中国科学院大学
University of Chinese Academy of Sciences

Introduction to MPI

Part2. *Intermediate Topics*（B）

人工智能学院
缪青海
miaoqh@ucas.ac.cn

Content

- Non-blocking communication
- Group (collective) communication
- **MPI Datatypes**
- Virtual Topology

Simple/Predefined Datatypes

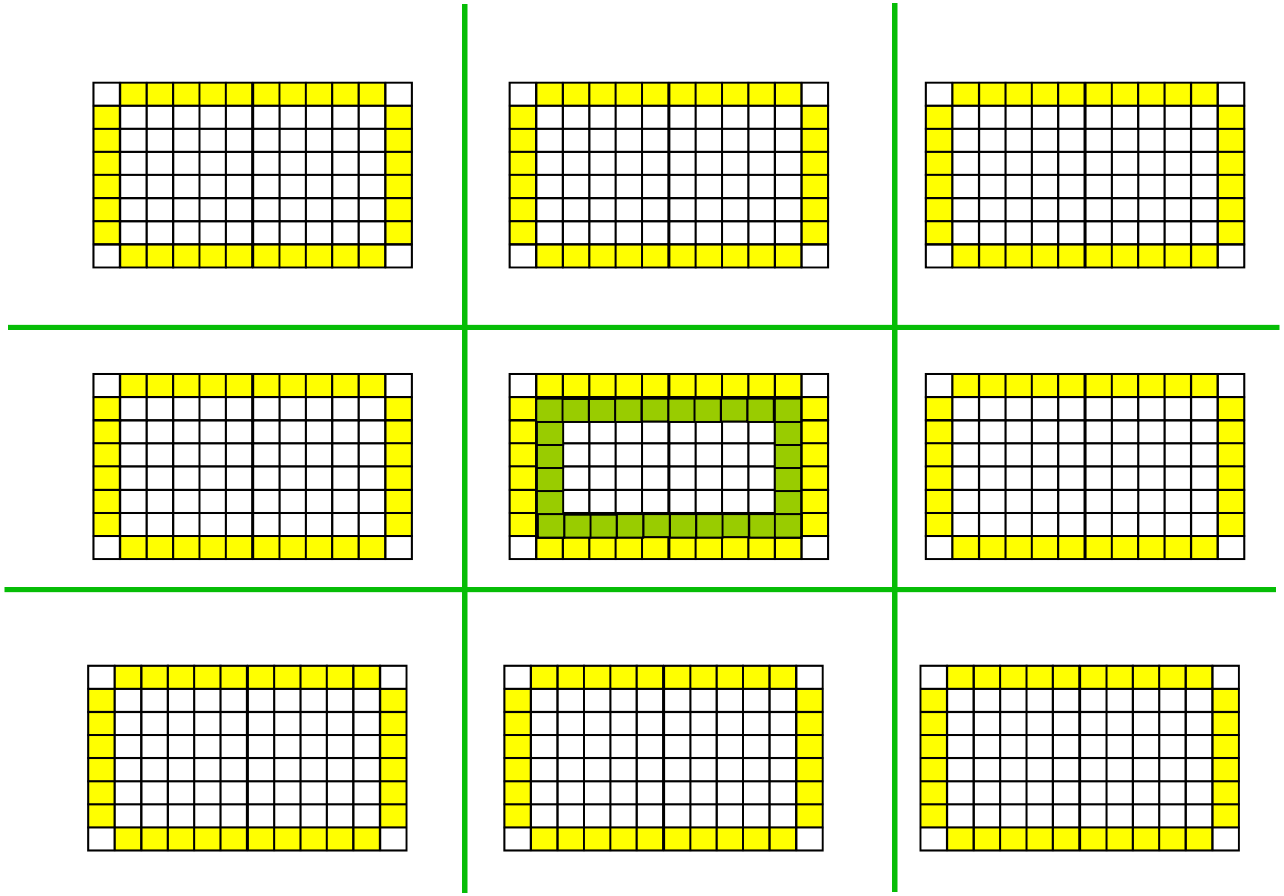
- Equivalents exist for all C, C++ and Fortran native datatypes:
  - C int → MPI_INT;
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T

Complex Datatypes

- For more complex or user-created datatypes, MPI provides routines to represent them as well:
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)
  - ……

