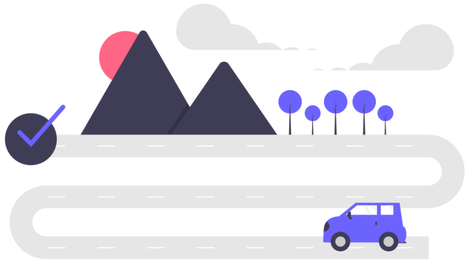
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Programs must be written for people to read, and only incidentally for machines to execute." — Abelson and Sussman

Stanford University

Recap



- Distributed memory and MPI (Message Passing Interface)
- Collective communications
- Mapping and binding
- Deadlocks in point-to-point communications

Non-blocking MPI communications

Point-to-point communication

- There are a few additional concepts in point-to-point communications that are important to understand.
- Two main orthogonal concepts:
  - Blocking/non-blocking
  - Synchronous/asynchronous (less common)

What we have seen so far

- **Blocking:** function only returns when buffers are ready.
- **Asynchronous:** operations can happen at different times on sender and receiver.

Advantages of blocking communications

- Simple to use.
- Issue command; once code returns, you know that the task is done (at least the resource is usable).
- Efficient.
- However, this is restrictive.

Improve concurrency between computations and communications

- When communications are happening, you probably want to do something else, such as do some useful computation or issue other communications.

- This is called **overlapping communication and computation.**

Non-blocking workflow

- Instead of blocking and waiting for some data to perform the next task, **you want to work on all the tasks for which data is available.**
- Then, check periodically for the **status of communications.**
- Non-blocking communications are **safer** and help **avoid deadlocks.**
- They consume **more memory,** which can be hazardous in some situations.

