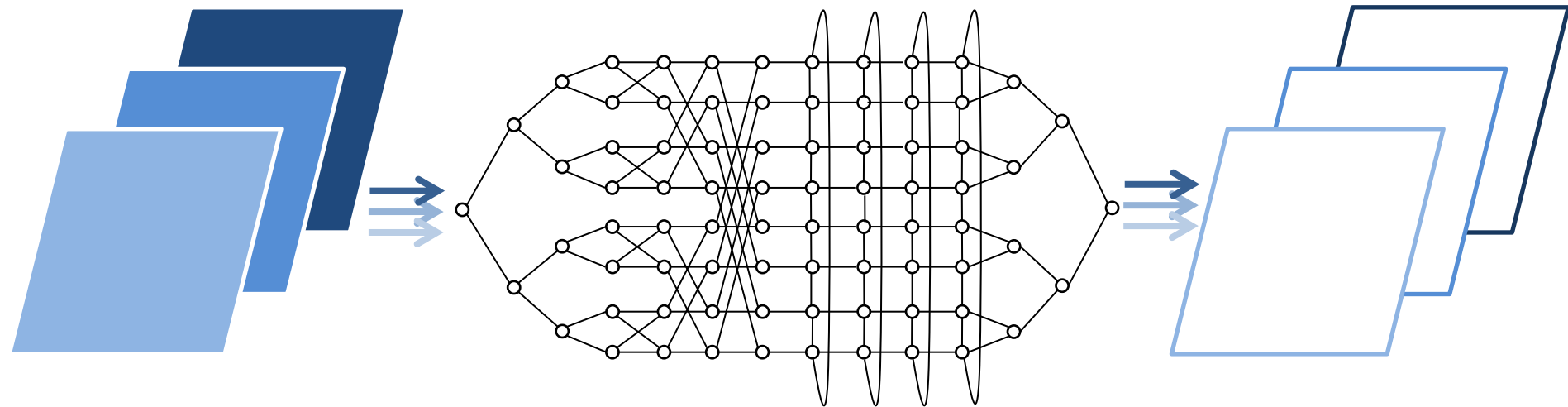


Programmation Parallèle

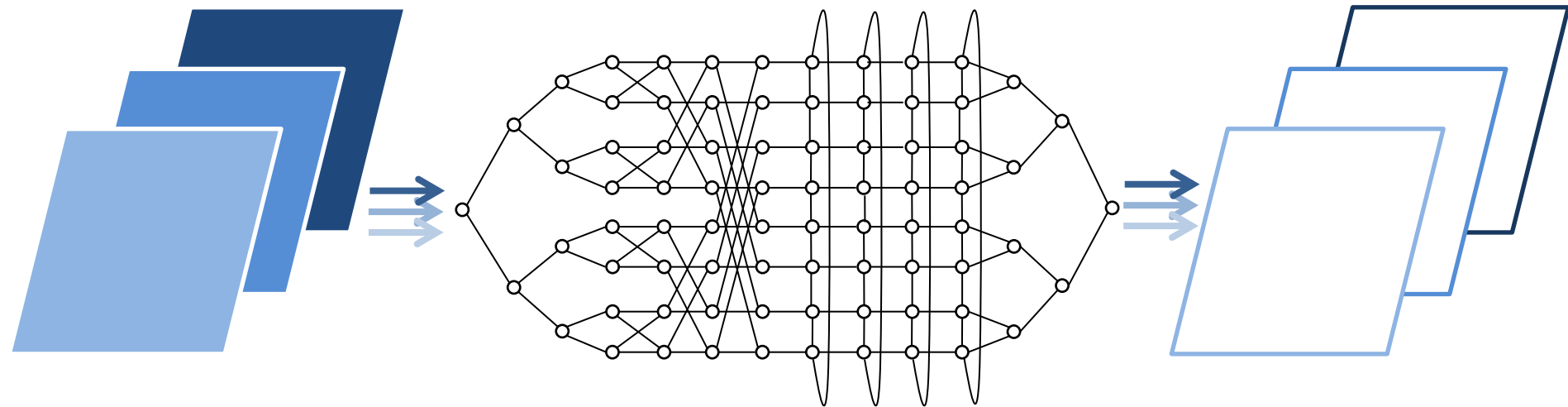
MPI: Message Passing Interface



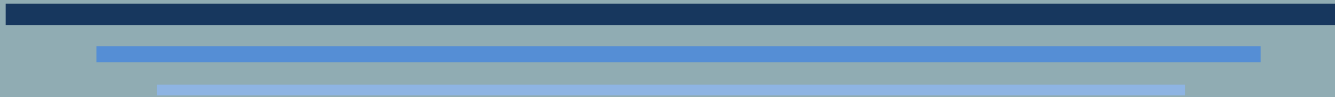
Advanced Collectives

Communications

ENSIIE-HPC/BigData-PP-IPAR-Lecture 4



MPI ASYNC. PROGRESSION



Reminder from course PP-C1



■ Definition

- A **non-blocking** communication has no guarantee when send function returns!

■ Meaning

- No safe access to input message when function send returns
- To be sure that message buffer can be reused, an additional function should be called and returned (*completion call*)

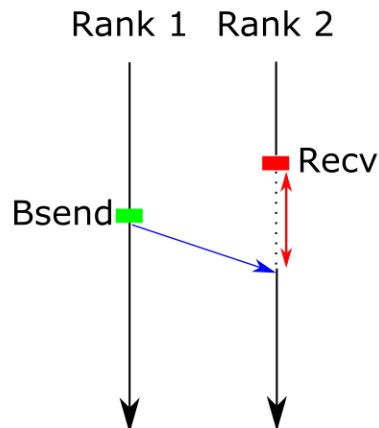
■ The function call only initiates the call

- The actual action (Send/Recv) will be realized at any time from the call
- A subsequent completion call ensures that the action is done after it is called
 - action is done just means the input buffer can be reused

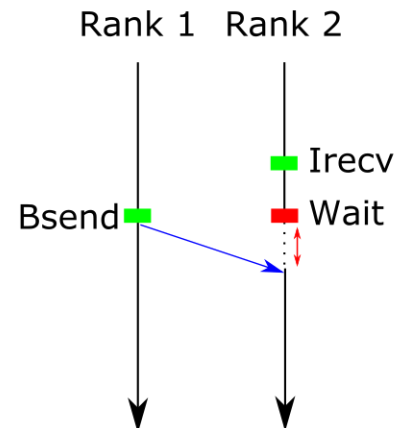
Asynchronous progression

- Allows to overlap communication with computation
 - Example: blocking send with
 - (a) blocking recv
 - (b) non-blocking recv partially overlapped
 - (c) non-blocking recv totally overlapped

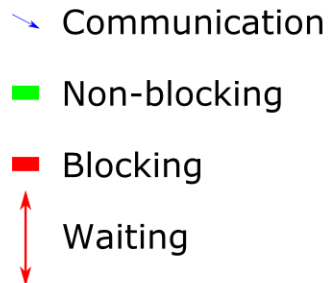
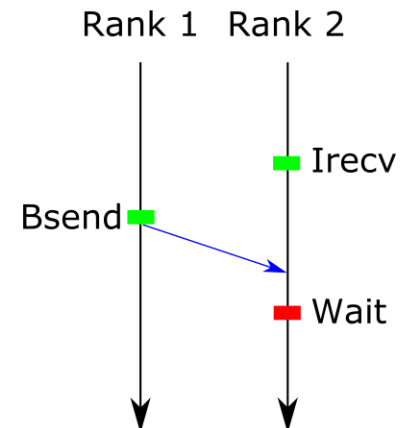
(a)



(b)



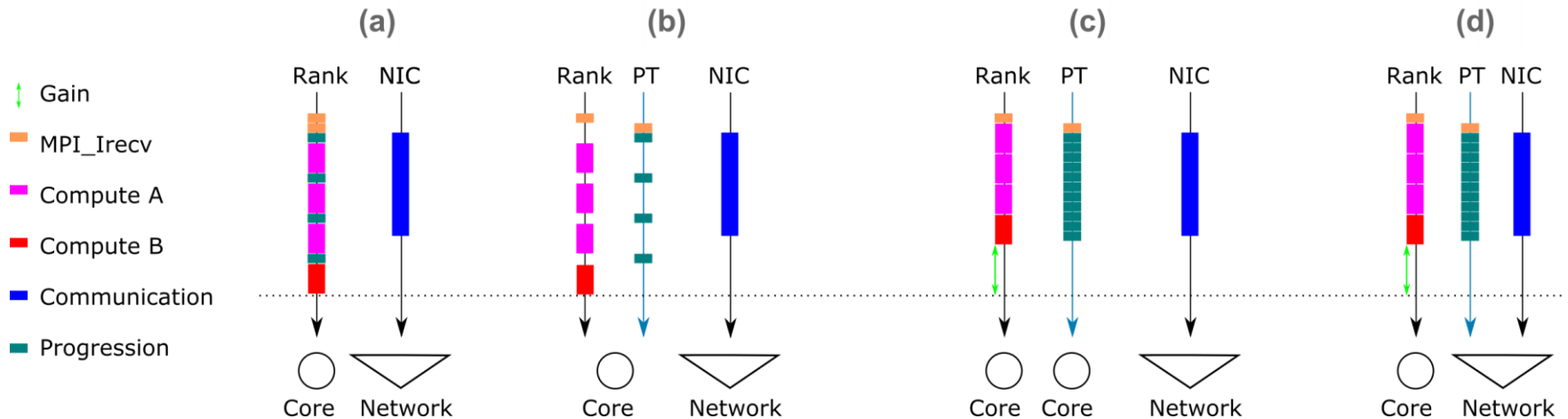
(c)



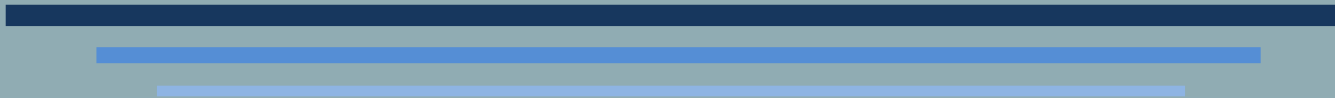
Asynchronous progression

■ How to progress messages?

