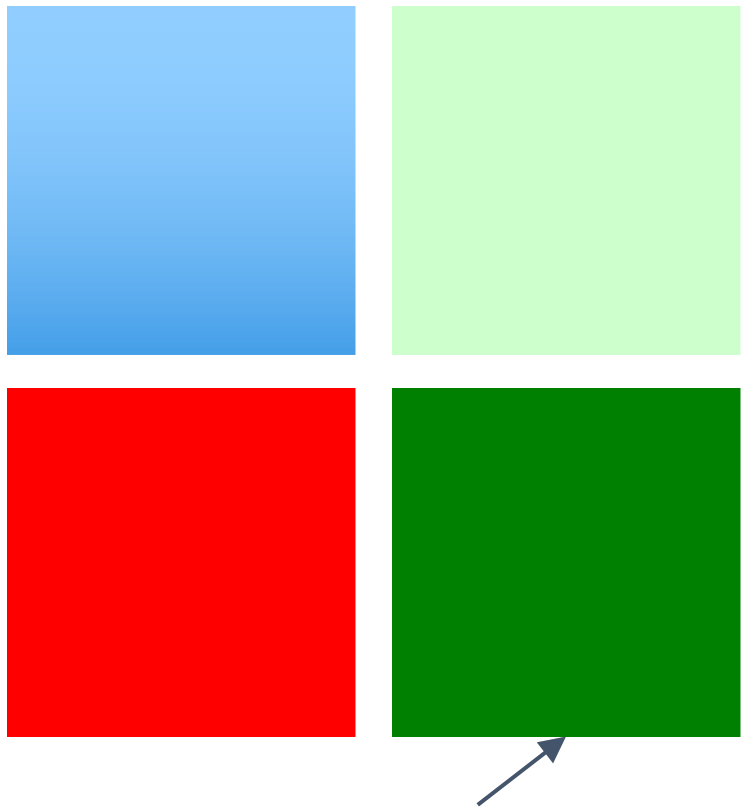
## Step 2 and 3

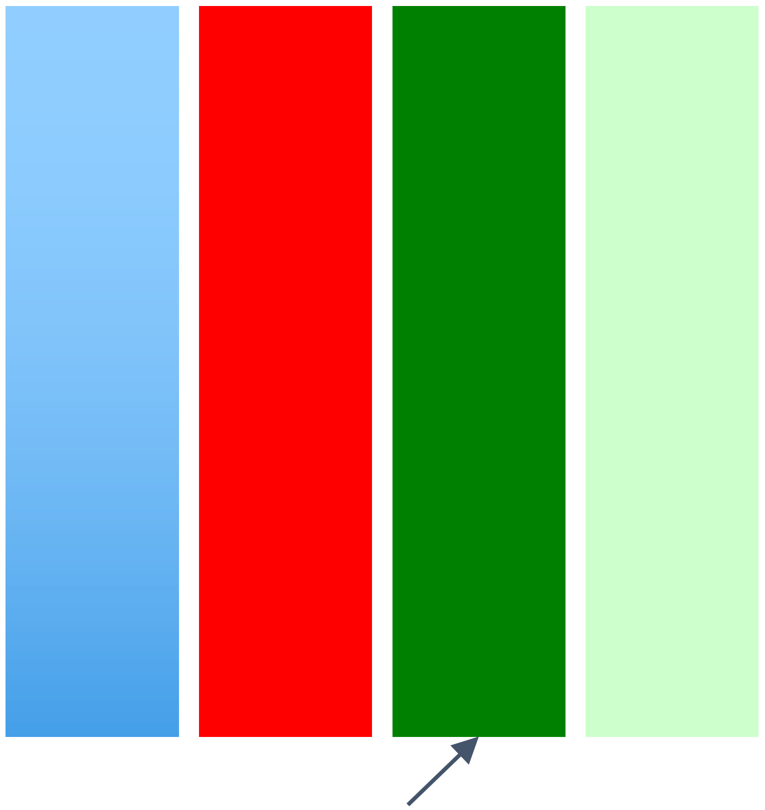
- Step 2: perform matrix-vector product locally
- Step 3: reduce across columns and store result in column 0.