How to use non-blocking communications

Online documentation

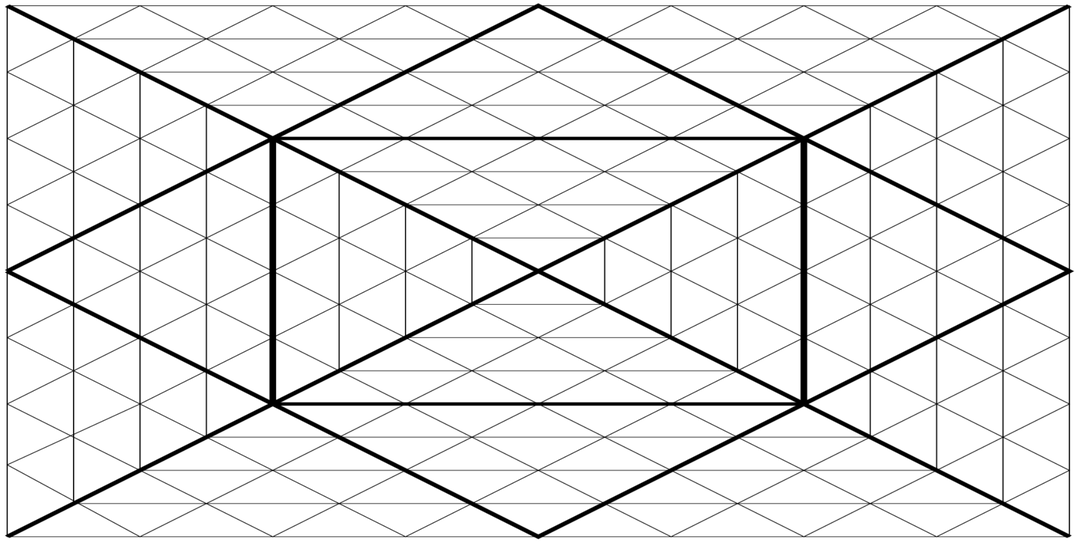
`MPI_Isend`

`MPI_Irecv`

`MPI_Test`

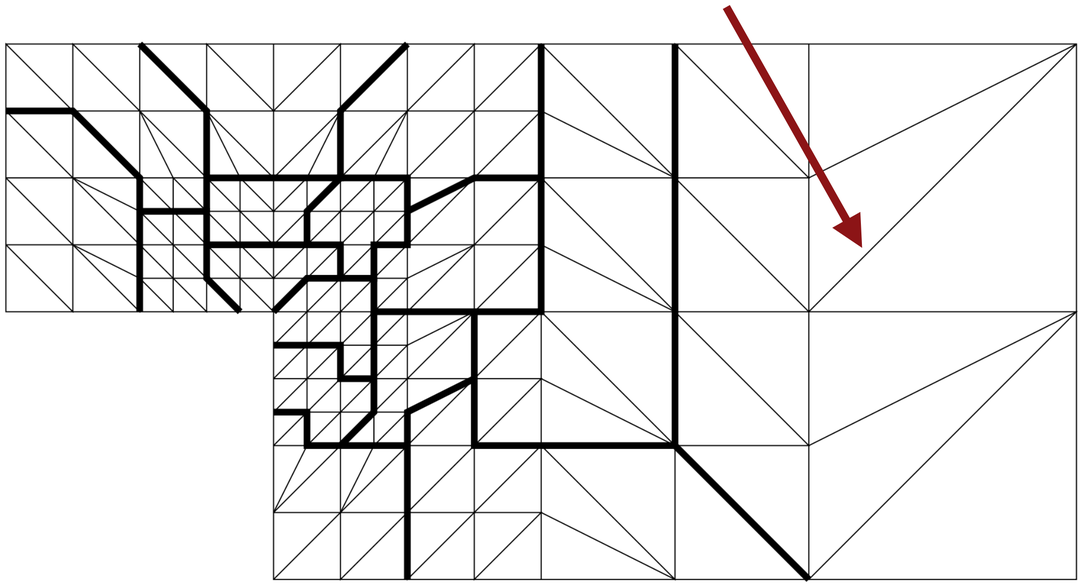
`MPI_Wait`

Motivating example



2D rectangular domain

Sub-domain = 1 process

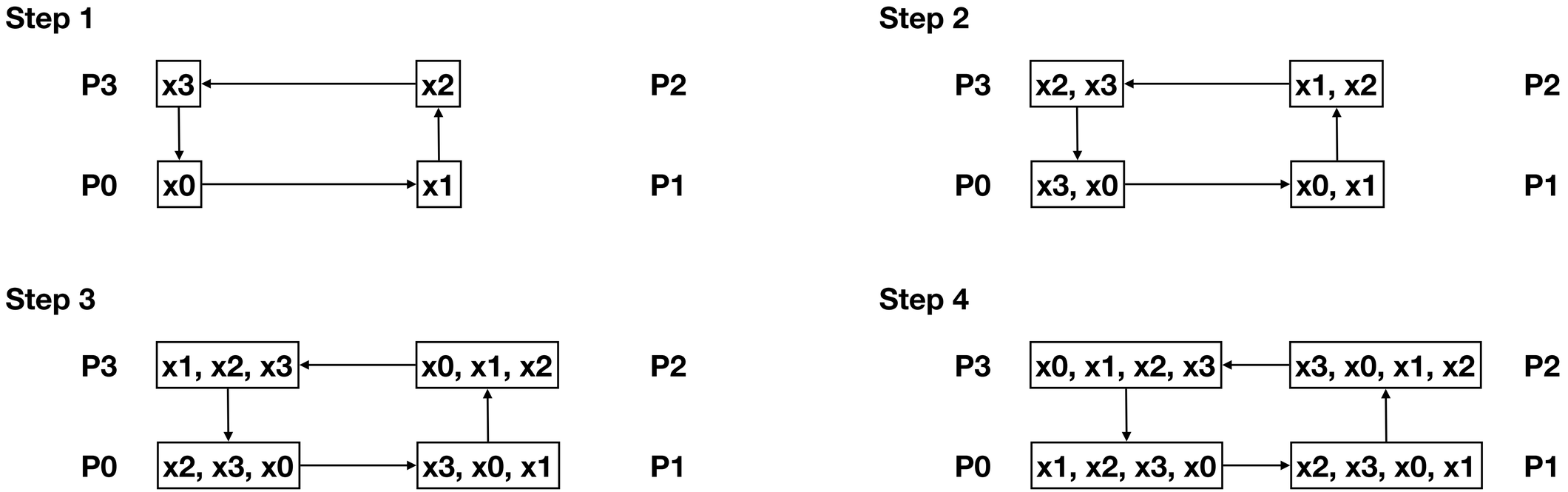


General domain

- MPI communication using Send and Recv. If buffers are used the program will run correctly. Otherwise, deadlocks are possible.

- It's easier to launch all the communications in a non-blocking manner, and then test to check whether they have been completed or not.

Gather ring using non-blocking communications



All gather collective communication using a ring topology. We use non-blocking communications for this.

Non-blocking communication example

```
// Send to the right: Isend
int i_send = (rank - i + nproc) % nproc;
assert(i_send >= 0 && i_send < nproc);
int *p_send = &numbers[i_send];
printf("MPI_Isend %2d to   %2d: %2d\n", rank, rank_receiver, i);
MPI_Isend(p_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD,
          &send_req[i]);
// We can proceed; no need to wait now.
// Receive from the left: Recv
int i_recv = (rank - i - 1 + nproc) % nproc;
assert(i_recv >= 0 && i_recv < nproc);
int *p_recv = &numbers[i_recv];
printf("MPI_Recv  %2d from %2d: %2d\n", rank, rank_sender, i);
MPI_Recv(p_recv, 1, MPI_INT, rank_sender, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
// We need to wait; we cannot move forward until we have that data.
```

- Send is non-blocking.
- Recv is blocking because the data is needed for the next iteration.

Wait

```
for (int i = 0; i < nproc - 1; ++i) {
  // Wait for communications to complete
  MPI_Status status;
  MPI_Wait(&send_req[i], &status);
}
```

- Wait can be used at the end to make sure all non-blocking communications have been completed.
- Wait is a blocking operation.

Key non-blocking functions

MPI_Isend

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

`MPI_Isend` starts a non-blocking send. Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. **The sender should not modify any part of the send buffer after a nonblocking send operation is called until the send completes.**

MPI_Irecv

```
int MPI_Irecv(void *buf, int count,
              MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request)
```

A nonblocking receive call indicates that the system may start writing data into the receive buffer.

The receiver should not access any part of the receive buffer after a nonblocking receive operation is called until the receive completes.

MPI_Test

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

A call to `MPI_Test` returns `flag = true` if the operation identified by request is complete.

In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to `MPI_REQUEST_NULL`.

The call returns `flag = false`, otherwise.

MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

A call to `MPI_Wait` returns when the operation identified by request is complete.

If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to `MPI_Wait` and the request handle is set to `MPI_REQUEST_NULL`.

MPI send modes
Optimization!

Algorithmic variants

Three main algorithmic variants:

1. `Buffered`—MPI uses a buffer to avoid blocking

2. `Eager`—MPI will try to send data immediately whether or not a Recv has been posted. Works well for small messages.