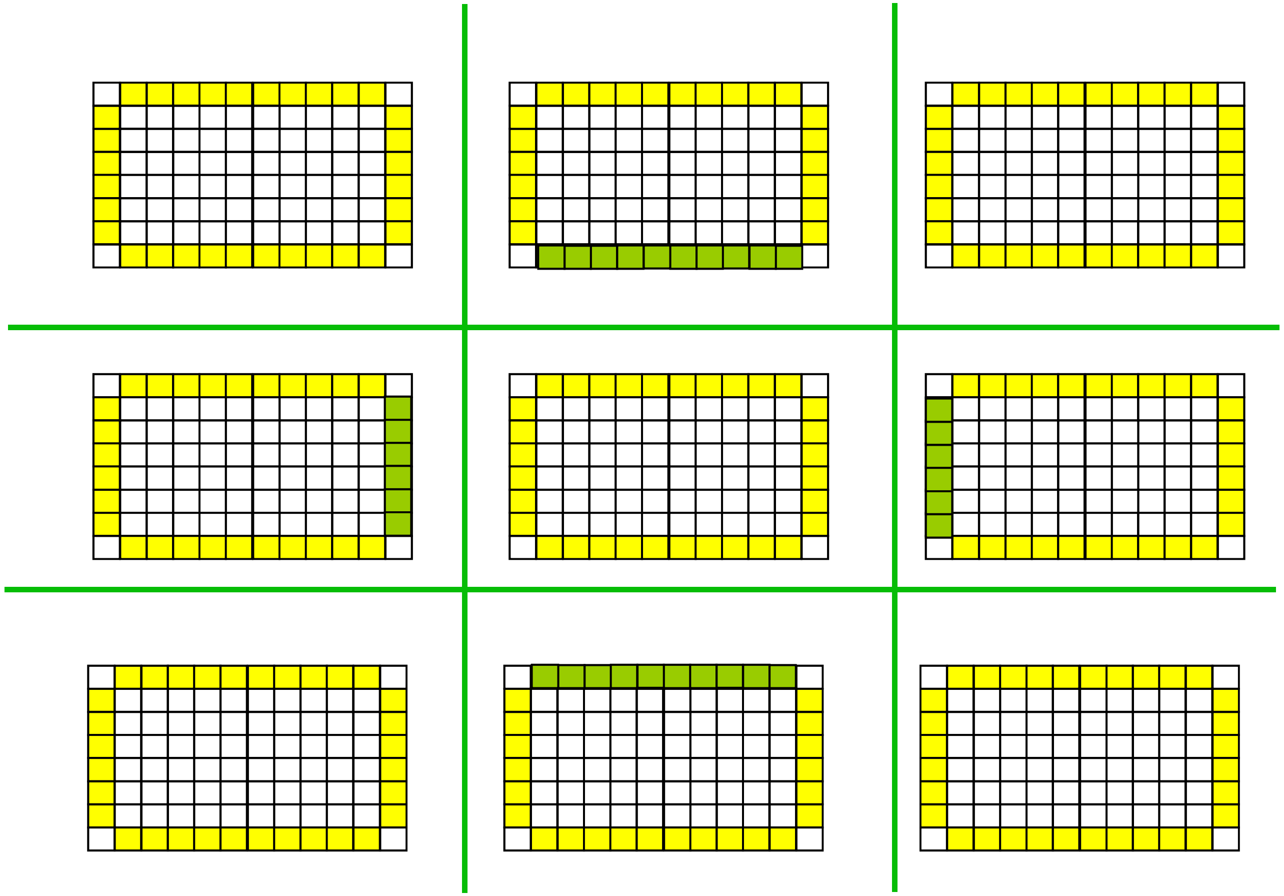
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)



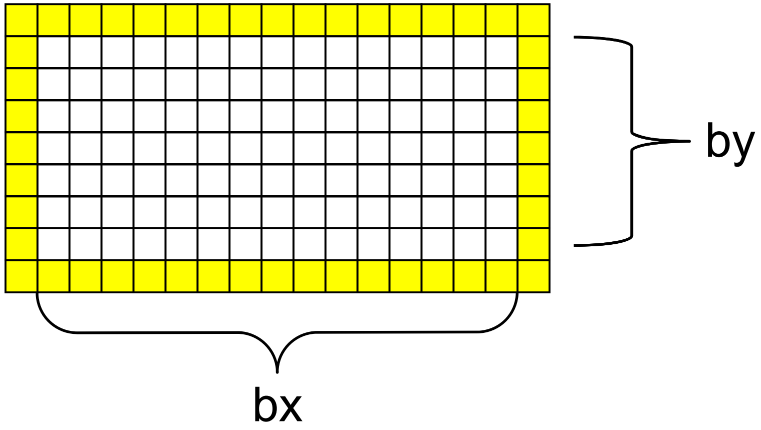
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)



The Local Data Structure

- Each process has its local "patch" of the global array
  - "bx" and "by" are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)



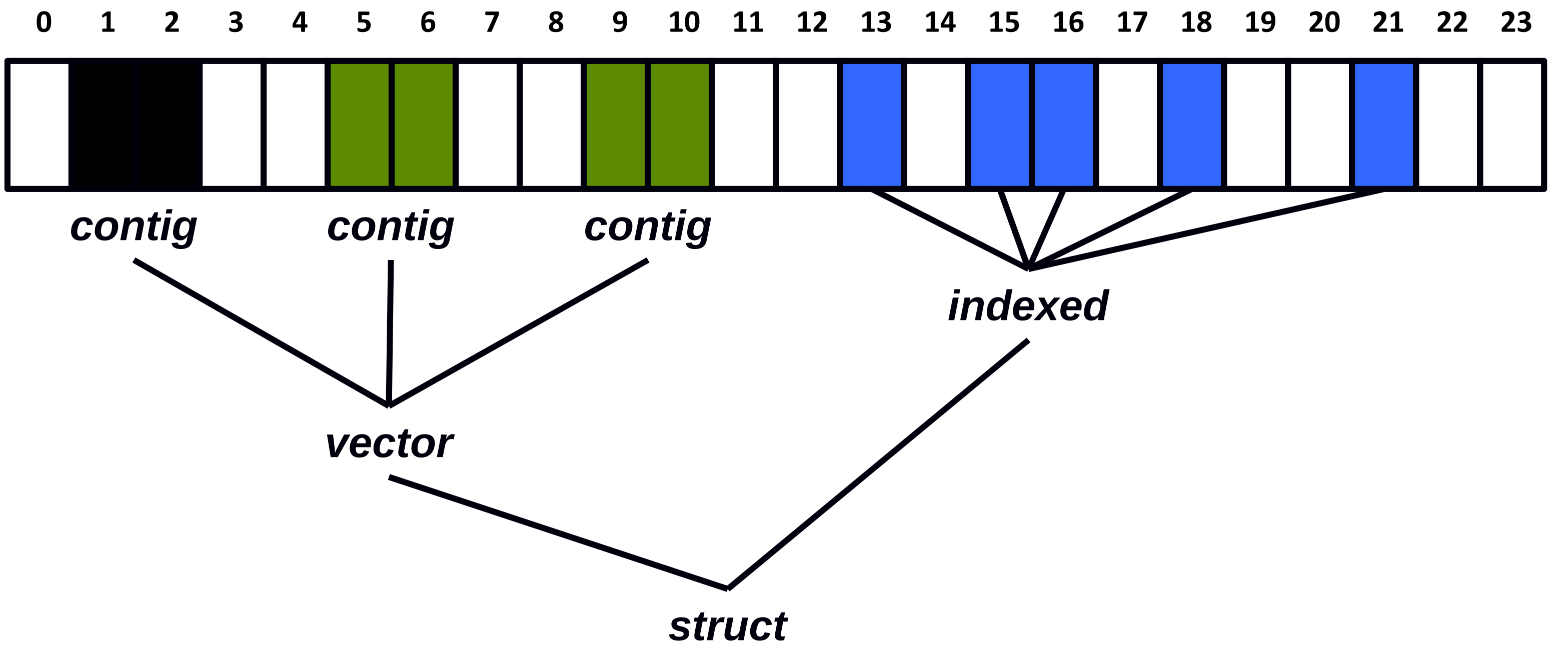
Introduction to Datatypes in MPI

- **Datatypes** allow to (de)serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O