- a) on the same thread, need to call functions to progress the message
- b) on another thread, only progression but competition on resources
- c) on another core, start and finish communications as soon as possible
- d) on the network card, if the functionality is available



MPI NON-BLOCKING COLLECTIVES



MPI Non-Blocking Collectives



- Provide all MPI collectives communications with non-blocking semantics
- Goal: get the best of collective operations...
 - Optimized communications patterns
- ... and non-blocking communications
 - Allows to overlap communication with computation
 - More communications involved in a collective call than p2p
 - => more opportunities to gain performances from overlapping
- Same semantic as Non-blocking point-to-point
 - Non-blocking communication calls initiate the communication
 - Operation is not finished upon return from communication call
 - Need a completion call to ensure input buffered can be reused
 - Wait* and Test* calls
 - Function name is the same but begins with "I"

Same rules as blocking collectives



- All processes in the group identified by the communicator must call the non-blocking collective routine
- Inside a communicator, processes should call the same sequence of collective communication (**blocking and non-blocking**)
- Between two ranks in different communicators, there are no restrictions

Same behavior as blocking collectives



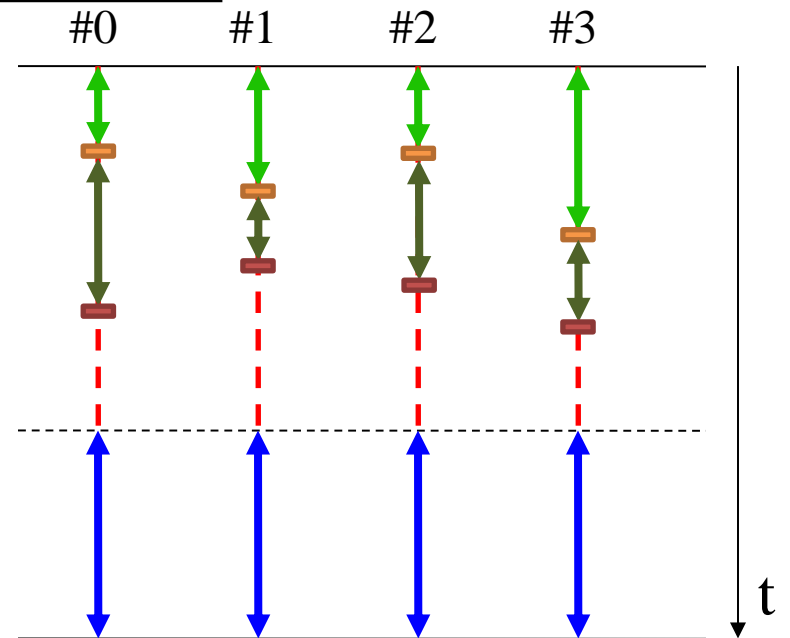
- Completing a non-blocking collective call (e.g. after the matching completion call) on a process does not mean that the collective is done for all processes
 - It just means that the contribution of the current process is done
- If the current process does not require the result but only send data, it basically is the same as `MPI_Isend`
 - After the completion call, the user can reuse the input buffer
 - It does not even mean that the data has been send
- Same function signature, adding an *MPI_Request* argument

Non-blocking Barrier



- Synchronize all processes belonging to target communicator

```
int MPI Barrier( MPI Comm comm,  
                MPI Request * req ) ;
```

```
MPI_Init(&argc, &argv);  
MPI_Request req;  
/* work 1 */  
MPI_Ibarrier(MPI_COMM_WORLD,  
             &req);  
/* work 2 */  
  
MPI_wait(req, status);  
  
/* work 2 */  
  
MPI_Finalize();
```





Non-blocking Broadcast



```
int MPI_Ibcast (  
    void *buf(inout),  
    int count(in),  
    MPI_Datatype datatype(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Scatter



```
int MPI_Iscatter (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sndtyp(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype rcvtyp(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Gather



```
int MPI_Igather (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sndtyp(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype rcvtyp(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Allgather



```
int MPI_Iallgather (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Alltoall



```
int MPI_Alltoall (  
    void *sendbuf(in),  
    int sendcount(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Reduction



```
int MPI_Ireduce (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Reduce to All



```
int MPI_Iallreduce (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Partial Reduction



```
int MPI_Iscan (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.

Non-blocking MPI_Scatterv



```
int MPI_Iscatterv (  
    void *sendbuf(in),  
    int *sendcounts(in),  
    int *displs(in),  
    MPI_Datatype sendtype(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype recvtype(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.

Non-blocking MPI_Alltoallw

```
int MPI_Ialltoallw (  
    void *sendbuf(in),  
    int *sendcounts(in),  
    int *displs(in),  
    MPI_Datatype *sendtypes(in),  
    void *recvbuf(out),  
    int *recvcounts(in),  
    MPI_Datatype *recvtypes(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.



Non-blocking Reduction-Scatter block



```
int MPI_Ireduce_Scatter_block (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int recvcount(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

Request **req** is update with non-blocking calls information to give to completion call.

Non-blocking Partial Reduction