3. `Rendez-vous`—Send data only when Recv has been posted; buffering is not needed; requires a synchronization of the two processes

MPI standard Send

`MPI_Send`

But the user has no control over the implementation. The decision is taken by the MPI library.

Message size	Strategy
Small	`eager`
Large	`rendez-vous`

MPI_Bsend

Send with user-specified buffering.

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

The user must have provided buffer space using:

```
MPI_Buffer_attach(void *buf, int size)
```

MPI_Ssend

Synchronous send; rendez-vous.

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

MPI_Ssend performs a synchronous mode, blocking send. Blocks until buffer in sending task is free for reuse and **destination process has started to receive the message.** Best performance for data transfer.

Can be used to detect potential deadlocks hidden by MPI buffering.

MPI_Rsend

Ready send; `eager`.

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

A ready send may only be called if the user can guarantee that a receive is already posted. It is an error if the receive is not posted before the ready send is called.

Uncommon.

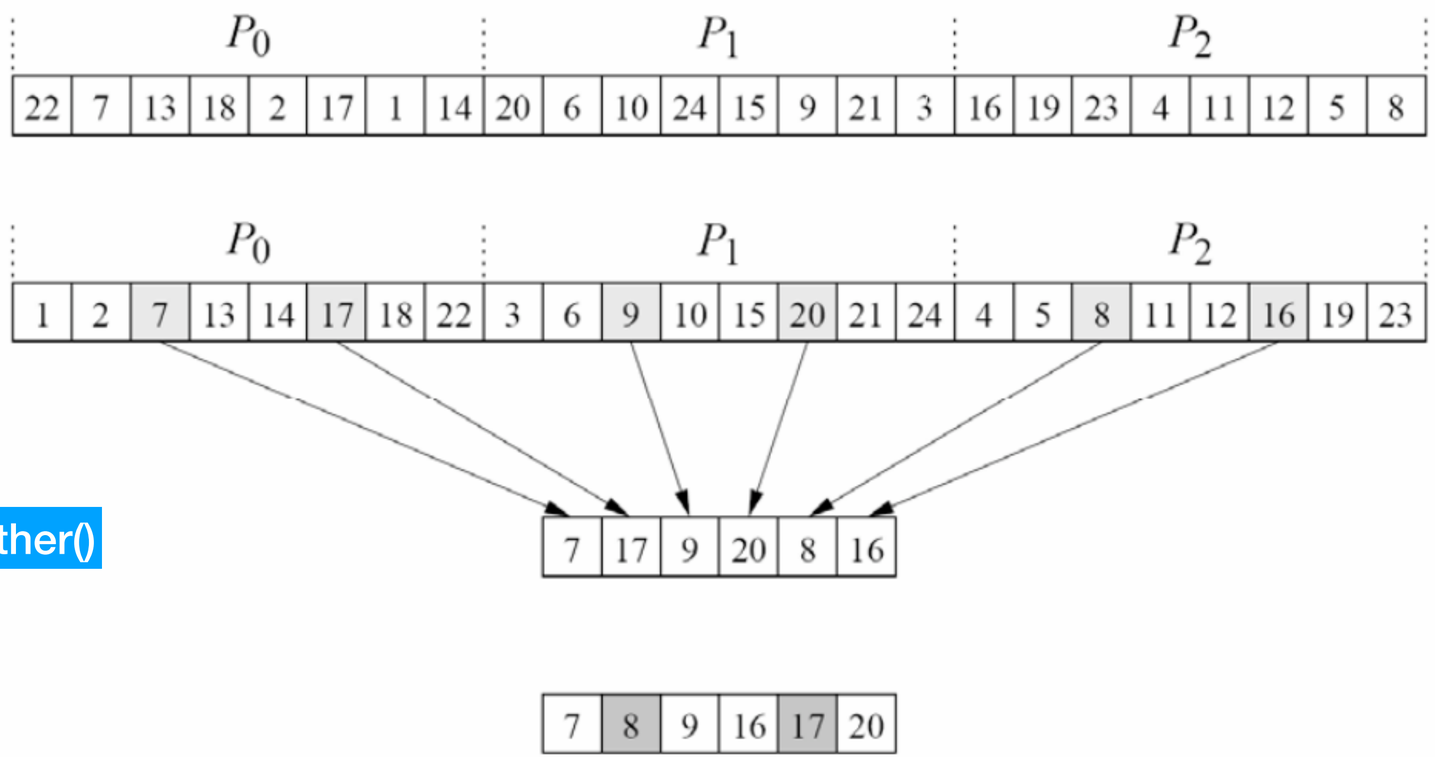
Summary

Send mode	MPI function	Completion condition
Standard send	`MPI_Send`	Message sent (receiver state unknown)
Buffered send	`MPI_Bsend`	Always completes immediately
Synchronous send	`MPI_Ssend`	Only completes when the receive operation has started
Ready send	`MPI_Rsend`	May be used only when the matching receive has already been posted

Example of MPI application with collective communications
Bucket and sample sort

- Bucket sort is a simpler parallel algorithm.
- Assume we have a sequence of integers in the interval $[a, b]$.
- Split $[a, b]$ into p sub-intervals.
- Move each element to the appropriate bucket (prefix sum required again).
- Sort each bucket in parallel.
- Simple and efficient!
- A variant of this is the radix sort.
- Problem: how should we split the interval? This process may lead to intervals that are unevenly filled.
- Improved version: sample (or splitter) sort.

