CME 213

SPRING 2012-2013

Eric Darve

Stanford University

MPI

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 are virtual — there may be no relation between the physical structure of the parallel machine and the process topology.

ADVANTAGES OF USING TOPOLOGIES

- Convenience: virtual topologies may be useful for applications with specific communication patterns that match an MPI topology structure.
- 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 into 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.
- 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.

NOTE ON PERIODIC CARTESIAN GRIDS

Example:

```
#include "mpi.h"
MPI_Comm comm_cart;
int ndims, dims[2], periods[2], reorder;
ndims = 2; dims[0] = 3; dims[1] = 2;
periods[0] = 1; periods[1] = 0; 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`; similarly, `i=-2` is mapped to `i=1`.
- There is no periodicity imposed on the second dimension. Any reference to a column index outside of its defined range results in an error.

OBTAINING YOUR RANK AND COORDINATES

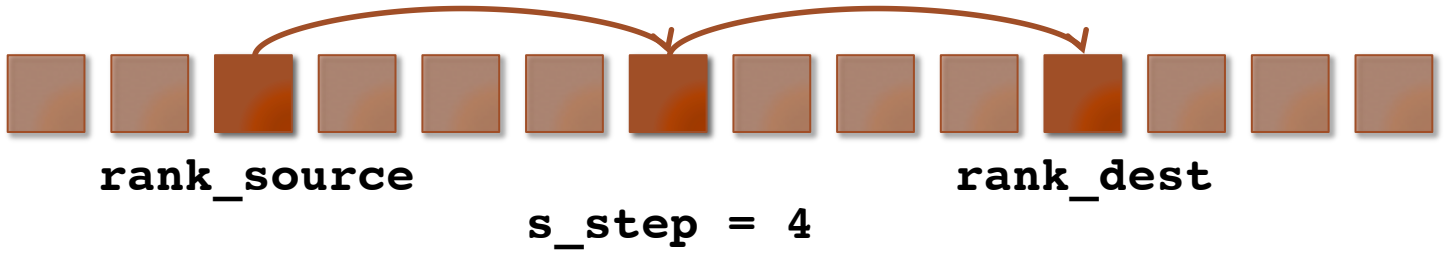
```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords,
      int *rank)
int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
      int maxdims, int *coords)
```

- 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.

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()`**.



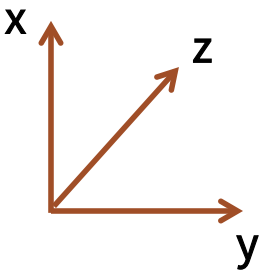
SPLITTING A CARTESIAN TOPOLOGY

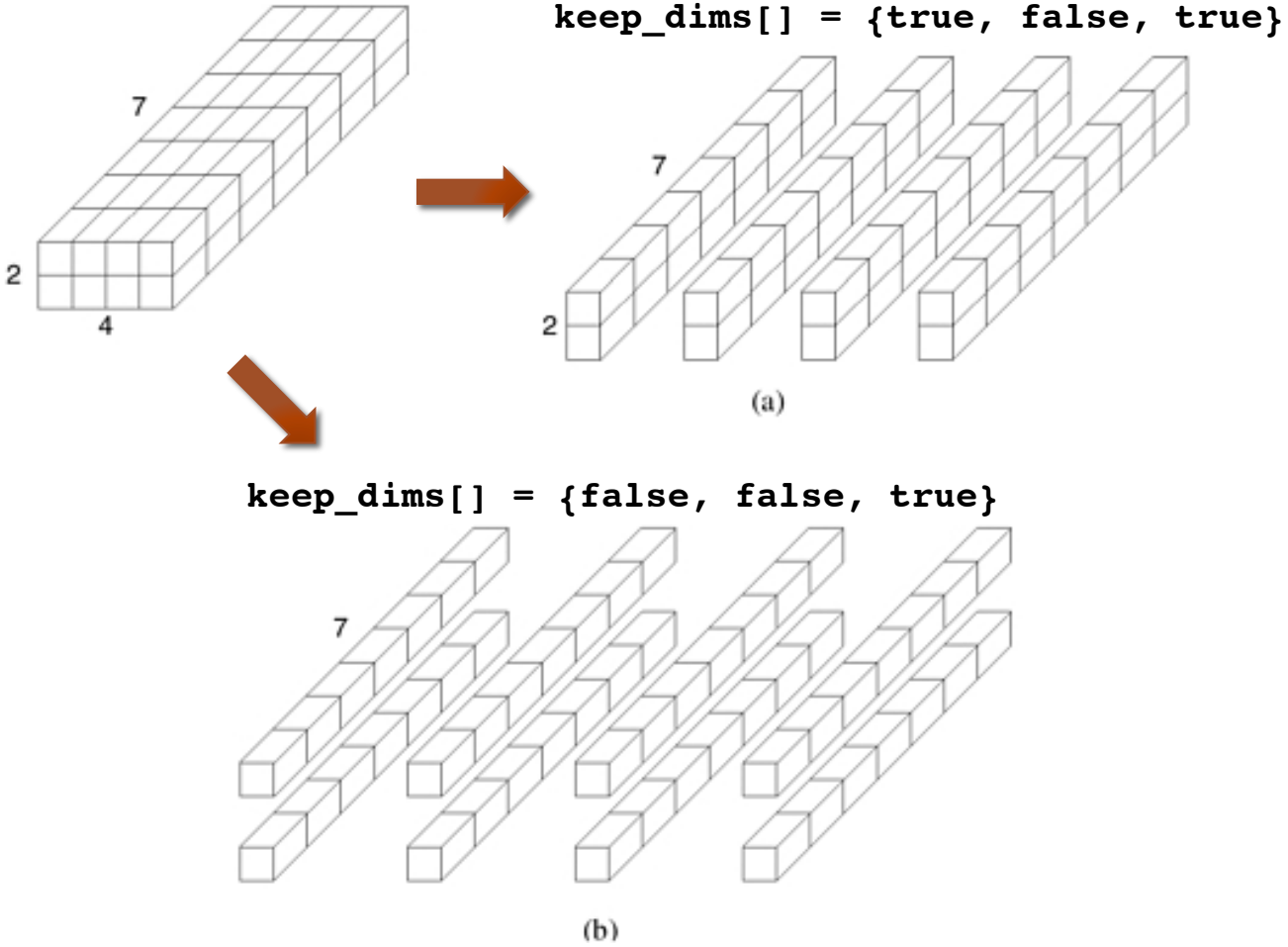
- It is very common that one wants to split a Cartesian topology along certain dimensions.
- For example, we may want to create a group for the columns or rows of a matrix.