```
int MPI_Iexscan (  
    void *sendbuf(in),  
    void *recvbuf(out),  
    int count(in),  
    MPI_Datatype datatype(in),  
    MPI_Op op(in),  
    int root(in),  
    MPI_Comm comm(in),  
    MPI_Request *req(inout)  
);
```

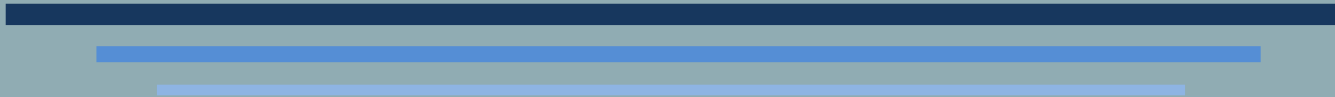
Request **req** is update with non-blocking calls information to give to completion call.

MPI Non-Blocking Collectives



- To obtain performance with NBC
 - Start communication as soon as possible
 - Insert completion call as late as possible
 - Maximize the overlap potential between communication and compute
- Depends on NBC implementation
 - If no progression thread, difficult to overlap
 - Still possible with a NIC handling message passing
 - With progression thread
 - Where? All cores may be used for MPI, or other models (threads...)
 - On the same node: may slow down computation
 - On another node: may slowdown another MPI process, or another thread
 - Allocate specific resources for progression thread: is it worth it?
 - Less communications but larger communication buffers and larger compute phase per MPI process

MPI TOPOLOGY



MPI Topology



- MPI processes are arbitrarily distributed on resources, with no regards to the communications patterns in play
- Simply because the MPI runtime does not have any clue of the communications that will happen on a communicator
- The user is responsible for the communication patterns, which reflect the data repartition across the MPI processes

MPI Topology

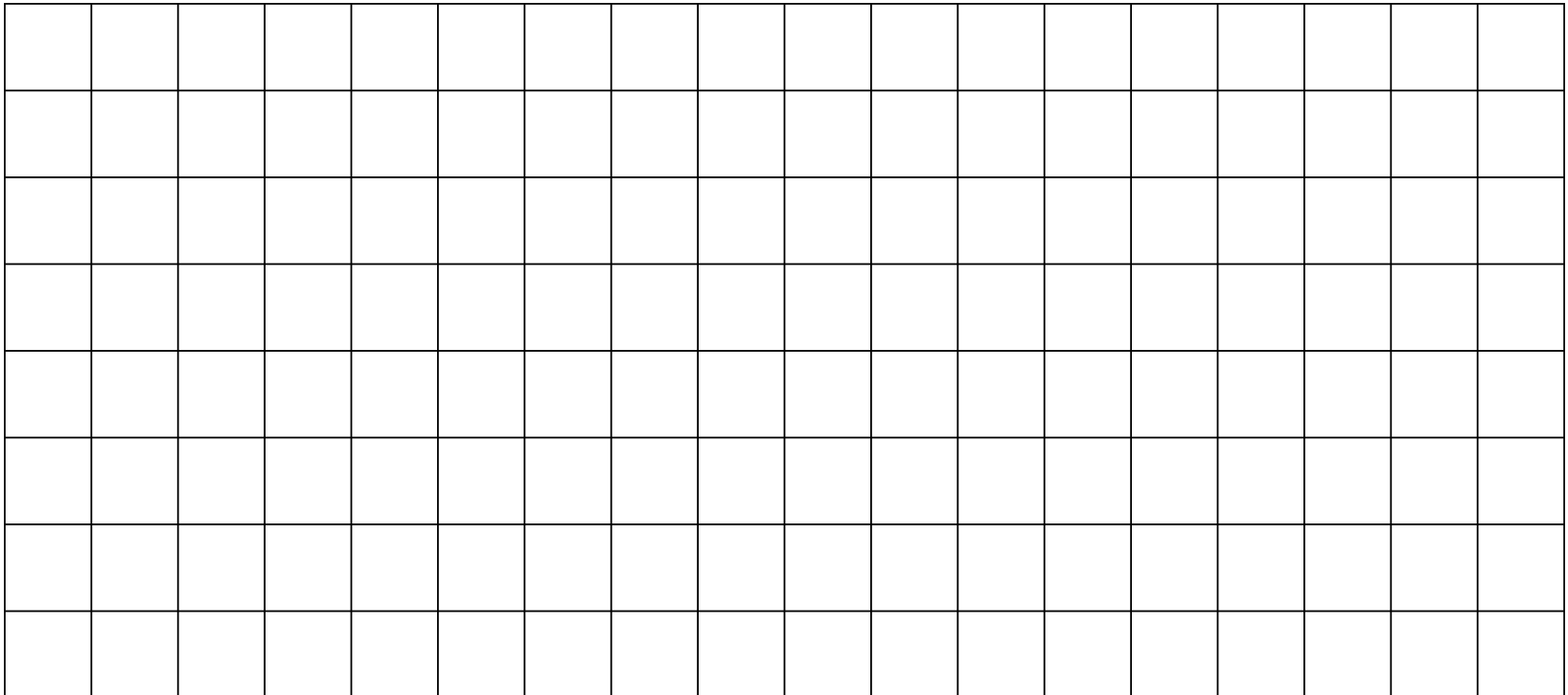


- Because it is easier for humans, data repartition often follows constructed patterns
- MPI Topology routines allow to virtually organize MPI processes according to a given topology
- Two types of topology necessary: Multi-dimensional grids or graphs

MPI Grid Topology



- Consider a multi-dimensional grid