Distributed memory splitter sort



Key MPI functions to implement a splitter sort

- `MPI_Allgather` to gather the splitters from all processes.
- The AllToAll communication is a bit complicated because the number of data per process is not the same. Hence, we need `MPI_Alltoallv`.
- 2 steps:
  1. `MPI_Alltoall` to gather **the size of the data to be sent.**
  2. Then `MPI_Alltoallv` for the actual data transfer.

Performance measurement: efficiency

With p processes, we expect a running timing that is p times faster than the sequential running time.

This suggests defining the efficiency as:

$$\text{Efficiency} = \frac{T_{\text{seq}}}{pT_{\text{par}}}$$

The efficiency should ideally be constant as we increase p.

Benchmark on icme-gpu

1 process;	efficiency:	0.592842;	runtime seq:	3.14523, par:	5.30535
2 processes;	efficiency:	0.674442;	runtime seq:	3.29324, par:	2.44145
4 processes;	efficiency:	0.680111;	runtime seq:	3.38574, par:	1.24455
8 processes;	efficiency:	0.679049;	runtime seq:	3.45379, par:	0.635777
16 processes;	efficiency:	0.62502;	runtime seq:	3.37577, par:	0.337566
32 processes;	efficiency:	0.092245;	runtime seq:	3.47554, par:	1.17741
64 processes;	efficiency:	0.0396358;	runtime seq:	3.33235, par:	1.31366