- Several constructors allow arbitrary layouts
  - Recursive specification possible
  - *Declarative* specification of data-layout
    - "what" and not "how", leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple

更灵活 / 更复杂

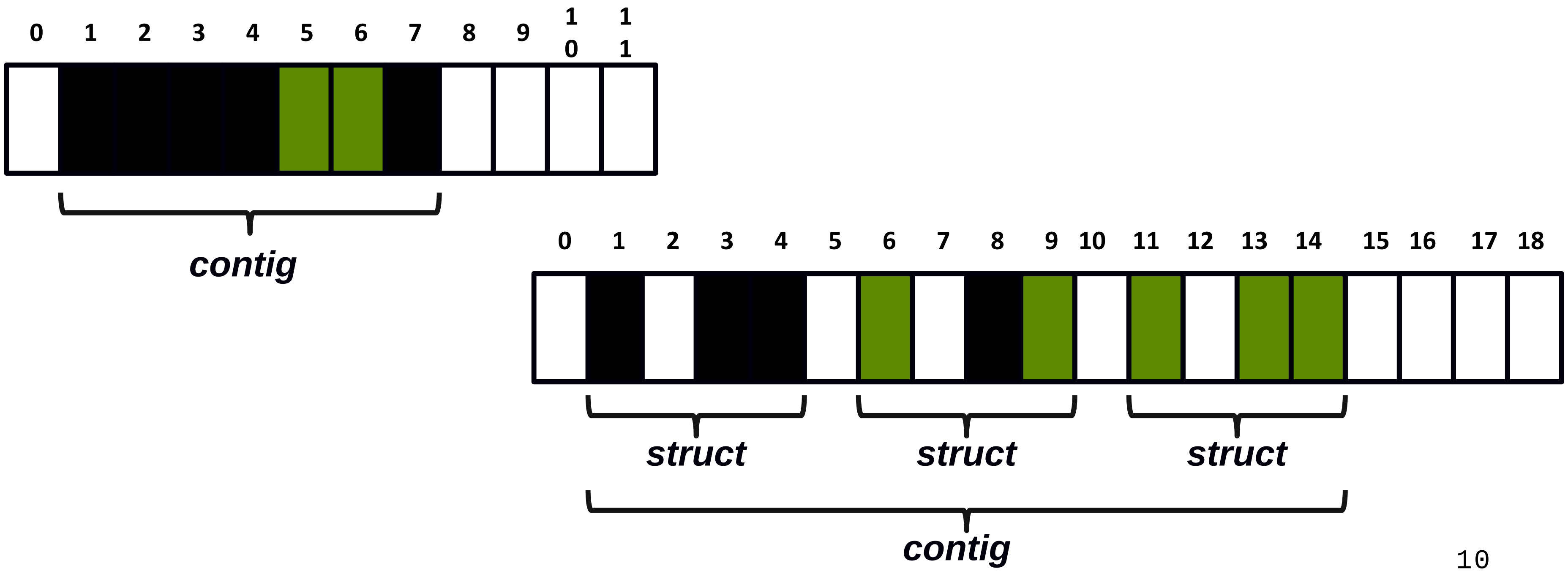
Derived Datatype Example



MPI_Type_contiguous

MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

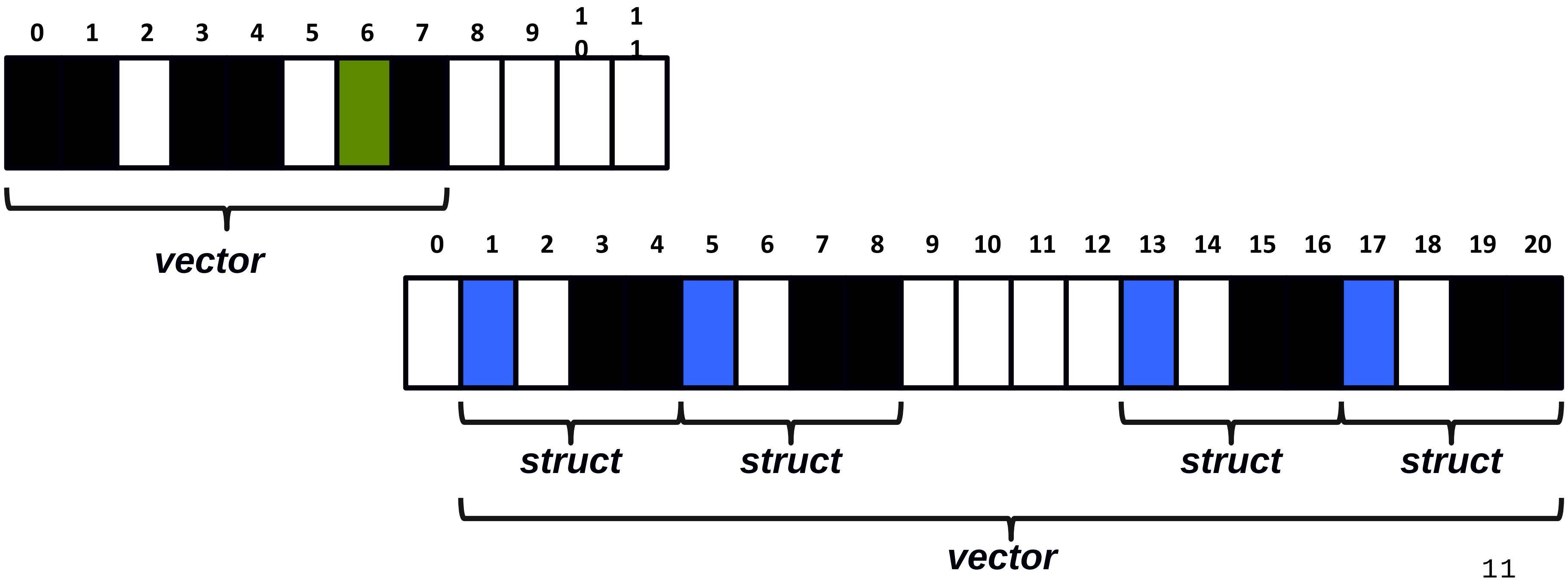
- Contiguous array of *oldtype*
- *newtype* is the datatype obtained by concatenating *count* copies of *oldtype*