Reduction across columns

## Communication cost (in a nutshell) Why is 2D partitioning better?





## Difficulties with 2D partitioning

- This type of decomposition brings some difficulties.
- We used two collective operations:
    - A broadcast inside a column.
    - A reduction inside a row.
- To do this in MPI, we need two concepts:
    - Communicators or process groups. This defines a subset of all the processes. For each subset, collective operations are allowed, e.g., broadcast for the group of processes inside a column.
    - Process topologies. For matrices, there is a natural 2D topology with (i,j) block indexing. MPI supports such grids (any dimension). Using MPI grids (called “Cartesian topologies”) simplifies many MPI commands.

## Process groups and communicators

## Process groups

- Groups are needed for many reasons.
- Enables collective communication operations across a subset of processes.
- Allows to easily assign independent tasks to different groups of processes.
- Provides a good mechanism to integrate a parallel library into an MPI code.

## Groups and communicators

- A group is an ordered set of processes.
- Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group.
- A group is always associated with a communicator object.
- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator.
- For example, the handle for the communicator that comprises all tasks is `MPI_COMM_WORLD`.
- From the programmer's perspective, a group and a communicator are almost the same. The group routines are primarily used to specify which processes should be used to construct a communicator.
- Processes may be in more than one group/communicator. They have a unique specific rank within each group/communicator.

## Main functions

**MPI provides over 40 routines related to groups, communicators, and virtual topologies!**

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

**Return group associated with communicator, e.g., `MPI_COMM_WORLD`**

```
int MPI_Group_incl(MPI_Group group, int p, int *ranks,
    MPI_Group *new_group)
```

`ranks` **integer array with p entries.**

**Creates a new group `new_group` with p processes, which have ranks from 0 to p-1. Process i is the process that has rank ranks[i] in group.**

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

**New communicator based on group.**

**See MPI code:** `groups/`

```c
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

/* Divide tasks into two distinct groups based upon rank */
int mygroup = 0;
if(rank >= NPROCS/2) {
  mygroup = 1;
}

int ranks1[4]= {0,1,2,3}, ranks2[4]= {4,5,6,7};
/* These arrays specify the rank to be used
 * to create 2 separate process groups.
 */
MPI_Group_incl(world_group, NPROCS/2, ranks1, &sub_group[0]);
MPI_Group_incl(world_group, NPROCS/2, ranks2, &sub_group[1]);

/* Create new new communicator and then perform collective communications */
MPI_Comm_create(MPI_COMM_WORLD, sub_group[mygroup], &sub_group_comm);
// Summing up the value of the rank for all processes in my group
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, sub_group_comm);

MPI_Group_rank(sub_group[mygroup], &group_rank);
printf("Rank= %d; Group rank= %d; recvbuf= %d\n",rank,group_rank,recvbuf);
```

## Process topologies

## Process topologies

- Many problems are naturally mapped to certain topologies such as grids.
- This is the case for example for matrices, or for 2D and 3D structured grids.
- The two main types of topologies supported by MPI are Cartesian grids and graphs.
- MPI topologies allow simplifying many common MPI tasks.
- MPI topologies are virtual — there may be no relation between the physical structure of the network and the process topology.

## Advantages of using topologies

- Convenience: virtual topologies may be useful for applications with specific communication patterns.
- Communication efficiency: a particular implementation may optimize the process mapping based upon the physical characteristics of a given parallel machine.
  - For example nodes that are nearby on the grid (East/West/North/South neighbors) may be close in the network (lowest communication time).
- The mapping of processes onto an MPI virtual topology is dependent upon the MPI implementation.

# MPI functions for topologies

Many functions are available.

We only cover the basic ones.

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

`ndims` number of dimensions

`dims[i]` size of grid along dimension i. Should not exceed the number of processes in `comm_old`.