Multiple nodes are used; communications go over ethernet

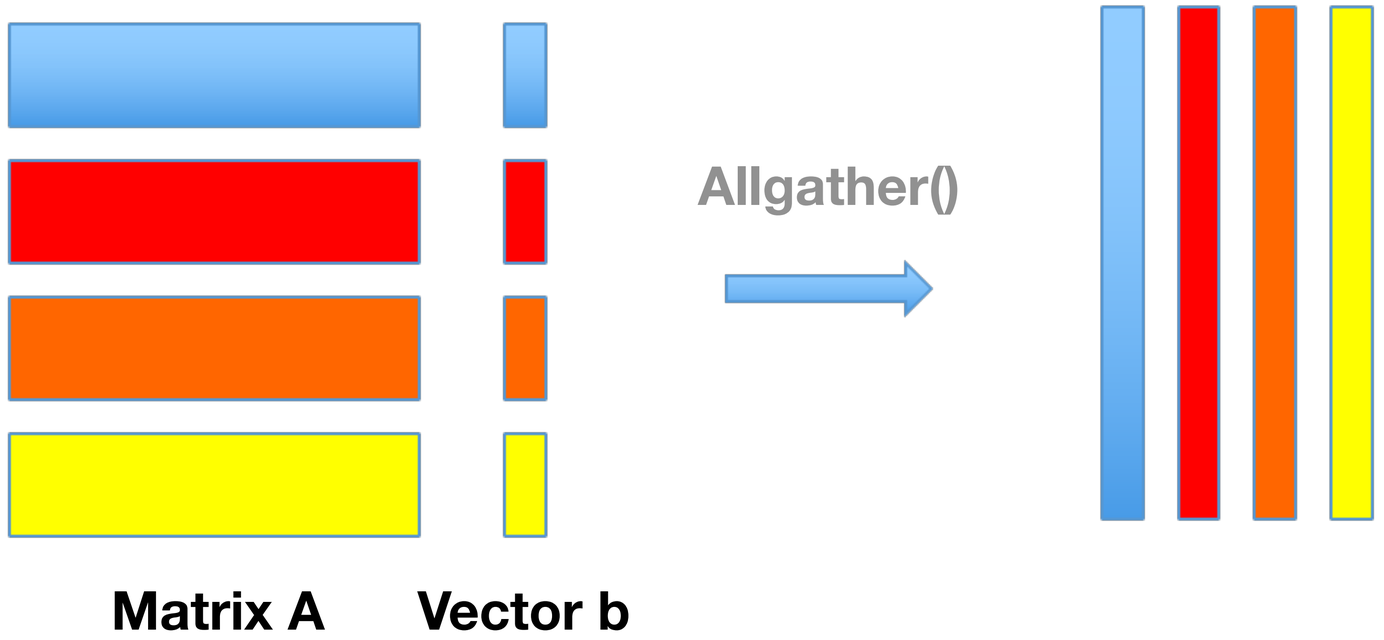
Linear algebra
Matrix-vector products

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

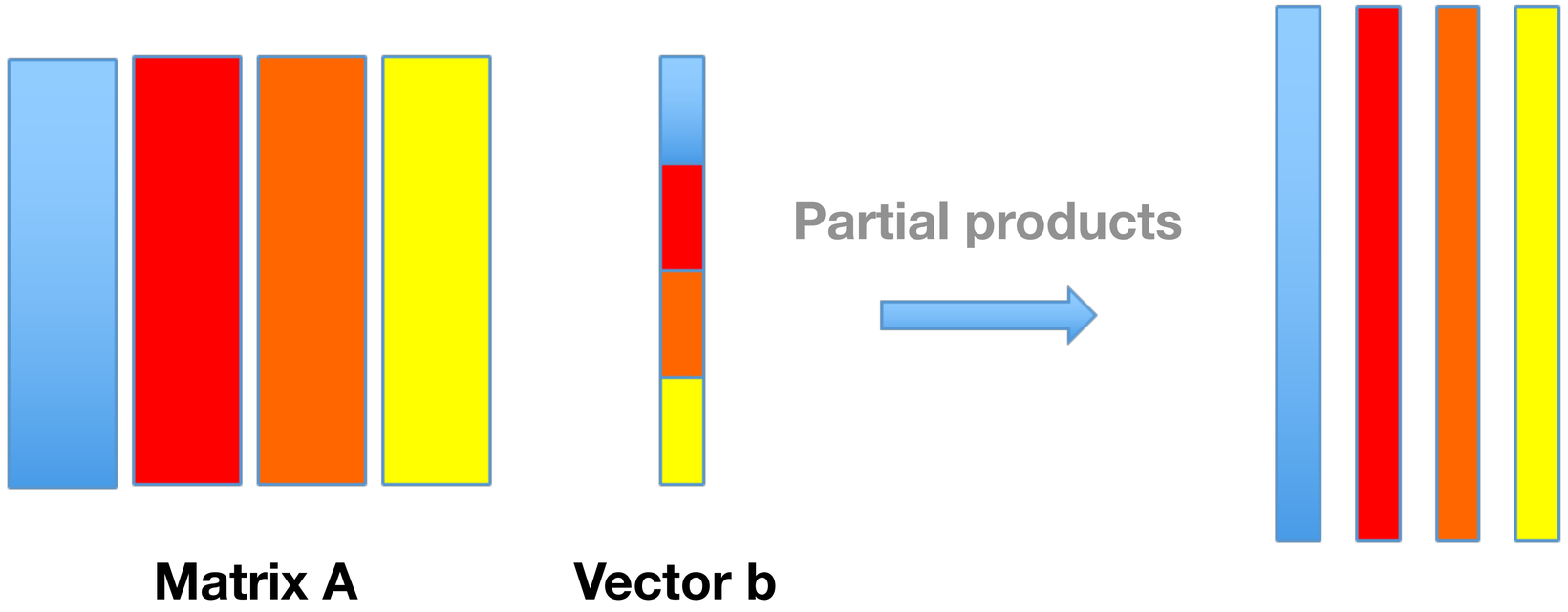
- This is the most natural.



- Step 1: replicate b on each process: `MPI_Allgather()`
- Step 2: perform product
- See MPI code: `matvecrow/`