 - 2D in the example

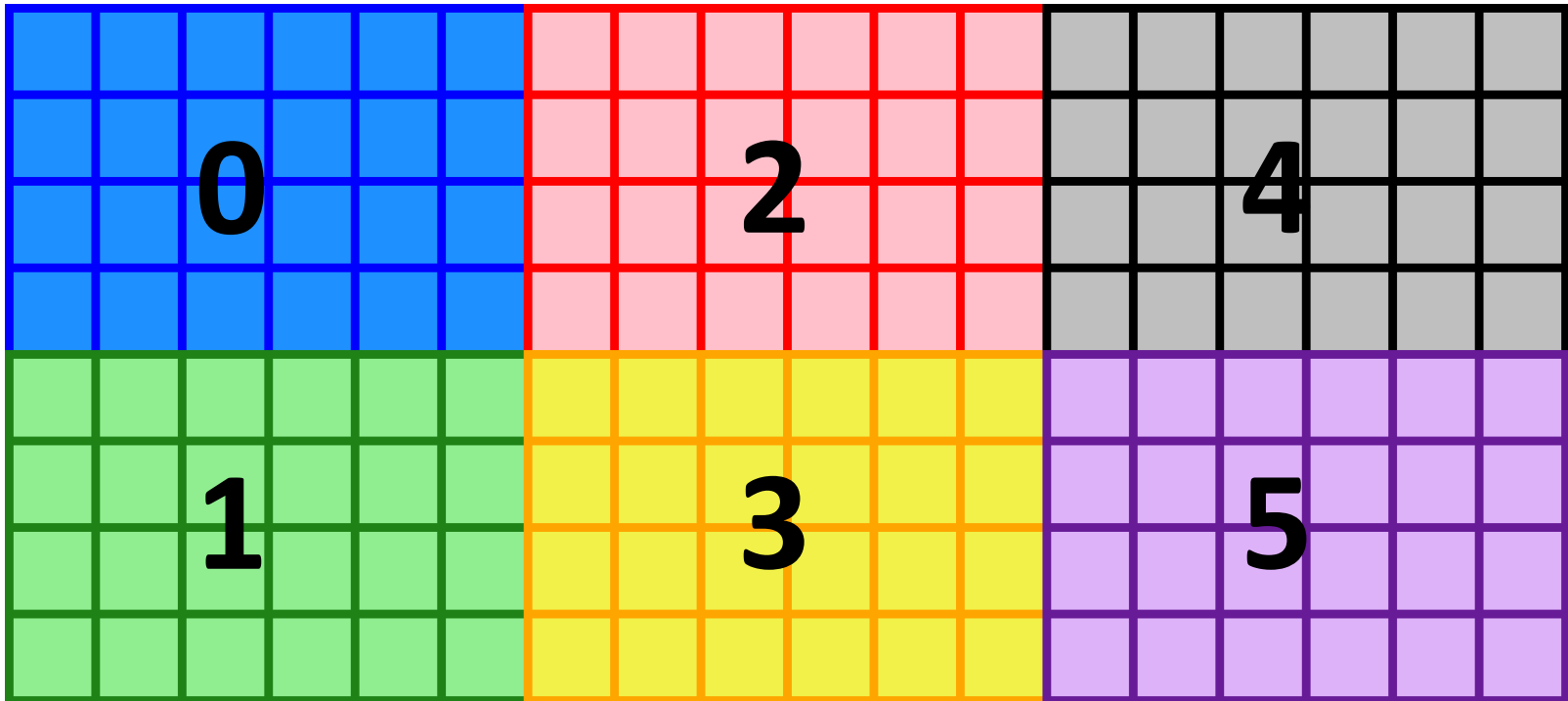




- [illegible]

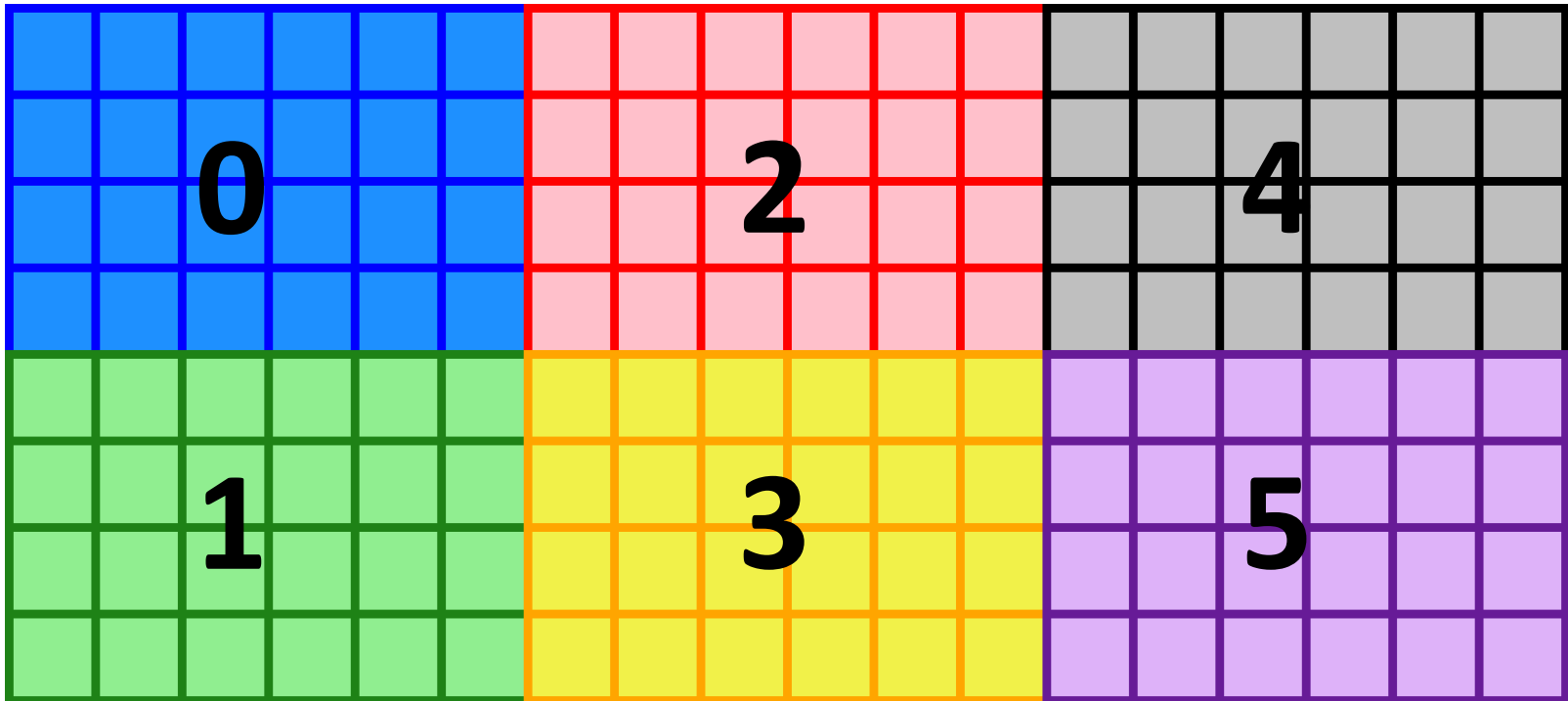
MPI Grid Topology

- Homogeneous decomposition
 - $6 \times 4 = 24$ cells per MPI rank



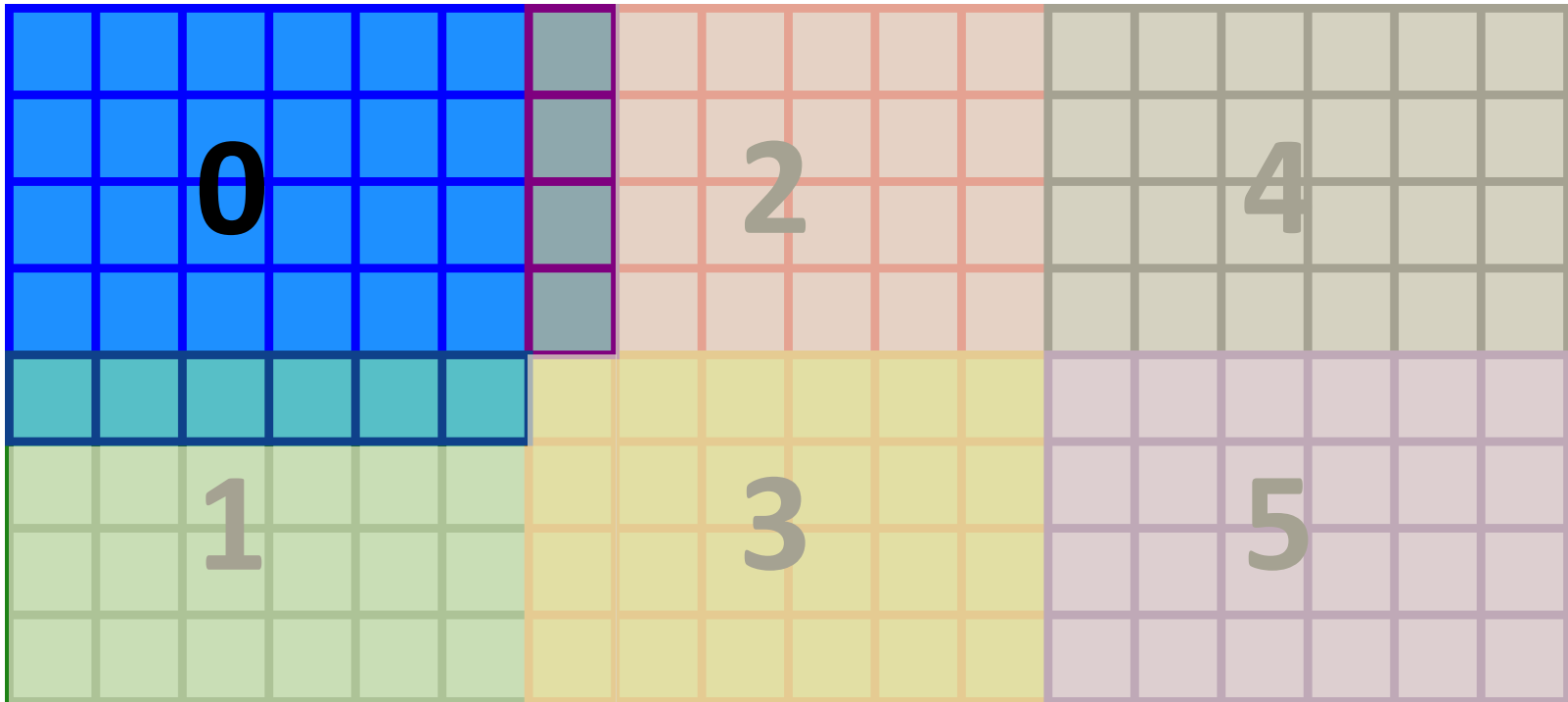
MPI Grid Topology

- Stencil code: updating a cell requires contribution from direct neighbors



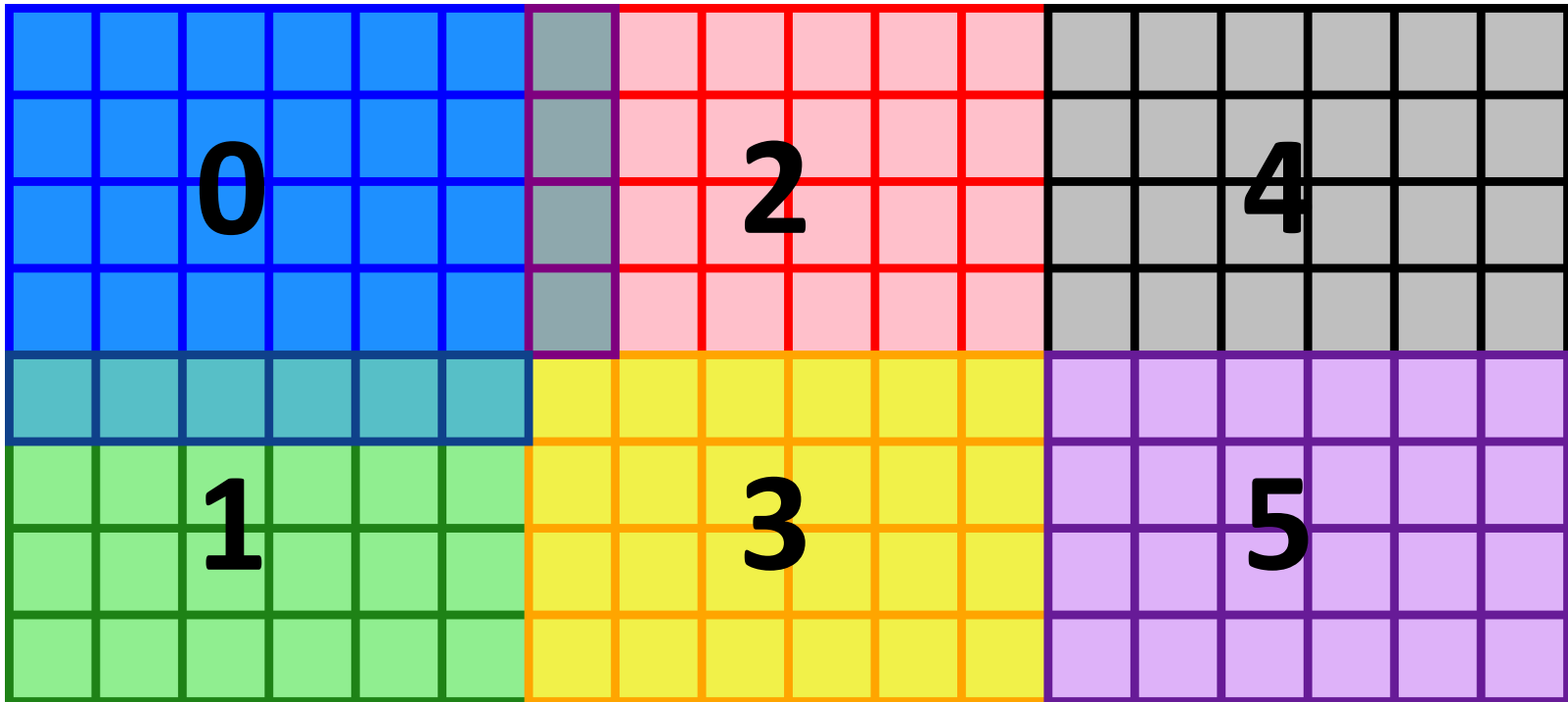
MPI Grid Topology

- Stencil code: updating a cell requires contribution from direct neighbors
 - Highlighted cell for MPI rank 0



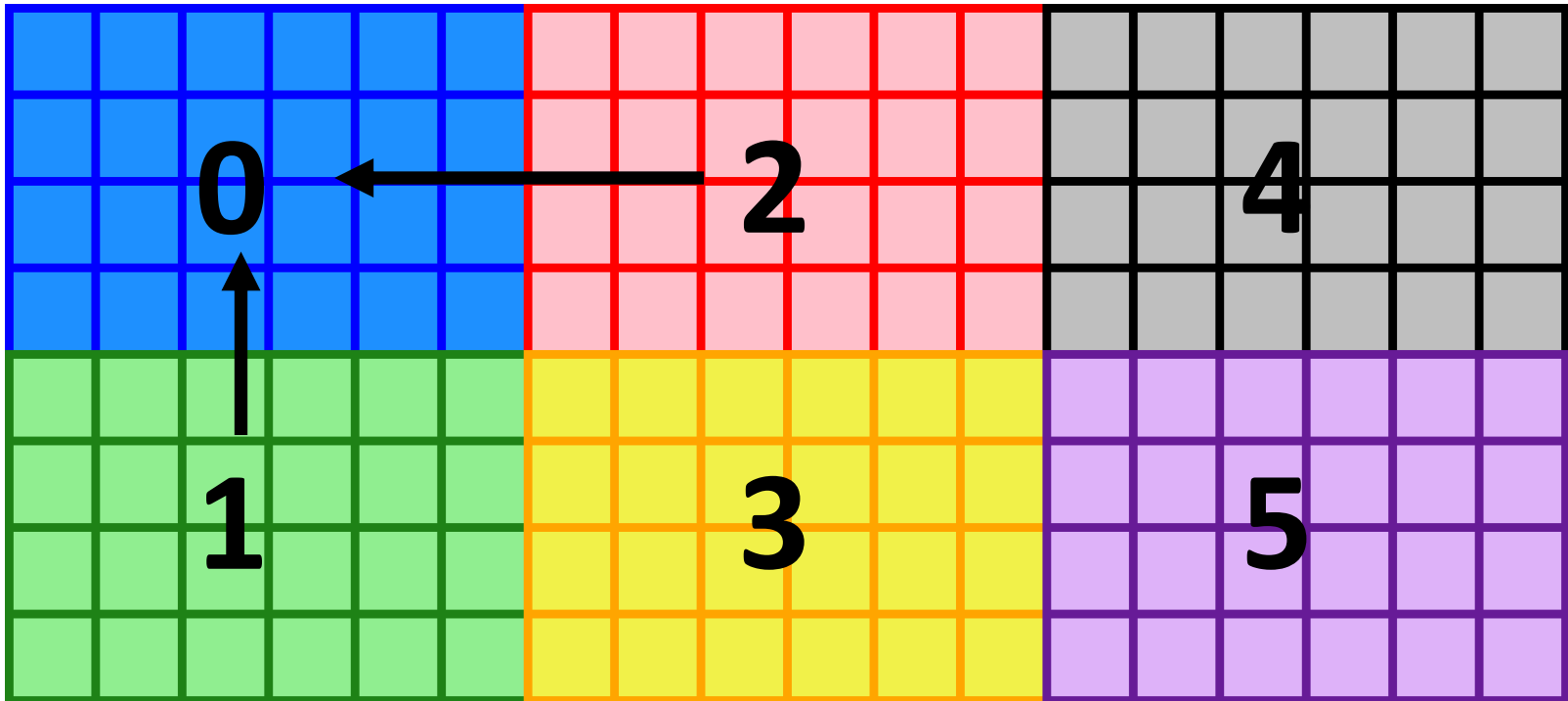
MPI Grid Topology

- Stencil code: updating a cell requires contribution from direct neighbors
 - Highlighted cell for MPI rank 0



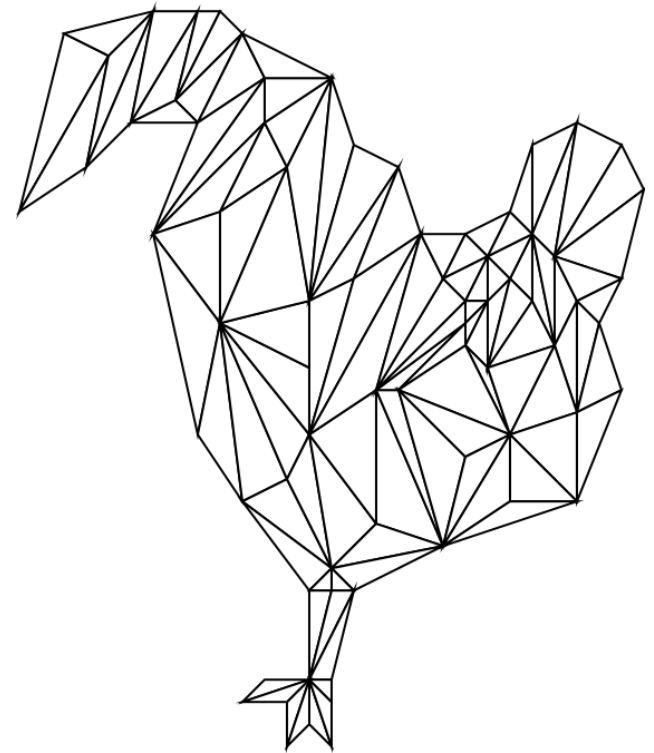
MPI Grid Topology

- A rank needs to exchange data only with its neighbors
 - Use a ***cart*** topology to fix rank neighbors in MPI runtime



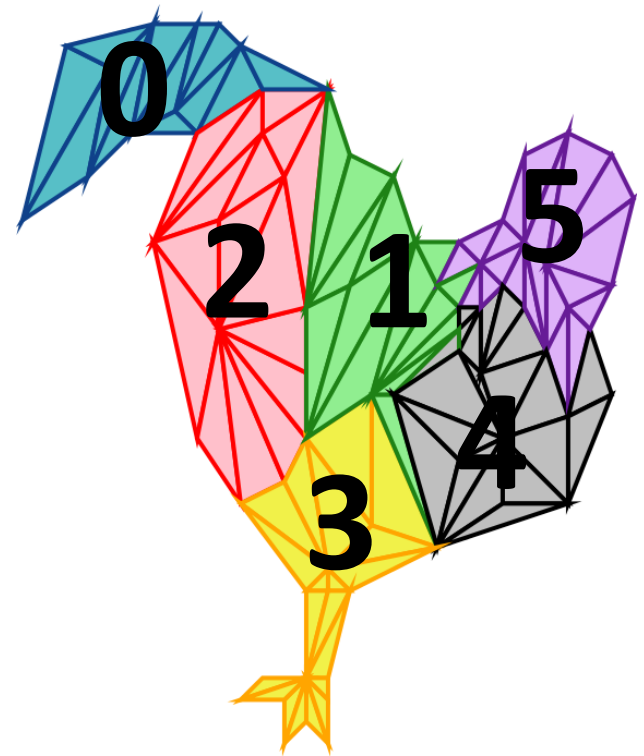
MPI Graph Topology

- Consider a non-uniform mesh representing an irregular form
 - 2D in the example
- Usually, call an automatic practitioner to decompose data across MPI ranks
 - With different metrics: number of cells/particles/comms, amount of comms...



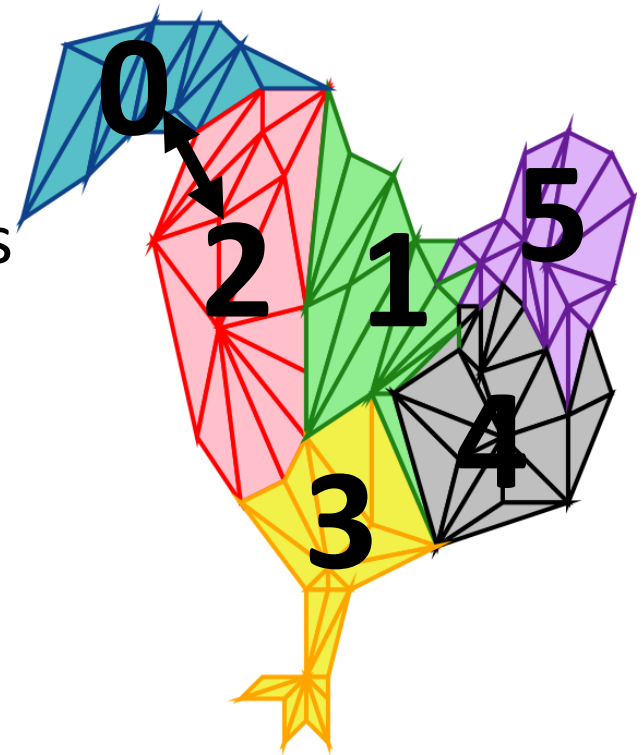
MPI Graph Topology

- Decomposing this mesh on 6 MPI ranks



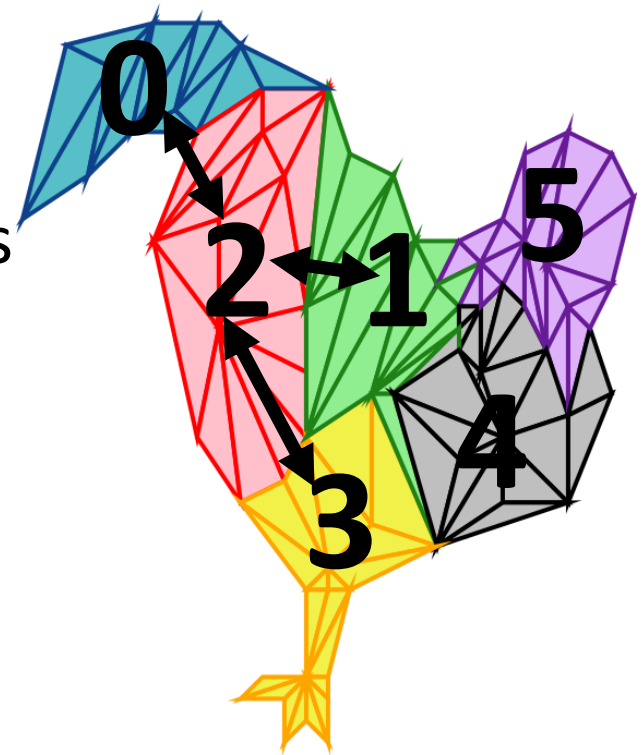
MPI Graph Topology

- Decomposing this mesh on 6 MPI ranks
- Particles movement: A rank needs to exchange data only with its neighbors
 - $0 \leftrightarrow 2$



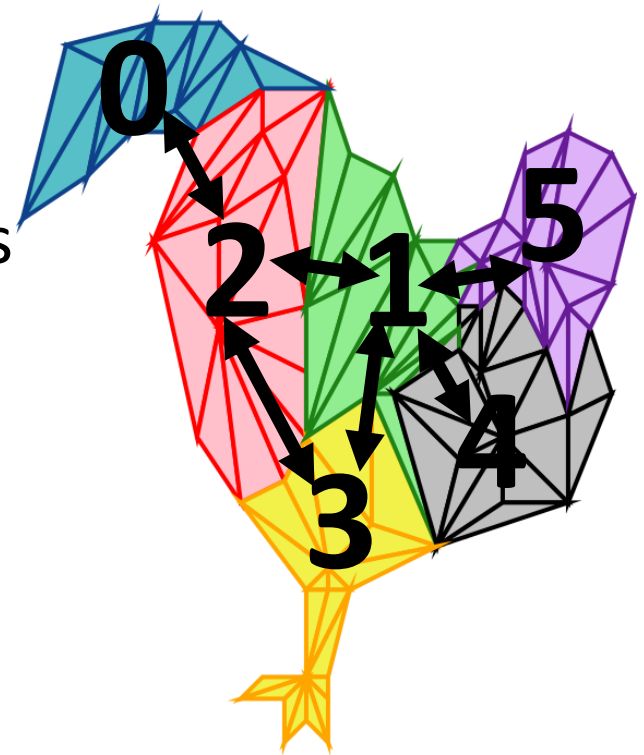
MPI Graph Topology

- Decomposing this mesh on 6 MPI ranks
- Particles movement: A rank needs to exchange data only with its neighbors
 - $0 \leftrightarrow 2$
 - $2 \leftrightarrow 1$; $2 \leftrightarrow 3$



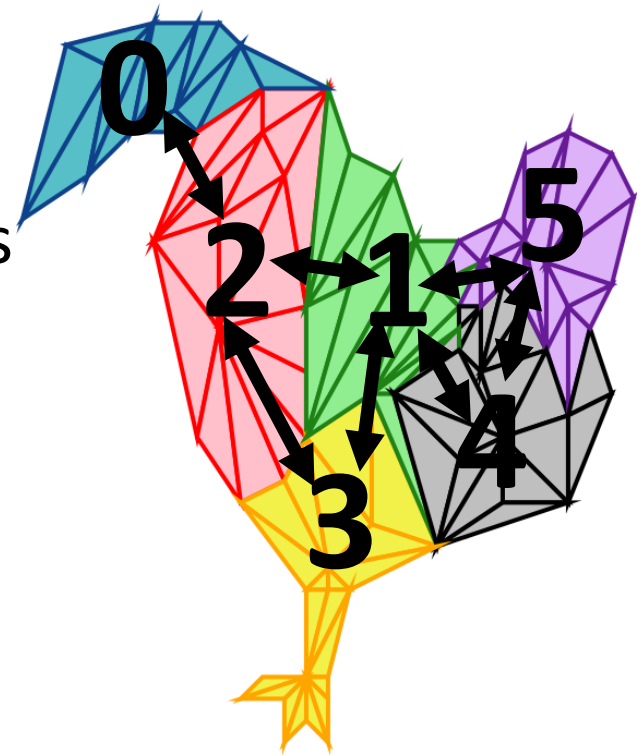
MPI Graph Topology

- Decomposing this mesh on 6 MPI ranks