The array `periods` is used to specify whether or not the topology has wraparound connections. If `periods[i]` is non-zero, then the topology has wraparound connections along dimension i.

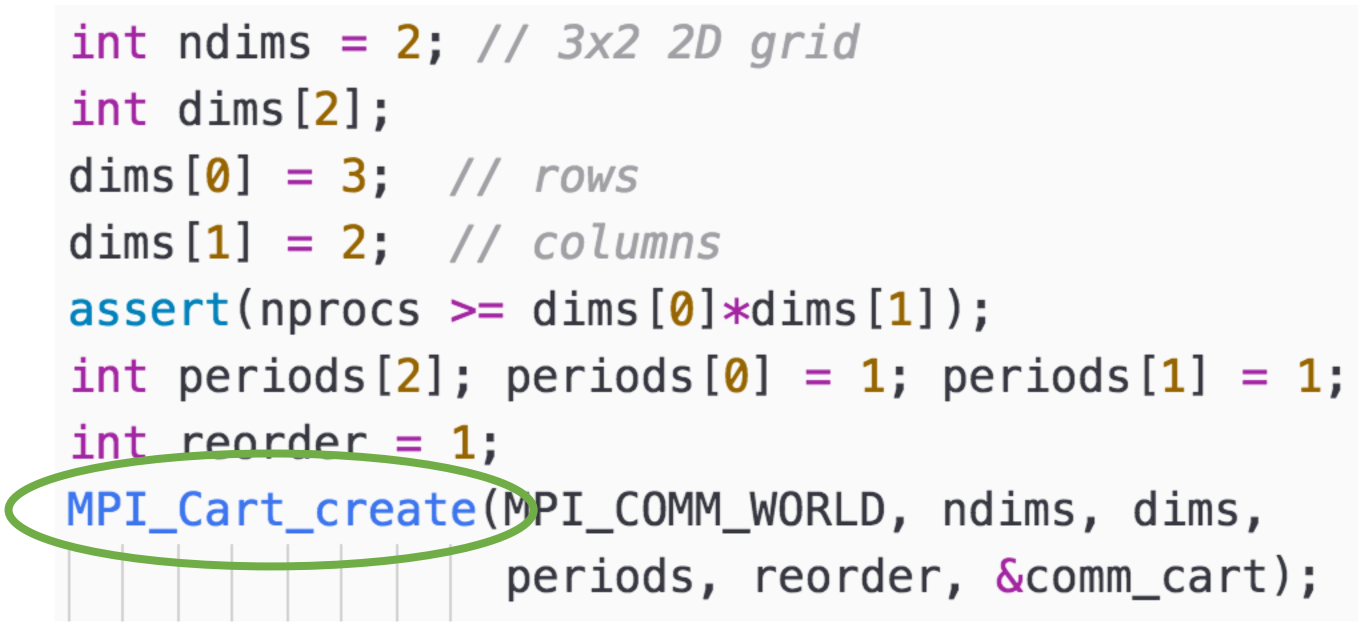
`reorder` is used to determine if the processes in the new group are to be reordered or not. If `reorder` is false, then the rank of each process in the new group is identical to its rank in the old group.

## Example

| | |
|---|---|
| 0 (0,0) | 1 (0,1) |
| 2 (1,0) | 3 (1,1) |
| 4 (2,0) | 5 (2,1) |

**The processes are ordered according to their rank row-wise in increasing order.**

## Periodic Cartesian grids



```c
int ndims = 2; // 3x2 2D grid
int dims[2];
dims[0] = 3;  // rows
dims[1] = 2;  // columns
assert(nprocs >= dims[0]*dims[1]);
int periods[2]; periods[0] = 1; periods[1] = 1;
int reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims,
                periods, reorder, &comm_cart);
```