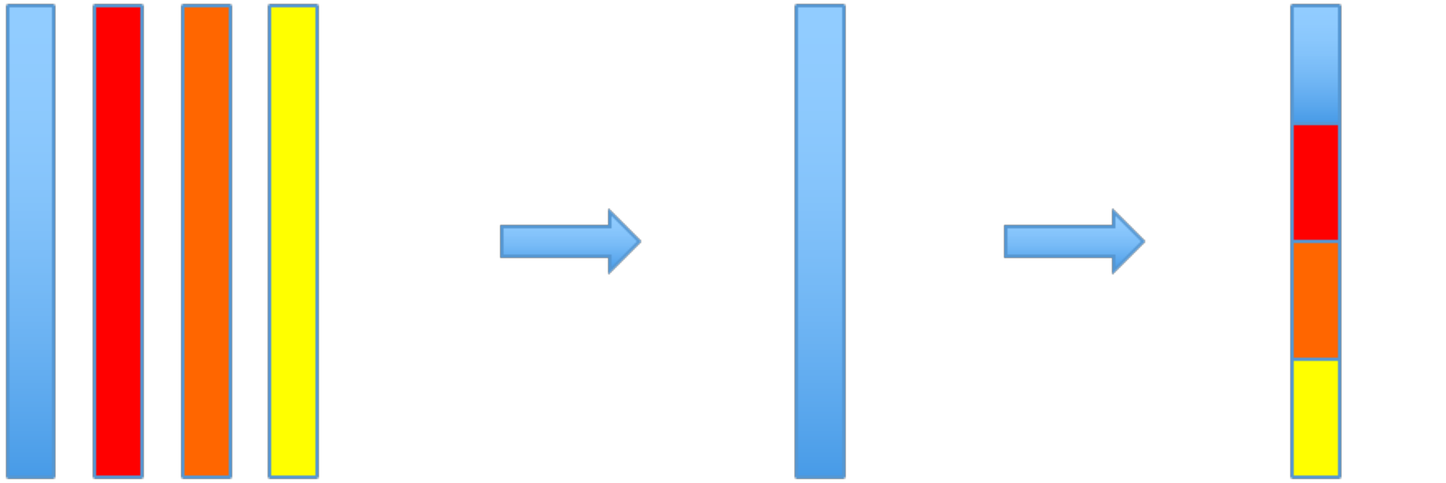
```
/* Gather entire vector b on each processor using Allgather */
MPI_Allgather(&bloc[0], nlocal, MPI_FLOAT, &b[0], nlocal, MPI_FLOAT,
              MPI_COMM_WORLD);
// sending nlocal and receiving nlocal from any other process

/* 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



Partial products **Vector Ab**

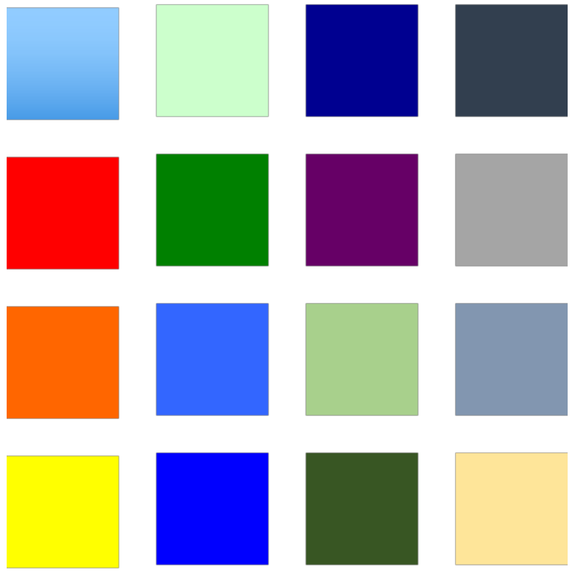
Step 2: reduce all partial results: `MPI_Reduce()`

Step 3: send sub-blocks to all processes: `MPI_Scatter()`

Steps are very similar to row partitioning.

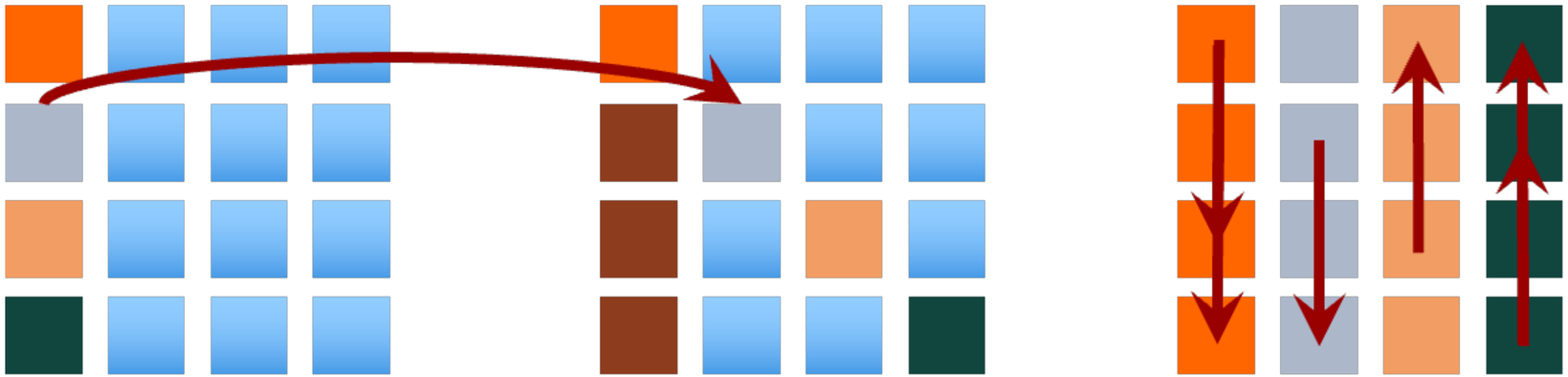
A better partitioning

- 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, theoretical analysis shows that this scheme runs faster.

Outline of algorithm: step 1



First column contains b

Send b down each column.

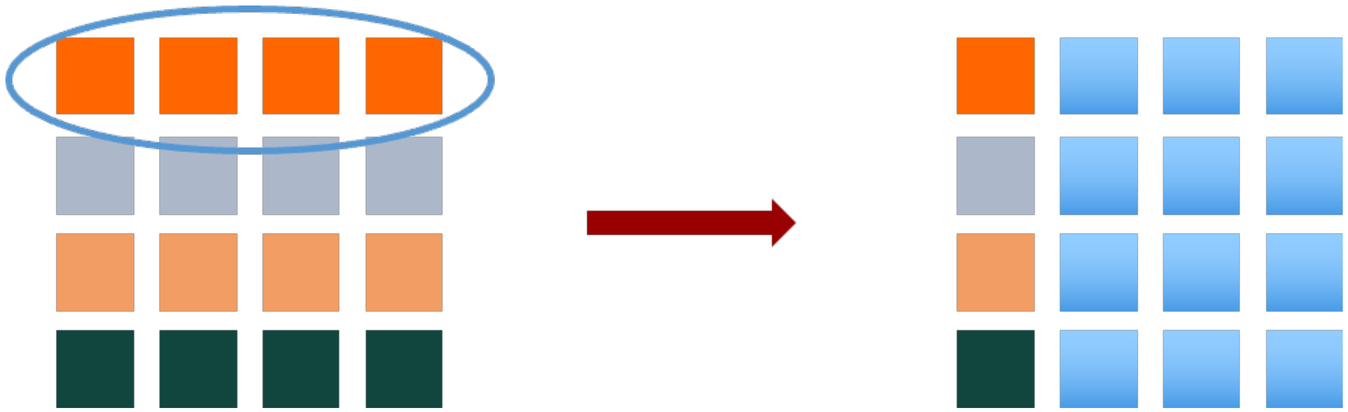
Send b to the diagonal processes

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 between both algorithms

- In both cases, the execution time for the computation part is $O(n^2/p)$.
- However, there are substantial differences for the cost of communication.

1D partitioning (Allgather): $O(\ln p + \gamma n)$.

2D partitioning (broadcast/reduction): $O\left(\ln \sqrt{p} + \gamma n \frac{\ln \sqrt{p}}{\sqrt{p}}\right)$

How to implement the 2D scheme in MPI

- 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 assign independent tasks to different groups of processes easily.
- Provide 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)
```

- Returns 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/`

mpi_group.cpp

```
/* 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 on 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.

MPI code—Periodic Cartesian grids