```
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


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

MATRIX-VECTOR PRODUCT WITH 2D PARTITIONING

MATRIX-VECTOR PRODUCT

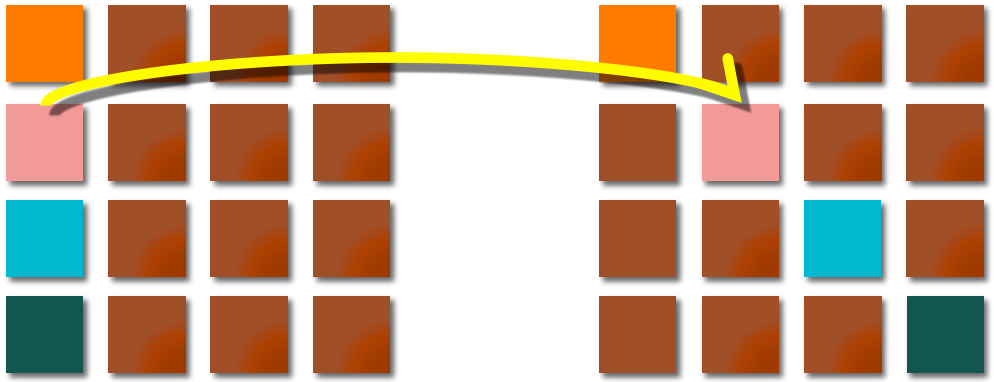
- Let's apply all these functions for a matrix-vector product:

$$x = A\ b$$

- Cartesian topology is used to partition the processes into a 2D grid.
- Groups and communicators are used for collective communication along rows and columns of the matrix.