MPI_Type_vector

MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)

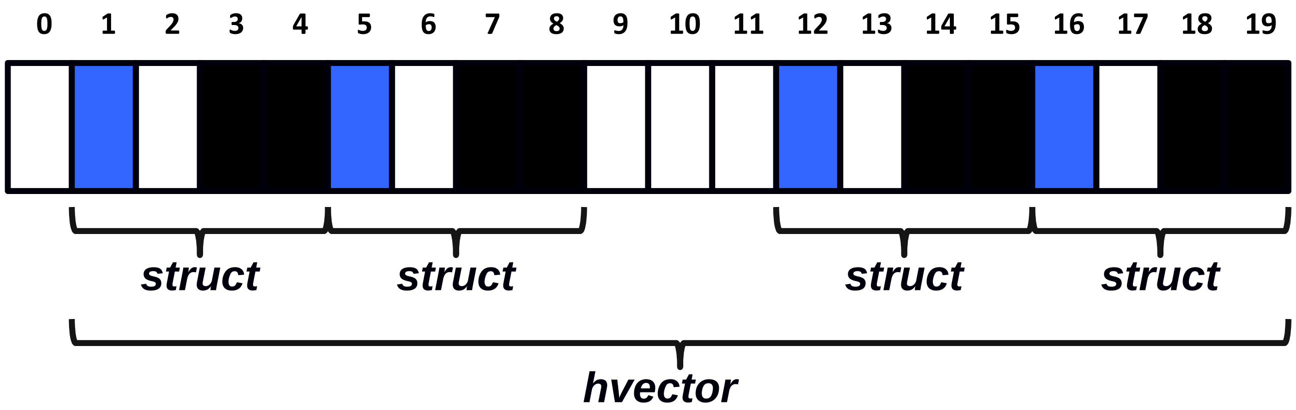
- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays



MPI_Type_create_hvector

MPI_Type_create_hvector(int count, int blocklength,
MPI_Aint stride, MPI_Datatype oldtype,
MPI_Datatype *newtype)

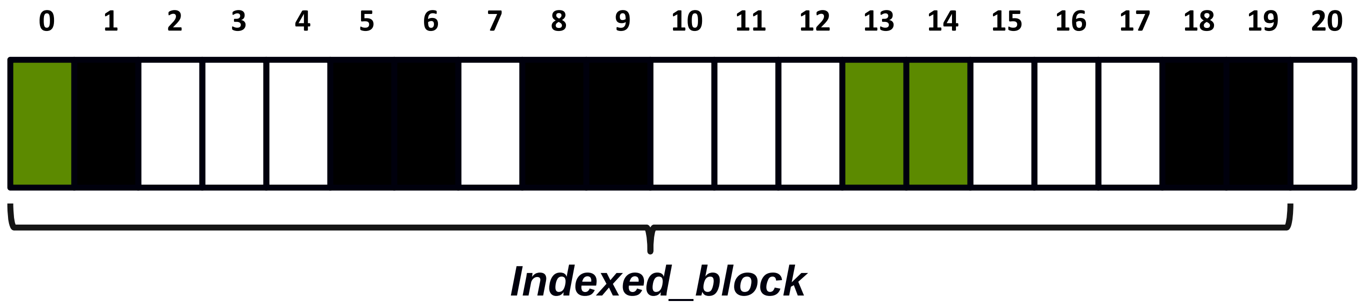
- Create **non-unit** strided vectors
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

MPI_Type_create_indexed_block(int count, int blocklength,
int *array_of_displacements, MPI_Datatype
oldtype, MPI_Datatype *newtype)

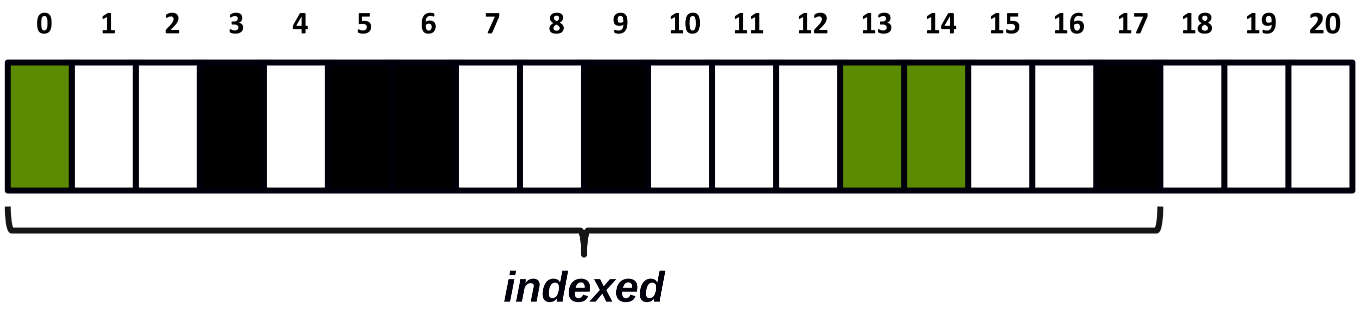
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype
oldtype, MPI_Datatype *newtype)