```
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);
```

mpi_cart.cpp

- 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`.
- If there is no periodicity imposed on the second dimension, any reference to a column index outside of its defined range results in an error message. Try it!

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 available to query your own rank.
- See MPI code: `mpi_cart/`

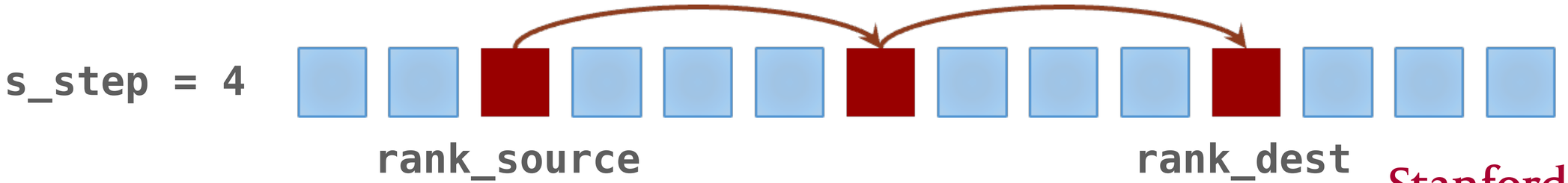
```c
/* Get my rank in the new topology */
int my2drank;
MPI_Comm_rank(comm_cart, &my2drank);

/* 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`	shift length
`rank_dest`	contains the group rank of the neighboring process in the specified dimension and distance.
`rank_source`	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

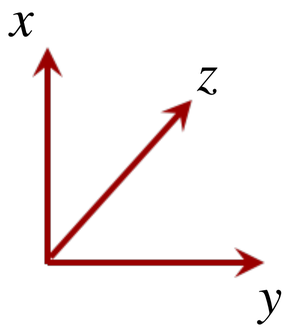
It is common to want to split a Cartesian topology along certain dimensions.

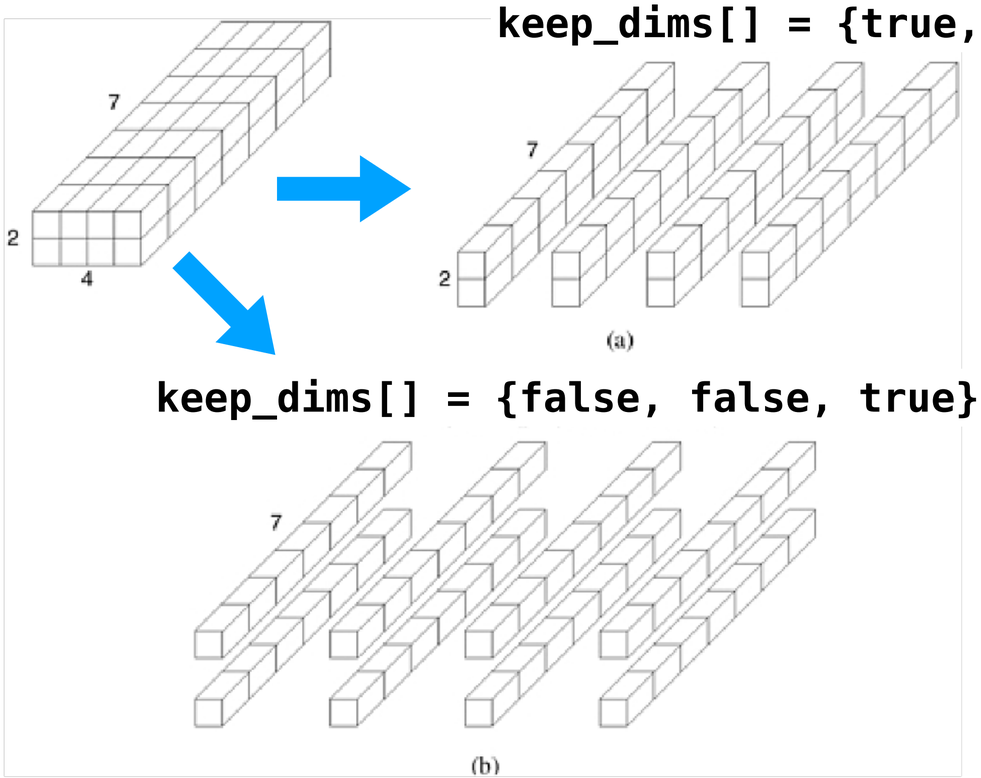
For example, we can create a group for the columns or rows of a matrix.

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

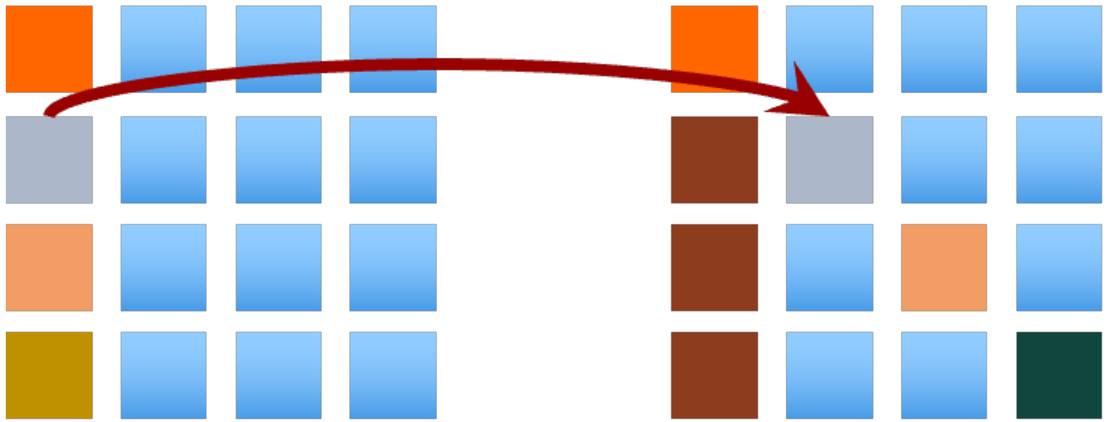
`keep_dims` boolean flag that determines whether that dimension is retained in the new communicators or split, e.g., if false then a split occurs.