- **We chose periodicity along the first dimension (`periods[0]=1`) which means that any reference beyond the first or last entry of any row will be wrapped around cyclically.**
- **For example, row index `i=-1` is mapped into `i=2`.**
- **There is no periodicity imposed on the second dimension. Any reference to a column index outside of its defined range results in an error. Try it!**

## Obtaining your rank and coordinates

```
int MPI_Cart_rank(MPI_Comm comm_cart,
   int *coords, int *rank)
int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
   int maxdims, int *coords)
```

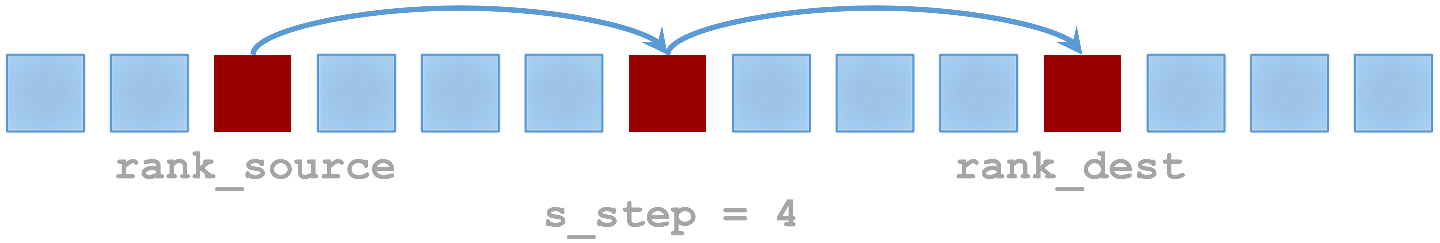
- This allows retrieving a rank or the coordinates in the grid. This may be useful to get information about other processes.
- `coords` are the Cartesian coordinates of a process.
- Its size is the number of dimensions.
- Remember that the function `MPI_Comm_rank` is still available to query your own rank.
- See MPI code: `mpi_cart/`

```c
/* Get my rank in the new topology */
int my2drank;
MPI_Comm_rank(comm_cart, &my2drank);

/* Get my coordinates */
int mycoords[2];
MPI_Cart_coords(comm_cart, my2drank, 2, mycoords);

/* Get coordinates of process below me */
int rank_down, coords[2];
coords[0] = mycoords[0]+1; // i coordinate (one row below in matrix)
coords[1] = mycoords[1];
MPI_Cart_rank(comm_cart, coords, &rank_down);

/* Get coordinates of process to my right */
int rank_right;
coords[0] = mycoords[0];
coords[1] = mycoords[1]+1; // j coordinate (to the right in matrix)
MPI_Cart_rank(comm_cart, coords, &rank_right);
```