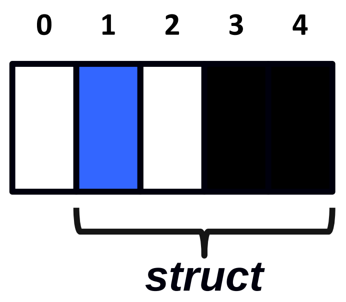
- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}



MPI_Type_create_struct

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],
                       MPI_Aint array_of_displacements[],
                       MPI_Datatype array_of_types[],
                       MPI_Datatype *newtype)
```

- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],
                         int array_of_subsizes[], int array_of_starts[],
                         int order, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

- Convenience function for creating datatypes for array segments
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)

(0,0)	**(0,1)**	**(0,2)**	**(0,3)**
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**
(3,0)	**(3,1)**	**(3,2)**	**(3,3)**

MPI_BOTTOM and MPI_Get_address

- **MPI_BOTTOM**
  - is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- **MPI_Get_address**
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use “&” operator in C!)
- Very important to
  - build struct datatypes
  - If data spans multiple arrays

Commit, Free, and Dup

- **MPI_Type_commit**
  - Types must be committed before use
  - Only the ones that are used!
  - may perform heavy optimizations (and will hopefully)
- **MPI_Type_free**
  - Free MPI resources of datatypes
  - Does not affect types built from it
- **MPI_Type_dup**
  - Duplicates a type
  - Library abstraction (composability)

Datatype Selection Order

- Simple and effective performance model:
  - More parameters == slower

predefined < contig < vector < index_block < index < struct

- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Try datatype "compression" bottom-up

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes

Code Example - 4

- *code/test4/*
  - mpi_dedrived_datatype_contiguous.c
  - mpi_dedrived_datatype_index.c
  - mpi_dedrived_datatype_struct.c
  - mpi_dedrived_datatype_vector.c

Content

- Non-blocking communication
- Group (collective) communication
- MPI Datatypes
- **Virtual Topology**

MPI Virtual Topologies

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".

- Virtual topologies are built upon MPI communicators and groups.
  - It is an attribute of processes only in the group.

MPI Virtual Topologies

- MPI topologies are virtual:
  - The term “Virtual Topology” gives this main idea: **machine independent**
  - no relation between the physical structure of the parallel machine and the process topology.

- Must be "programmed" by the application developer.

Why use Virtual Topologies

- **Convenience**:
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

Why use Virtual Topologies

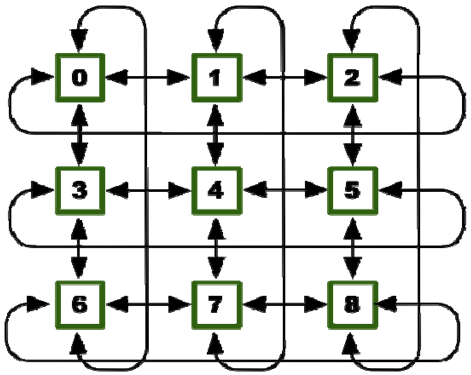
- **Communication Efficiency** :
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

MPI Topology History

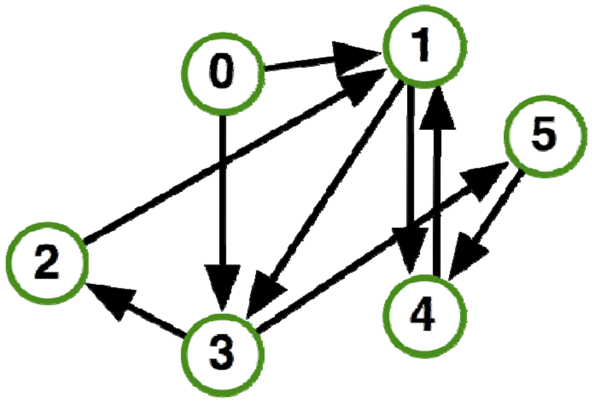
- Convenience functions (in MPI-1)
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph ☹
- Scalable Graph topology (MPI-2.2)
  - Graph topology: each rank specifies its neighbors **or** an arbitrary subset of the graph
- Neighborhood collectives (MPI-3.0)
  - Adding communication functions defined on graph topologies (neighborhood of distance one)

Types of Virtual Topologies

- There are two types of MPI topologies:



Cartesian Topology



Graph Topology

1 - Cartesian virtual topology

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

- Specify **ndims**-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product sum of dims must be <= P
- *Reorder* argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually

MPI_Cart_create Example

```
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Creates logical 3-d Torus of size 5x5x5
- But we're starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as "square" as possible, MPI can help!

MPI_Dims_create

MPI_Dims_create(int nnodes, int ndims, int *dims)

- Create dims array for Cart_create with *nnodes* and *ndims*
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes

MPI_Dims_create Example

```
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though

Cartesian Query Functions

- Library support and convenience!
  - MPI_Cartdim_get()
    - Gets dimensions of a Cartesian communicator
  - MPI_Cart_get()
    - Gets size of dimensions
  - MPI_Cart_rank()
    - Translate coordinates to rank
  - MPI_Cart_coords()
    - Translate rank to coordinates

Cartesian Communication Helpers

MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, …)
  - May return MPI_PROC_NULL
- Very convenient, all you need for nearest neighbor communication
  - No "over the edge" though

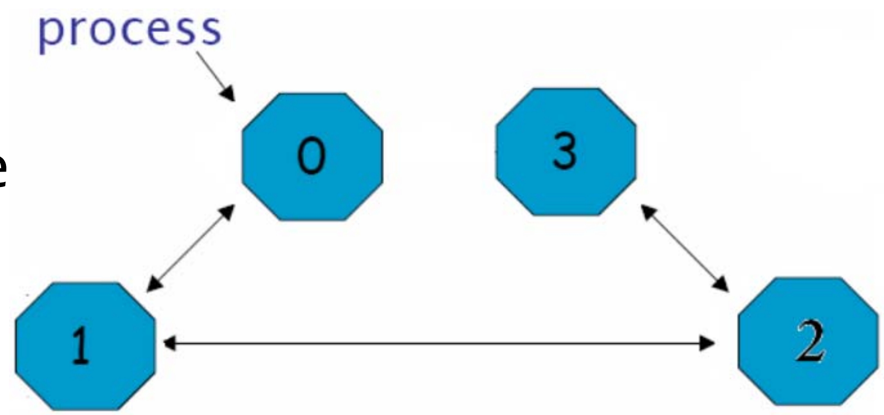
Cartesian Example

- Code/test5/
  - mpi_topology_cartesian.c
- A simplified mapping of processes into a Cartesian virtual topology (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)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

2 - Graph Topology

- More generally, the process organizing is described by a graph.
- Elements of Graph Topology:
  - Communication link
  - Nodes in the graph
  - Neighbours of per node
  - Type of mapping

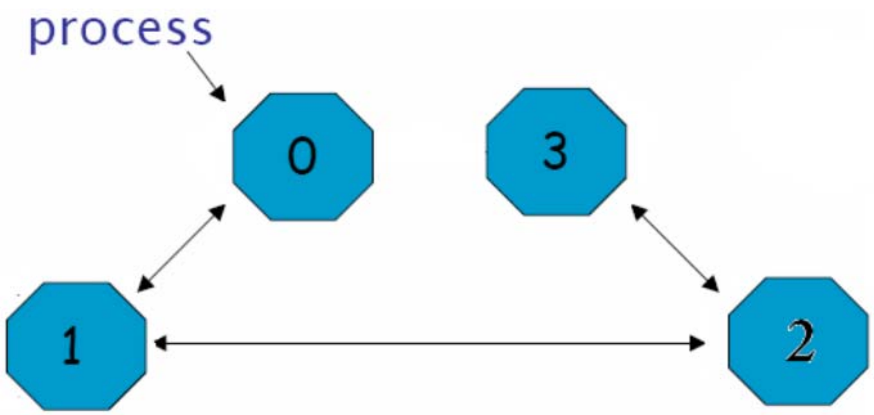


Graph Topology

- Elements of Graph Topology:

  - Nodes:
    - Processors
  - Lines:
    - Communicators between nodes
  - Arrows:
    - Show origins and destinations of links
  - Index:
    - array of integers describing node degrees

Node	Neighbors	Edges
0	1	1
1	2	0， 2
2	2	1， 3
3	1	2



Distributed Graph

- *MPI_Graph_create* **is discouraged**
  - Not scalable
  - Not deprecated yet but hopefully soon
- **New distributed interface**:
  - Scalable, allows distributed graph specification
    - Either local neighbors **or** any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights

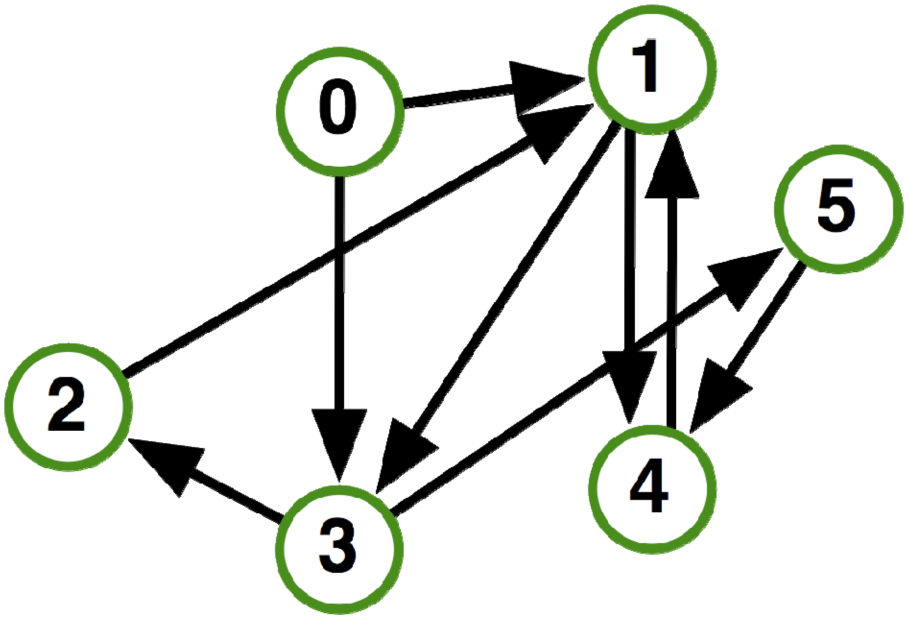
MPI_Dist_graph_create_adjacent

```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
            int indegree, const int sources[], const int sourceweights[],
            int outdegree, const int destinations[], const int destweights[],
            MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

- indegree, sources, ~weights – source proc. Spec.
- outdegree, destinations, ~weights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)

MPI_Dist_graph_create_adjacent

- Process 0:
  - Indegree: 0
  - Outdegree: 2
  - Dests: {3,1}
- Process 1:
  - Indegree: 3
  - Outdegree: 2
  - Sources: {4,0,2}
  - Dests: {3,4}
- …