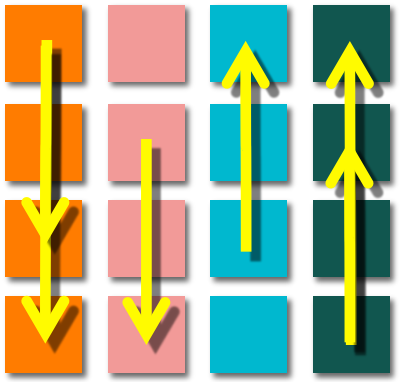
See MPI code.

STEP 1: COMMUNICATE B



First column contains b

Send b to the diagonal processes

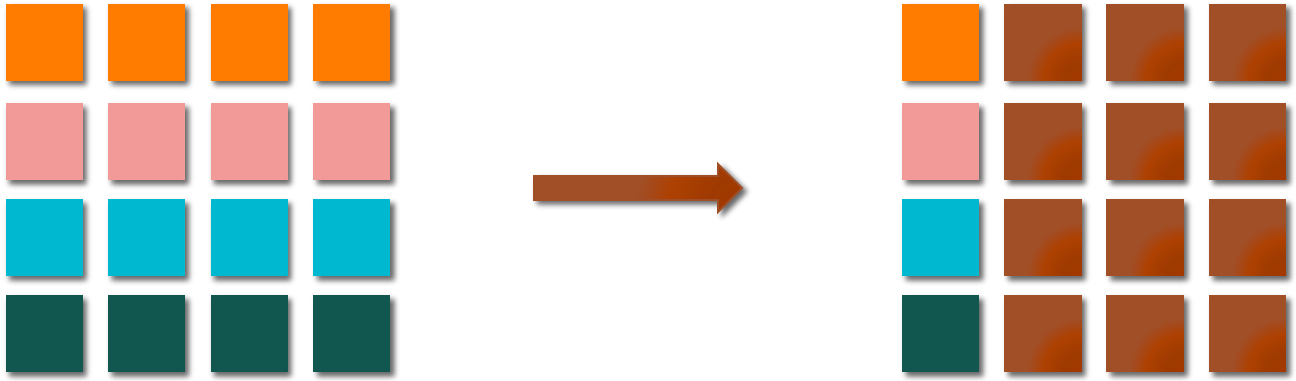


Send b down each column.
This step requires communicators and a broadcast.

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.
This requires communicators and a reduce().

Performance metrics

WHY PERFORMANCE METRICS?

Understanding the performance of a code is important:

- to develop efficient code
- understand the bottlenecks of a code
- compare algorithms in a meaningful way, e.g., matrix-vector products using different partitioning schemes for the matrix.

The total runtime $T_p(n)$ can be broken down, generally speaking, into the following categories:

- Local computations
- Data exchange and communication
- Waiting time because of load imbalance

THE BASIC CONCEPT: SPEED-UP

- This quantity measures how much faster the code runs because we are using many processes.
- Define:
  $T^\star(n)$
  the optimal (reference) running time with a single process.
- Define:
  $T_p(n)$
  the running time with p processes.
- The speed-up is then the ratio:

$$S_p(n) = \frac{T^\star(n)}{T_p(n)}$$

- We expect this number to go up as p as we keep increasing the number of processes.

A MORE APPROPRIATE CONCEPT: EFFICIENCY

- The previous definition has a problem. As p increases, we want to know whether the speed-up scales as p or not.
- This might be difficult to assess from a plot. Ideally the speed-up is a straight line.
- It is therefore more convenient to look at the efficiency:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^\star(n)}{pT_p(n)}$$

- Ideally that quantity is simply a constant as p increases. That is easier to read from a plot.
- The maximum value for efficiency is 1 (except in some rare circumstances because of cache effects).

AMDAHL'S LAW

- Although this is a crude model, it can provide a general sense of what speed-up can be achieved. This is a good measure to understand how much of the code one should try to parallelize.
- Assume that a fraction f of the code is executed sequentially. Then the speed-up is given by:

$$S_p(n) = \frac{T^\star(n)}{f\,T^\star(n) + ((1-f)/p)\,T^\star(n)} = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$

- Note the assumption: regardless of the problem size, the serial vs parallel fraction of the code is the same.
- This means that the speed-up has an upper bound. The efficiency must go to 0 eventually. In most cases, this statement is true as p increases.
- However we are saved by another result: f typically decreases with n, that is the fraction of parallel work increases with n.
- This is why we can still benefit from Peta and Exaflop machines with 10,000+ cores.