# Getting the rank of your neighbors

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir,
    int s_step, int *rank_source, int *rank_dest)
```

- `dir` direction
- `s_step` length shift
- `rank_dest` contains the group rank of the neighboring process in the specified dimension and distance.
- `rank_source` is the rank of the process for which the calling process is the neighboring process in the specified dimension and distance.
- Thus, the group ranks returned in `rank_dest` and `rank_source` can be used as parameters for `MPI_Sendrecv()`.



## MPI virtual topology reflects t topology of the underlying hardware network

**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

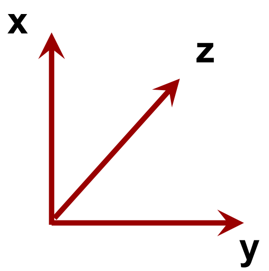
## Splitting a Cartesian topology

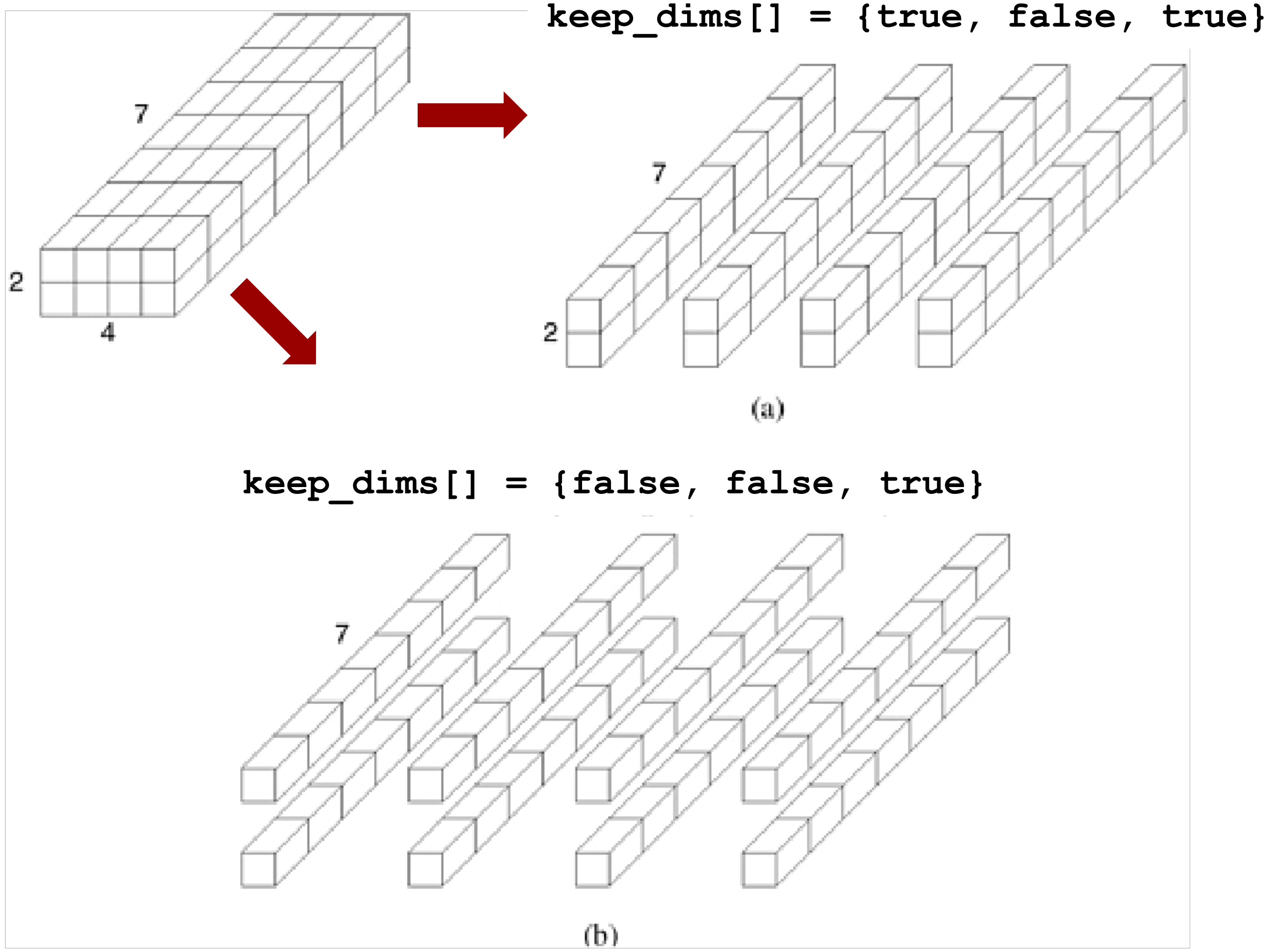
- It is very common that one wants to split a Cartesian topology along certain dimensions.
- For example, we may want to create a group for the columns or rows of a matrix.