- Particles movement: A rank needs to exchange data only with its neighbors
 - $0 \leftrightarrow 2$
 - $2 \leftrightarrow 1$; $2 \leftrightarrow 3$
 - $1 \leftrightarrow 3$; $1 \leftrightarrow 4$; $1 \leftrightarrow 5$



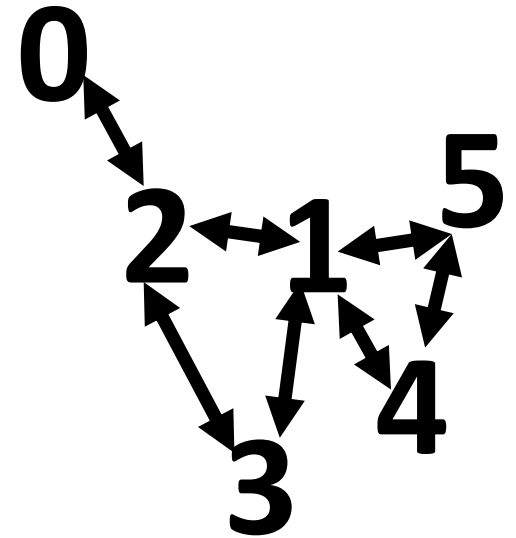
MPI Graph Topology

- Decomposing this mesh on 6 MPI ranks
- Particles movement: A rank needs to exchange data only with its neighbors
 - $0 \leftrightarrow 2$
 - $2 \leftrightarrow 1$; $2 \leftrightarrow 3$
 - $1 \leftrightarrow 3$; $1 \leftrightarrow 4$; $1 \leftrightarrow 5$
 - $4 \leftrightarrow 5$;



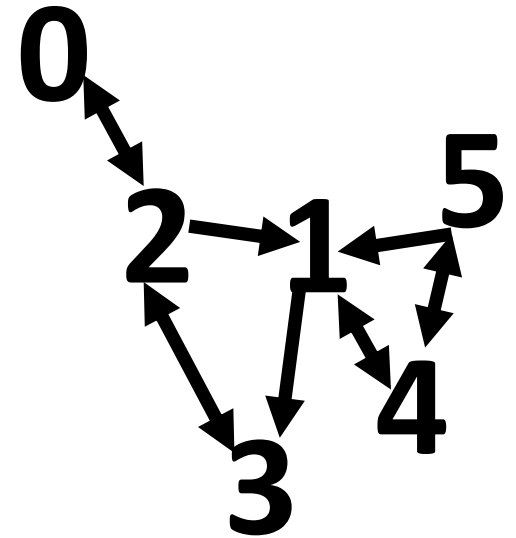
MPI Graph Topology

- You get a graph of communications between MPI ranks
- Using a **graph** topology allow to specify this graph to the MPI runtime
- Collectives communications will only involve neighbors according to the topology



MPI Graph Topology

- Need to specify neighbors per MPI ranks
 - Possible to have unidirectional vertices in the graph
- Graph creation exists in two forms:
 - Each MPI Rank must specify the complete topology
 - Not scalable
 - Each MPI Rank only specifies its own neighbors





Cartesian Topology creation



- `MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart);`
 - IN `comm_old` (input communicator)
 - IN `ndims` (number of dimensions of Cartesian grid)
 - IN `dims` (integer array of size `ndims` specifying the number of processes in each dimension)
 - IN `periods` (logical array of size `ndims` specifying whether the grid is periodic *-true-* or not *-false-* in each dim.)
 - IN `reorder` (ranking may be logically reordered *-true-* or not *-false-*)
 - OUT `comm_cart` (communicator with new Cartesian topology)

MPI_Cart_create example 1



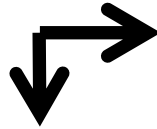
```
int size_array=2;  
int dims[2] = {3 , 2};  
int periods[2] = {false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