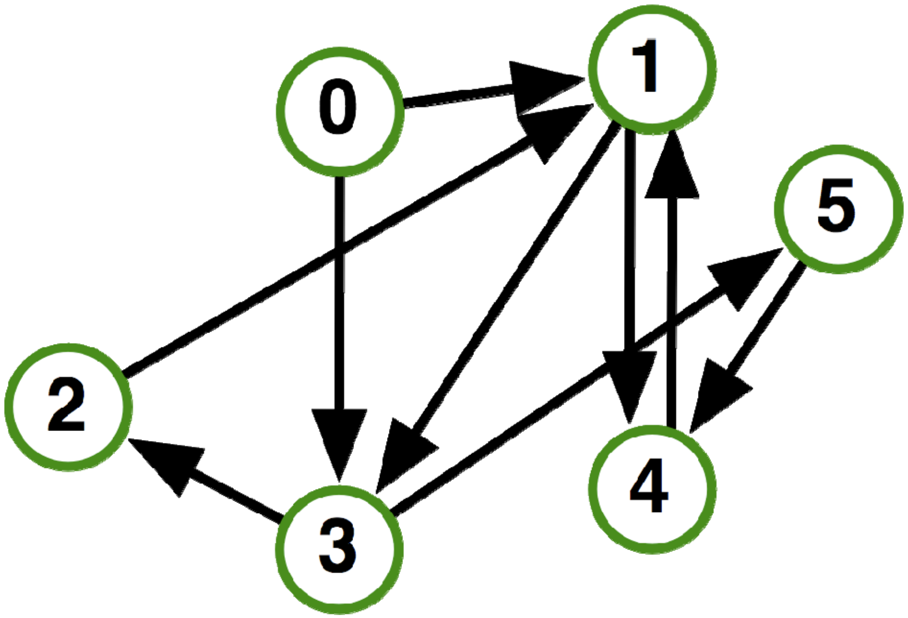
MPI_Dist_graph_create

```
MPI_Dist_graph_create(MPI_Comm comm_old,  int n,
            const int sources[], const int degrees[],
            const int destinations[], const int weights[],
            MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

MPI_Dist_graph_create

- Process 0:
  - N: 2
  - Sources: {0,1}
  - Degrees: {2,1}*
  - Dests: {3,1,4}
- Process 1:
  - N: 2
  - Sources: {2,3}
  - Degrees: {1,1}
  - Dests: {1,2}
- …



* Note that in this example, process 0 specifies only one of the two outgoing edges of process 1; the second outgoing edge needs to be specified by another process

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors_count(
            MPI_Comm comm,
            int *indegree, int *outdegree,  int *weighted)
```

- Query the number of neighbors of **calling process**
- Returns indegree and outdegree ;
- Also info if weighted

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors(MPI_Comm comm,
            int maxindegree, int sources[], int sourceweights[],
            int maxoutdegree, int destinations[], int destweights[])
```

- Query the neighbor list of **calling process**
- Optionally return weights

Further Graph Queries

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

- Status is either:
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)
- Enables to write libraries on top of MPI topologies!

Neighborhood Collectives

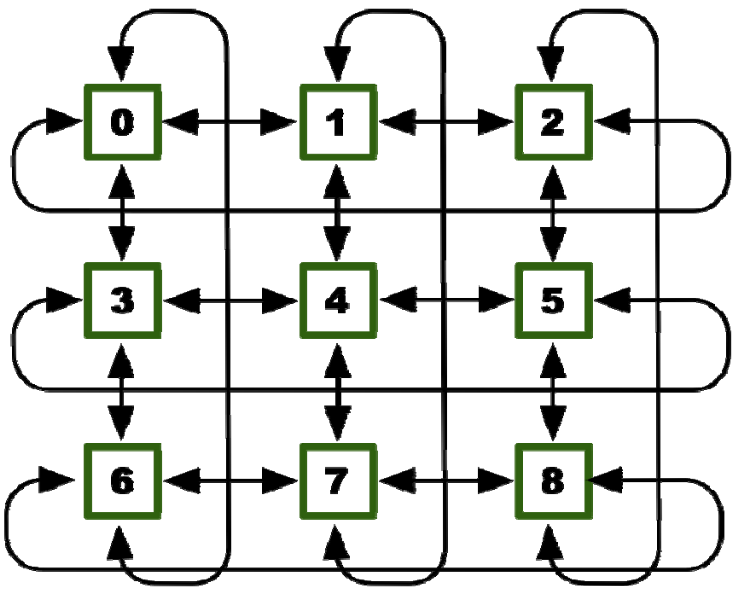
- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for "build your own collective" functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!

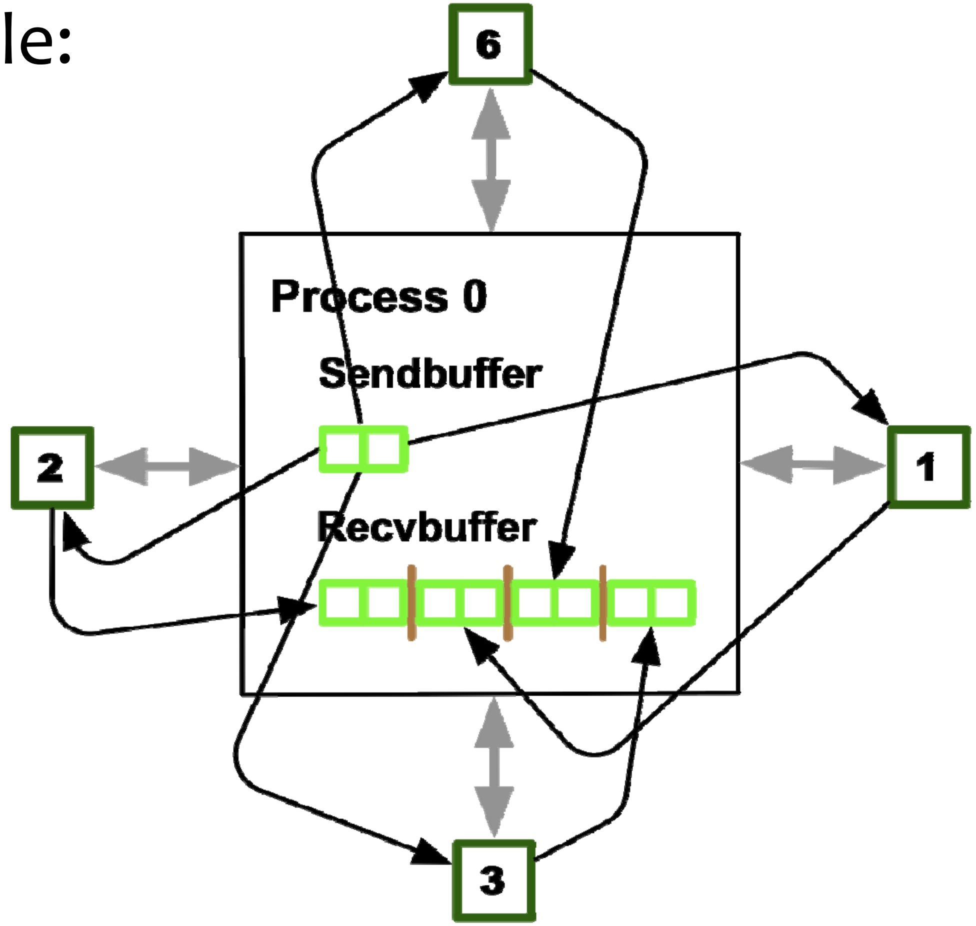
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive
    - 2*ndims sources and destinations
    - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

Cartesian Neighborhood Collectives

- Buffer ordering example:





Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations ☺
  - Any persistent communication pattern!

MPI_Neighbor_allgather