```
int MPI_Cart_sub(MPI_Comm comm_cart,
   int *keep_dims, MPI_Comm *comm_subcart)
```

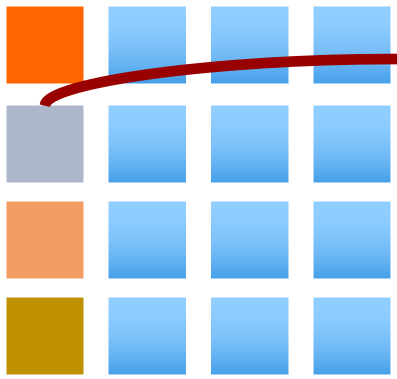
- `keep_dims` boolean flag that determines whether that dimension is retained in the new communicators or split, e.g., if false then a split occurs.

## Example

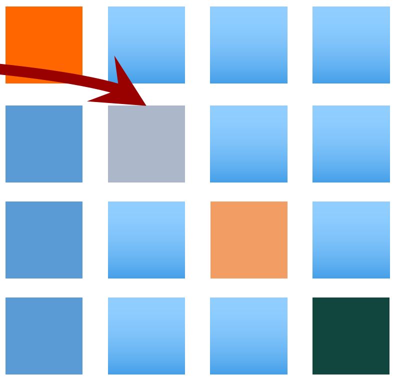

x
z
y


keep_dims[] = {true, false, true}
7
2
4
7
2
(a)
keep_dims[] = {false, false, true}
7
(b)

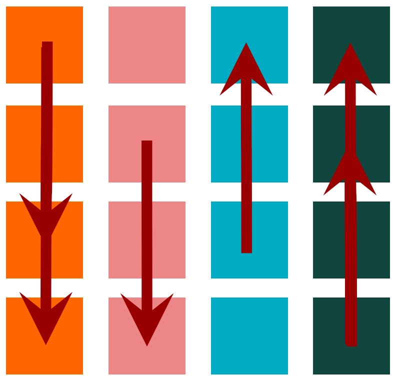
# Application example: 2D partitioning



First column contains b



Send b to the diagonal processes



Send b down each column. **Broadcast!**

Start with 2D communicator

Use column group

## 2D topology for matrix

```c
int periods[2];
periods[ROW] = periods[COL] = 1;
/* We will use wrap-around connections. */

MPI_Comm comm_2d;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);

/* Get my rank in the new topology */
int my2drank;
MPI_Comm_rank(comm_2d, &my2drank);

/* Get my coordinates */
int mycoords[2];
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
```

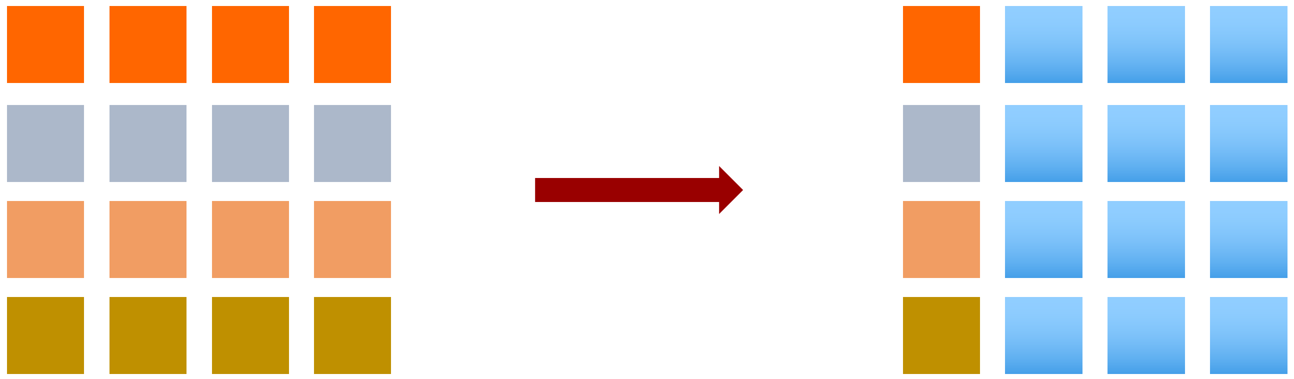
## Send to diagonal block