■ Size_array = 2 => 2D array ↘

MPI_Cart_create example 1

```
int size_array=2;  
int dims[2] = {3 , 2};  
int periods[2] = {false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

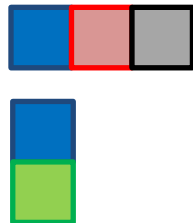
- Size_array = 2 => 2D array
- dims[0] = 3



MPI_Cart_create example 1

```
int size_array=2;  
int dims[2] = {3 , 2};  
int periods[2] = {false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

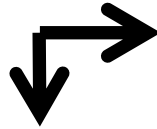
- Size_array = 2 => 2D array
- dims[0] = 3
- dims[1] = 2



MPI_Cart_create example 1

```
int size_array=2;  
int dims[2] = {3 , 2};  
int periods[2] = {false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

- Size_array = 2 => 2D array
- dims[0] = 3
- dims[1] = 2
- topology



MPI_Cart_create example 1

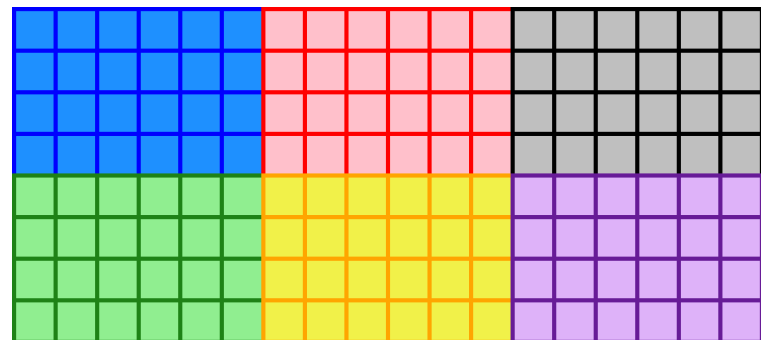
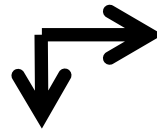
```
int size_array=2;  
int dims[2] = {3 , 2};  
int periods[2] = {false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

■ Size_array = 2 => 2D array

■ dims[0] = 3

■ dims[1] = 2

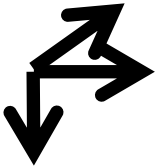
■ topology



MPI_Cart_create example 2



```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

■ Size_array = 3 => 3D array 

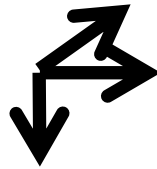
MPI_Cart_create example 2

```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

- Size_array = 3 => 3D array

- dims[0] = 3


--	--	--

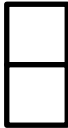


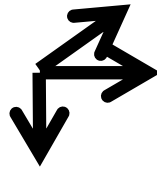
MPI_Cart_create example 2

```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

■ Size_array = 3 => 3D array

■ dims[0] = 3 

■ dims[1] = 2 



MPI_Cart_create example 2

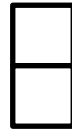
```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

■ Size_array = 3 => 3D array

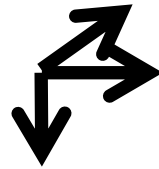
■ dims[0] = 3



■ dims[1] = 2



■ dims[3] = 4



MPI_Cart_create example 2

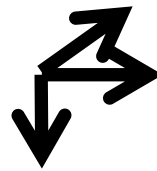
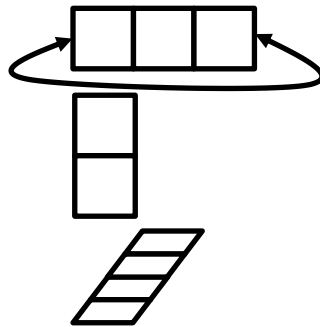
```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

- Size_array = 3 => 3D array

- dims[0] = 3

- dims[1] = 2

- dims[3] = 4



periods[1] = true

MPI_Cart_create example 2

```
int size_array=3;  
int dims[3] = {3 , 2, 4};  
int periods[2] = {true , false, false};  
MPI_Cart_create(comm_old, size_array, dims, periods, false, &comm_cart);
```

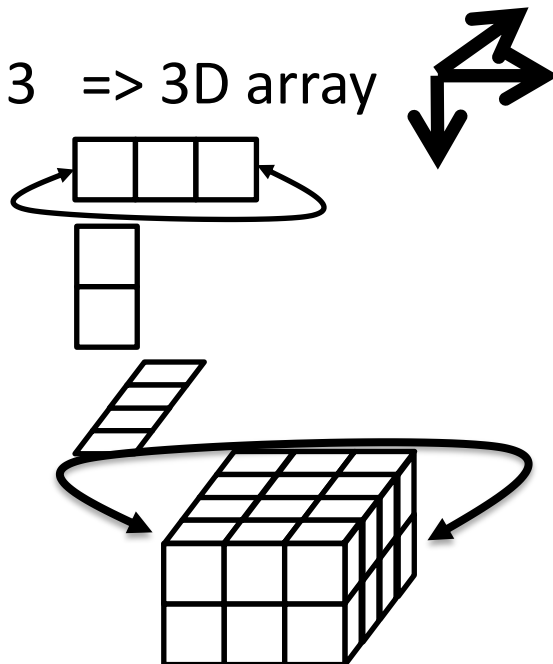
- Size_array = 3 => 3D array

- dims[0] = 3

- dims[1] = 2

- dims[2] = 4

- topology:



periods[1] = true

Get MPI rank coordinate

- `MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]);`
 - IN `comm` (communicator with Cartesian structure)
 - IN `rank` (rank of a process within group of `comm`)
 - IN `maxdims` (length of `coords` - *ndims* - in the calling program)
 - OUT `coords` (integer array (of size *ndims*) containing the Cartesian coordinates of specified rank)
- Returns, in array *coords*, the coordinates of the specified *rank* in the cartesian topology associated with communicator *comm*.

Get MPI rank from coordinates



- `MPI_Cart_rank`(MPI_Comm comm, const int coords[], int *rank);
 - IN comm (communicator with Cartesian structure)
 - IN coords (integer array (of size *ndims*) specifying the Cartesian coordinates of a process)
 - OUT rank (rank of specified process)
- Returns, in *rank*, the rank located at the coordinates *coords* in the cartesian topology associated with communicator *comm*.

Get Cartesian Topology dimensions





- `MPI_Cartdim_get(MPI_Comm comm, int *ndims);`
 - IN `comm` (communicator with Cartesian structure)
 - OUT `ndims` (number of dimensions of the Cartesian structure)
- Returns, in *ndims*, the number of dimensions in the cartesian topology associated with communicator *comm*.

Get Cartesian Topology infos



- `MPI_Cart_get`(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[]);
 - IN comm (communicator with Cartesian structure)
 - IN maxdims (length of vectors dims, periods, and coords)
 - OUT dims (number of processes for each Cartesian dimension)
 - OUT periods (periodicity *-true/false-* for each Cartesian dimension)
 - OUT coords (coordinates of calling process in Cartesian structure)
- Returns the number of dimension, and the periodicity of these dimensions, of the cartesian topology associated with communicator *comm*. Also returns the coordinates of the calling *rank* in the cartesian topology

Automatically create topo dimensions

- 
- 
- `MPI_DIMS_CREATE(nnodes, ndims, dims);`
 - IN `nnodes` (number of nodes in a grid)
 - IN `ndims` (number of Cartesian dimensions)
 - INOUT `dims` (integer array of size `ndims` specifying the number of nodes in each dimension)
 - This function will compute the size of each dimensions according to
 - The number of MPI ranks for the cartesian topology
 - The number of dimensions in the future topology
 - A first partial repartition specified by the user
 - If `dims[i]` is not 0 on input, the value is kept
 - If `dims[i]` is 0 on input, the function replace it with the computed value for dimension `i`
 - The dimensions are as close to each other as possible

Automatically create topo dimensions

- Examples from the MPI API document
 - Chapter 7.5, page 293

DIMS BEFORE CALL	FUNCTION CALL	DIMS ON RETURN
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

Automatically create topo dimensions

- Examples from the MPI API document
 - Chapter 7.5, page 293

DIMS BEFORE CALL	FUNCTION CALL	DIMS ON RETURN
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- 6 ranks, 2 dimensions, no predefined sizes
 - $6 = 6 \times 1 = \mathbf{3 \times 2} = 2 \times 3 = 1 \times 6$

Automatically create topo dimensions

- Examples from the MPI API document
 - Chapter 7.5, page 293

DIMS BEFORE CALL	FUNCTION CALL	DIMS ON RETURN
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- 7 ranks, 2 dimensions, no predefined sizes
 - $7 = \mathbf{7 \times 1} = 1 \times 7$

Automatically create topo dimensions

- Examples from the MPI API document
 - Chapter 7.5, page 293

DIMS BEFORE CALL	FUNCTION CALL	DIMS ON RETURN
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- 6 ranks, 3 dimensions, 1 predefined dimension
 - $6 = \mathbf{2 \times 3 \times 1} = 1 \times \underline{3} \times 2 = \cancel{3 \times 2 \times 1} = \dots = \cancel{6 \times 1 \times 1} = \cancel{1 \times 6 \times 1} = \dots$

Automatically create topo dimensions

- Examples from the MPI API document
 - Chapter 7.5, page 293

DIMS BEFORE CALL	FUNCTION CALL	DIMS ON RETURN
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- 7 ranks, 3 dimensions, 1 predefined dimension
 - $7 = \cancel{7 \times 1 \times 1} = \cancel{1 \times 7 \times 1} = \cancel{1 \times 1 \times 7}$



Graph Topology creation



- `MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph);`
 - IN `comm_old` (input communicator)
 - IN `nnodes` (number of nodes in graph)
 - IN `index` (array of integers describing node degrees)
 - IN `edges` (array of integers describing graph edges)
 - IN `reorder` (ranking may be reordered *-true-* or *-false-*)
 - OUT `comm_graph` (communicator with graph topology added)
- Index: cumulative number of neighbors
 - The *i*-th entry of array *index* stores the total number of neighbors of the first *i* graph nodes.
- Edges: linearized array of edges indexes of all nodes (ordered)

MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- nnodes = 2

0 2
1 3

MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

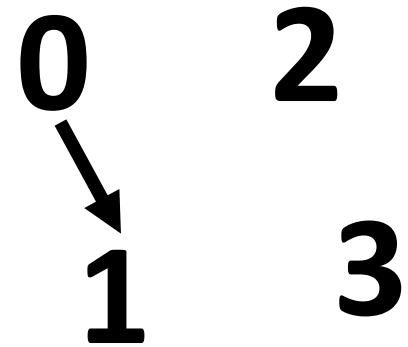
```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- nnodes = 2
- index[0] = 2
 - edge[0]=1



MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

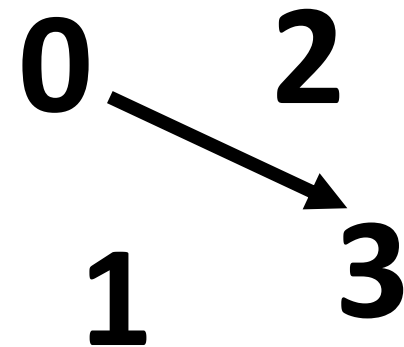
```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- `nnodes = 2`
- `index[0] = 2`
 - `edge[0]=1`
 - `edge[1]=3`



MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

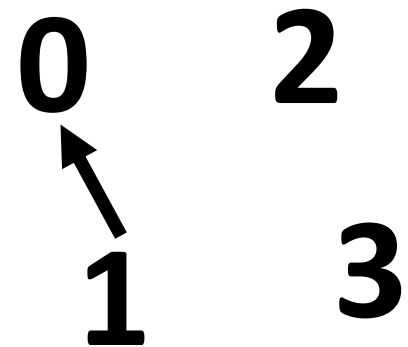
```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- nnodes = 2
- index[0] = 2
 - edge[0]=1
 - edge[1]=3
- Index[1] = 1
 - edge[2] = 0



MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- nnodes = 2

- index[0] = 2

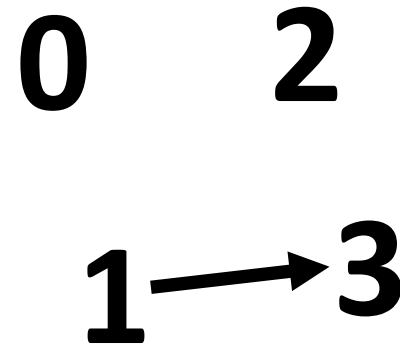
- edge[0]=1

- edge[1]=3

- Index[1] = 1

- edge[2] = 0

- edge[3]=3



MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- `nnodes = 2`

- `index[0] = 2`

- `edge[0]=1`

- `edge[1]=3`

- `Index[1] = 1`

- `edge[2] = 0`

- `edge[3]=3`

- `index[2] = 0`

0

2

1

3

MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

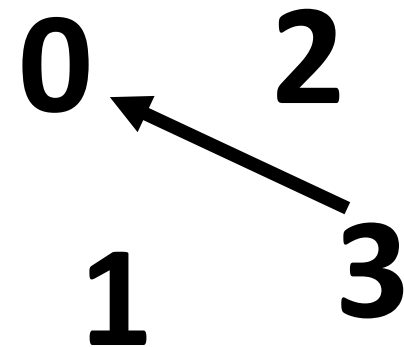
```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- `nnodes = 2`
- `index[0] = 2`
 - `edge[0]=1`
 - `edge[1]=3`
- `Index[1] = 1`
 - `edge[2] = 0`
- `edge[3]=3`
- `index[2] = 0`
- `Index[3] = 2`
 - `edge[4] = 0`



MPI_Graph_create example

- What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

- `nnodes = 2`

- `index[0] = 2`

- `edge[0]=1`

- `edge[1]=3`

- `Index[1] = 1`

- `edge[2] = 0`

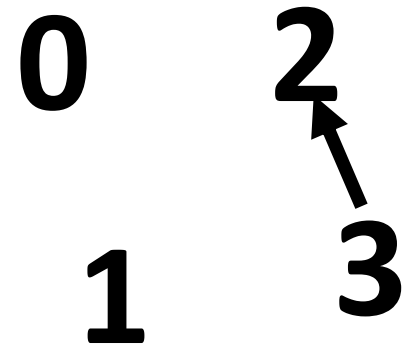
- `edge[3]=3`

- `index[2] = 0`

- `Index[3] = 2`

- `edge[4] = 0`

- `edge[5] = 2`



MPI_Graph_create example

■ What you want:

process	neighbors
0	1,3
1	0,3
2	
3	0,2

```
int nnodes=4;
```

```
int index[4] = {2 , 4, 4, 6};
```

```
int edges[index[nnodes-1]] = {1, 3, 0, 3, 0, 2};
```

```
MPI_Graph_create(comm_old, nnodes, index, edges, false, &comm_graph);
```

■ nnodes = 2

■ index[0] = 2

■ edge[0]=1

■ edge[1]=3

■ Index[1] = 2

■ edge[2] = 0

■ edge[3]=3

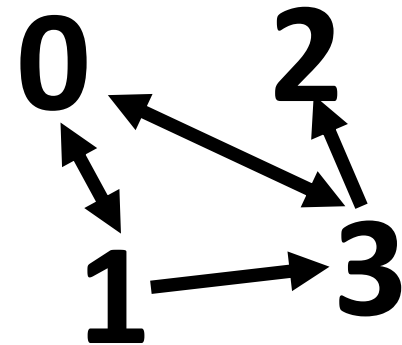
■ index[2] = 0

■ Index[3] = 2



■ edge[4] = 0

■ edge[5] = 2

■ Topology:





Neighborhood Graph Topology creat.



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

comm_old is the initial communicator containing the processes we want in the topology

Neighborhood Graph Topology creat.



```
int MPI_Dist_graph_create_adjacent (
```

```
    MPI_Comm comm_old(in),
```

```
    int indegree(in),
```

```
    const int sources[](in),
```

```
    const int sourceweights[](in),
```

```
    int outdegree (in),
```

```
    const int destinations[](in),
```

```
    const int destweights[] (in),
```

```
    MPI_Info info(in),
```

```
    int reorder(in),
```

```
    MPI_Comm *comm_dist_graph (out)
```



```
);
```

indegree : number of incoming edges
= number of processes with current rank as destination.

sources[*indegree*] : list of processes
with current rank as a destination.

sourceweights[*indegree*] : weight of
each incoming edges.

Neighborhood Graph Topology creat.



```
int MPI_Dist_graph_create_adjacent (
```

```
    MPI_Comm comm_old(in),
```

```
    int indegree(in),
```

```
    const int sources[](in),
```

```
    const int sourceweights[](in),
```

```
    int outdegree (in),
```

```
    const int destinations[](in),
```

```
    const int destweights[] (in),
```

```
    MPI_Info info(in),
```

```
    int reorder(in),
```

```
    MPI_Comm *comm_dist_graph (out)
```



```
);
```

outdegree : number of exiting edges =
number of processes with current rank as a
source.

destinations[*outdegree*] : list of
processes with current rank as a source.

Sourceweights[*outdegree*] : weight
of each exiting edges.



Neighborhood Graph Topology creat.



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

info may be used to influence interpretation of weights (higher is better or lower is better, ...)



Neighborhood Graph Topology creat.



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

reorder boolean, *true* or *false*, to specify if ranks may be reordered or not

Neighborhood Graph Topology creat.



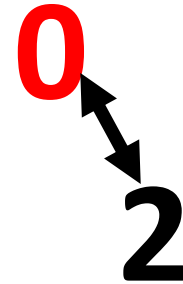
```
int MPI_Dist_graph_create_adjacent (  
    MPI_Comm comm_old(in),  
    int indegree(in),  
    const int sources[](in),  
    const int sourceweights[](in),  
    int outdegree (in),  
    const int destinations[](in),  
    const int destweights[] (in),  
    MPI_Info info(in),  
    int reorder(in),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

Comm_dist_graph is the new
communicator embedding the topology

MPI_Dist_graph_create_adjacent ex.