Example


x
z
y

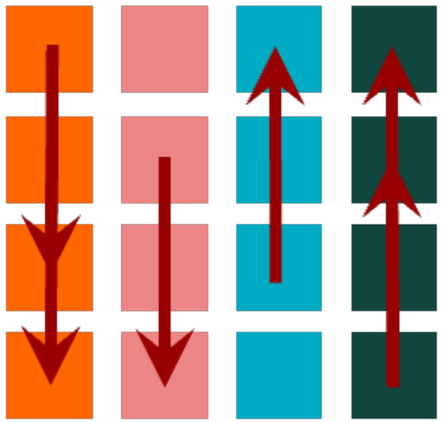

keep_dims[] = {true, false, true}
7
2
4
7
2
(a)
keep_dims[] = {false, false, true}
7
(b)

Let’s apply this to our 2D partitioning algorithm



First column contains b

Send b to the diagonal processes



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

Start with 2D communicator

Use column group

Send to diagonal block

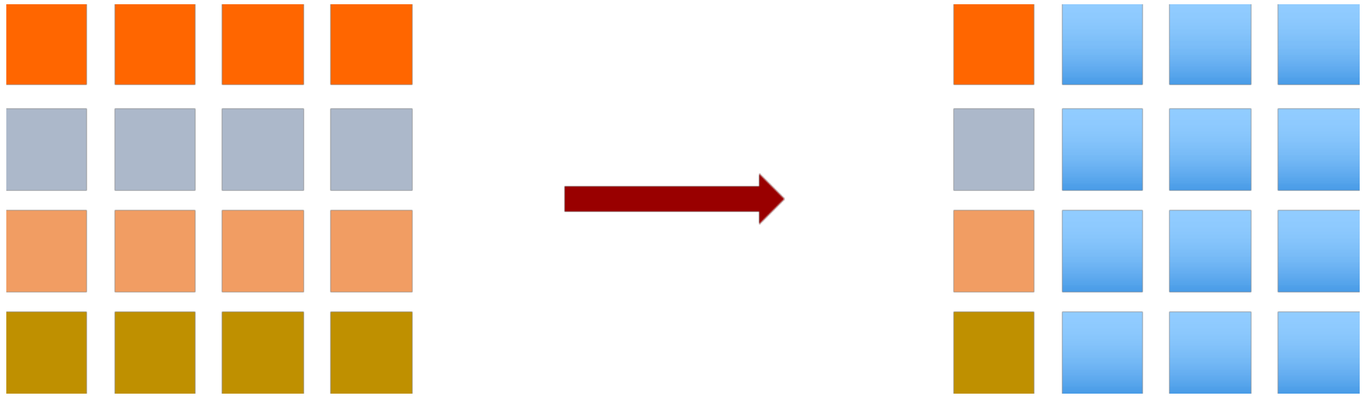
```
// Send to diagonal block
if(mycoords[COL] == 0 && mycoords[ROW] != 0) {
  /* I'm in the first column */
  int drank;
  int coords[2];
  coords[ROW] = mycoords[ROW];
  coords[COL] = mycoords[ROW]; // coordinates of diagonal block
  MPI_Cart_rank(comm_2d, coords, &drank); // 2D communicator
  /* Send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

See MPI code: `matvec2D/`

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);
```

Row-wise reduction

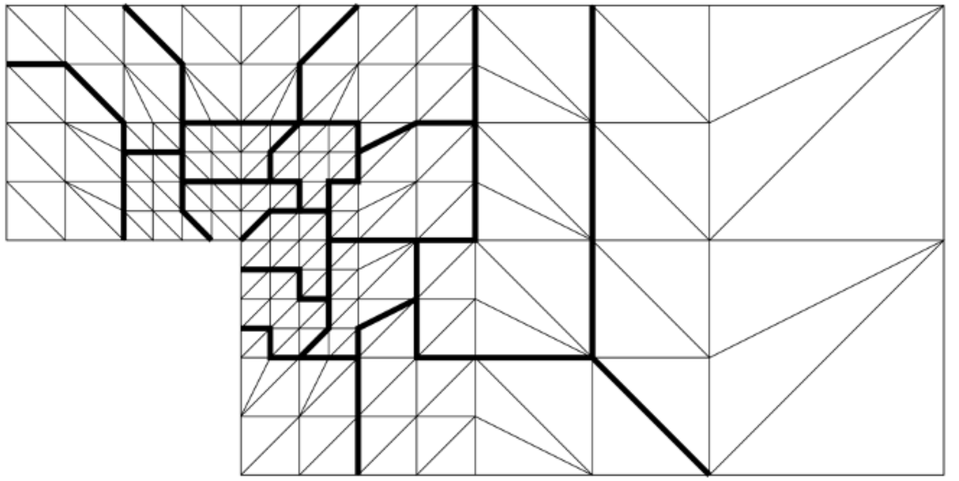


Reduction!
Use row group

MPI code for row reduction

```
/* 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.**
- 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

thank you!