```c
// Send to diagonal block
if(mycoords[COL] == 0 && mycoords[ROW] != 0) {
  /* I'm in the first column */
  int drank;
  int coords[2];
  coords[ROW] = mycoords[ROW];
  coords[COL] = mycoords[ROW]; // coordinates of diagonal block
  MPI_Cart_rank(comm_2d, coords, &drank); // 2D communicator
  /* Send data to the diagonal block */
  MPI_Send(&b[0], nlocal, MPI_FLOAT, drank, 1, comm_2d);
}

// Receive from column 0
if(mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
  /* I am a diagonal block */
  int col0rank;
  int coords[2];
  coords[ROW] = mycoords[ROW];
  coords[COL] = 0; // Receiving from column 0
  MPI_Cart_rank(comm_2d, coords, &col0rank); // 2D communicator
  MPI_Recv(&b[0], nlocal, MPI_FLOAT, col0rank, 1, comm_2d,
           MPI_STATUS_IGNORE);
}
```

## Column-wise broadcast

```
/* Create the column-based sub-topology */
MPI_Comm comm_col;
int keep_dims[2];
keep_dims[ROW] = 1;
keep_dims[COL] = 0;
MPI_Cart_sub(comm_2d, keep_dims, &comm_col);

/* Broadcast inside column */
int drank;
int coord = mycoords[COL]; // Coordinate in 1D column topology
MPI_Cart_rank(comm_col, &coord, &drank);
MPI_Bcast(&b[0], nlocal, MPI_FLOAT, drank, comm_col);
```

## matvec2D



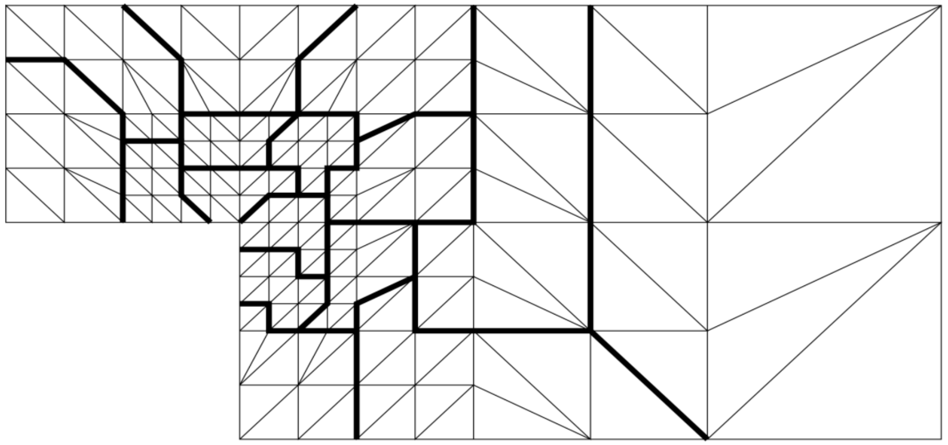
Reduction!
Use row group

**See MPI code:** `matvec2D/`

## Code for row reduction

```c
/* Create the row-based sub-topology */
MPI_Comm comm_row;
int keep_dims[2];
keep_dims[ROW] = 0;
keep_dims[COL] = 1;
MPI_Cart_sub(comm_2d, keep_dims, &comm_row);

// Row-wise reduction
int col0rank;
int coord = 0; // Coordinate in 1D row topology
MPI_Cart_rank(comm_row, &coord, &col0rank);
MPI_Reduce(&px[0], &x[0], nlocal, MPI_FLOAT, MPI_SUM, col0rank, comm_row);
```

## Topologies for finite-element calculations



- A typical situation is that processes need to communicate with their neighbors.
- This becomes complicated to organize for unstructured grids.
- In that case, graph topologies are very convenient. They allow defining a neighbor relationship in a general way, using a graph. Example: `MPI_Graph_create`
- Examples of collective communications:
  - `MPI_neighbor_allgather():` gather data, and all processes get the result
  - `MPI_neighbor_alltoall():` processes send to and receive from all neighbor processes