```
MPI_Neighbor_allgather(
        const void* sendbuf, int sendcount, MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Vector version for full flexibility

MPI_Neighbor_alltoall

```
MPI_Neighbor_alltoall(
        const void* sendbuf, int sendcount, MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```

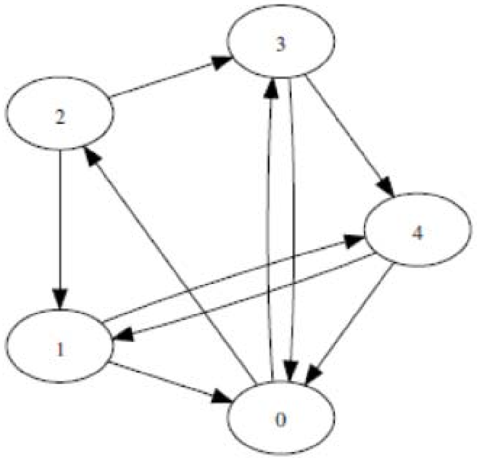
- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility

Nonblocking Neighborhood Collectives

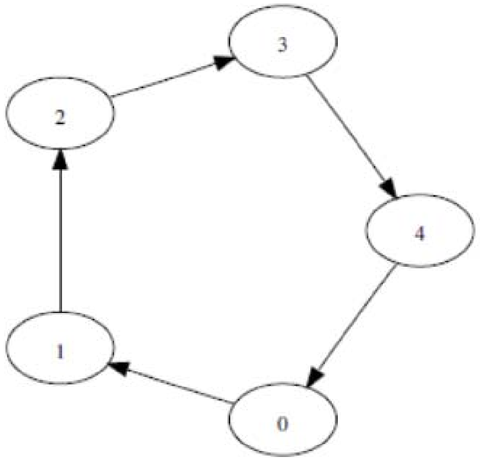
```
MPI_Ineighbor_allgather(…, MPI_Request *req);
MPI_Ineighbor_alltoall(…, MPI_Request *req);
```

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!

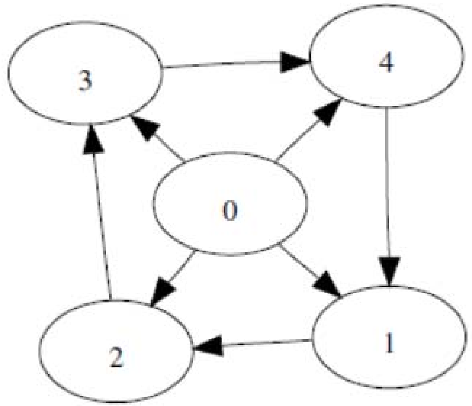
Examples of Graph Topology



(a) Random



(b) Ring



(c) Wheel

Example topologies

MPI_Dist_graph_create_adjacent：

Topology/Process	indegree	sources	outdegree	destinations
random/0	3	1,3,4	2	2,3
random/1	2	2,4	2	0,4
random/2	1	0	2	1,3
random/3	2	0,2	2	0,4
random/4	2	1,3	2	0,1
ring/i ($0 \leq i \leq 4$)	1	$i - 1 \bmod 5$	1	$i + 1 \bmod 5$
wheel/0	0	-	4	1,2,3,4
wheel/1	2	0,4	1	2
wheel/2	2	0,1	1	3
wheel/3	2	0,2	1	4
wheel/4	2	0,3	1	1

Example topologies

MPI_Dist_graph_create:

Topology/Process	n	sources	degrees	destinations
random/0	1	2	1	1
random/1	1	1	2	0,4
random/2	2	0,4	2,1	2,3,1
random/3	2	4,3	1,1	0,0
random/4	2	2,3	1,1	3,4
ring/i $(0 \leq i \leq 4)$	1	$i - 1 \bmod 5$	1	i
wheel/0	1	2	1	3
wheel/1	1	3	1	4
wheel/2	1	4	1	1
wheel/3	1	1	1	2
wheel/4	1	0	4	1,2,3,4

Graph Example

- Code/test5/
  - mpi_graph_topology_adjacent.cpp
  - mpi_graph_topology.cpp

- Reference:
  - The Scalable Process Topology Interface of MPI 2.2

references

- Parallel Programming with MPI, Argonne National Laboratory, http://www.anl.gov/events/parallel-programming-mpi.
- A Comprehensive MPI Tutorial Resource, https://github.com/wesleykendall/mpitutorial.
- 张武生等，MPI并行程序设计实例教程，清华大学出版社。
- 迟学斌等，并行计算与实现技术，科学出版社。