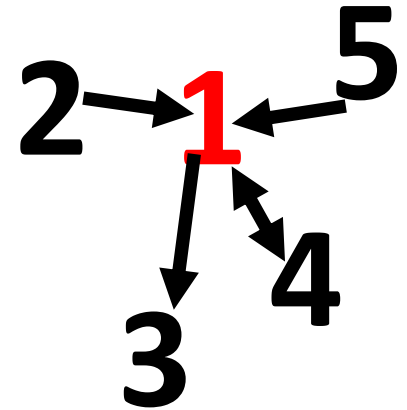


- Rank 0:
 - `sources[1] = {2}`
 - `destinations[1] = {2}`



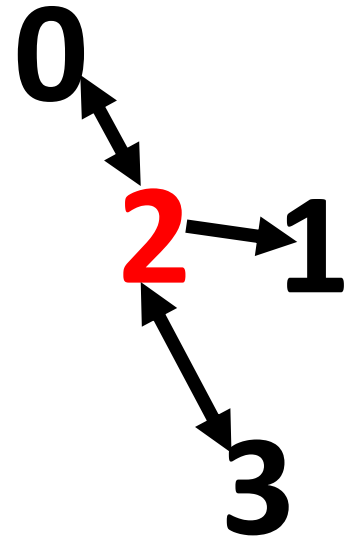
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - `sources[1] = {2}`
 - `destinations[1] = {2}`
- Rank 1:
 - `sources[3] = {2,4,5}`
 - `destinations[2] = {3,4}`



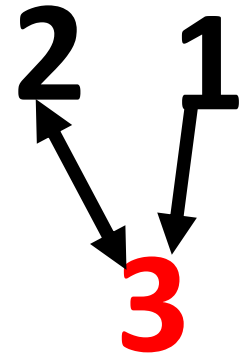
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - `sources[1] = {2}`
 - `destinations[1] = {2}`
- Rank 1:
 - `sources[3] = {2,4,5}`
 - `destinations[2] = {3,4}`
- Rank 2:
 - `sources[2] = {0,3}`
 - `destinations[3] = {0,1,3}`



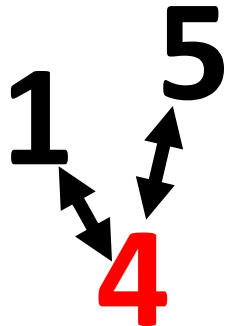
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - $\text{sources}[1] = \{2\}$
 - $\text{destinations}[1] = \{2\}$
- Rank 1:
 - $\text{sources}[3] = \{2,4,5\}$
 - $\text{destinations}[2] = \{3,4\}$
- Rank 2:
 - $\text{sources}[2] = \{0,3\}$
 - $\text{destinations}[3] = \{0,1,3\}$
- Rank 3:
 - $\text{sources}[2] = \{2,1\}$
 - $\text{destinations}[1] = \{2\}$



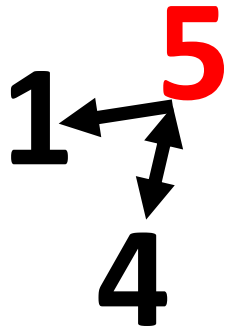
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - sources[1] = {2}
 - destinations[1] = {2}
- Rank 1:
 - sources[3] = {2,4,5}
 - destinations[2] = {3,4}
- Rank 2:
 - sources[2] = {0,3}
 - destinations[3] = {0,1,3}
- Rank 3:
 - sources[2] = {2,1}
 - destinations[1] = {2}
- Rank 4:
 - sources[2] = {1,5}
 - destinations[2] = {1,5}



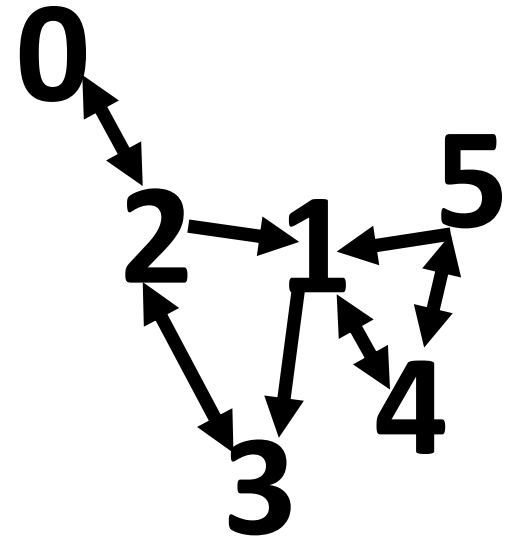
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - `sources[1] = {2}`
 - `destinations[1] = {2}`
- Rank 1:
 - `sources[3] = {2,4,5}`
 - `destinations[2] = {3,4}`
- Rank 2:
 - `sources[2] = {0,3}`
 - `destinations[3] = {0,1,3}`
- Rank 3:
 - `sources[2] = {2,1}`
 - `destinations[1] = {2}`
- Rank 4:
 - `sources[2] = {1,5}`
 - `destinations[2] = {1,5}`
- Rank 5:
 - `sources[1] = {4}`
 - `destinations[2] = {1,4}`



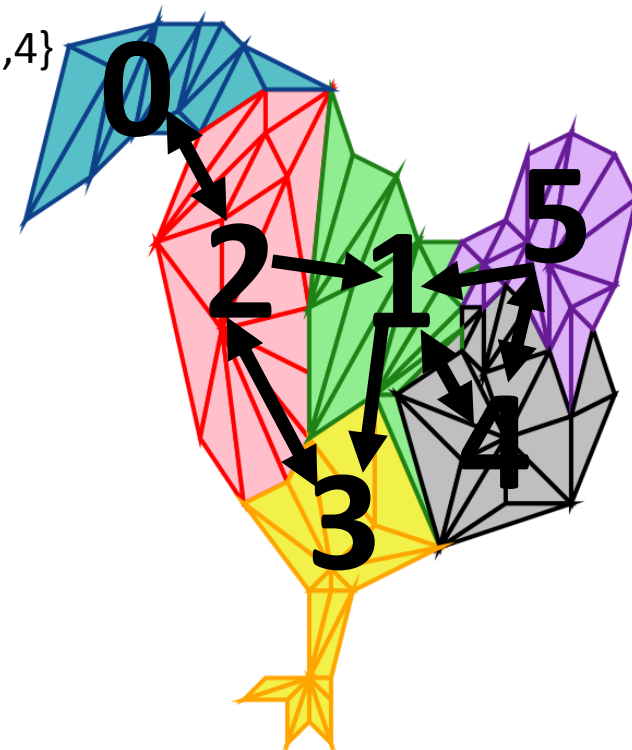
MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - sources[1] = {2}
 - destinations[1] = {2}
- Rank 1:
 - sources[3] = {2,4,5}
 - destinations[2] = {3,4}
- Rank 2:
 - sources[2] = {0,3}
 - destinations[3] = {0,1,3}
- Rank 3:
 - sources[2] = {2,1}
 - destinations[1] = {2}
- Rank 4:
 - sources[2] = {1,5}
 - destinations[2] = {1,5}
- Rank 5:
 - sources[1] = {4}
 - destinations[2] = {1,4}
- Topology:



MPI_Dist_graph_create_adjacent ex.

- Rank 0:
 - sources[1] = {2}
 - destinations[1] = {2}
- Rank 1:
 - sources[3] = {2,4,5}
 - destinations[2] = {3,4}
- Rank 2:
 - sources[2] = {0,3}
 - destinations[3] = {0,1,3}
- Rank 3:
 - sources[2] = {2,1}
 - destinations[1] = {2}
- Rank 4:
 - sources[2] = {1,5}
 - destinations[2] = {1,5}
- Rank 5:
 - sources[1] = {4}
 - destinations[2] = {1,4}
- Topology:



Dist Graph Topology creation



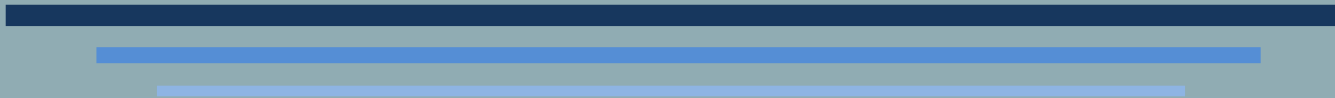
- `MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[], const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph);`
- Combine both of the two previous graph topology constructor
 - Each MPI rank specify only a subpart of the topology
 - May be any subpart, not only subparts where the current MPI rank is involved
 - Same edges may be specified several times by different MPI ranks

MPI Topology test



- `MPI_Topo_test(MPI_Comm comm, int *status);`
 - IN comm communicator (handle)
 - OUT status topology type of communicator comm (state)
- This function returns in *status* the type of the topology associated with communicator *comm*
- Returned value in status may be :
 - `MPI_GRAPH`
 - `MPI_CART`
 - `MPI_DIST_GRAPH`
 - `MPI_UNDEFINED`

MPI NEIGHBORHOOD COLLECTIVES



MPI Neighborhood collectives



- Definition

- The MPI Neighborhood collectives are collective communications that apply on a communicator with a topology
- Only collectives considered are All-to-All collectives with no operations: Allgather and Alltoall
 - And there derivatives:
 - Allgatherv, Alltoallv, Alltoallw
 - And there non-blocking equivalent
 - lallgather, lallgatherv, lalltoall, lalltoallv, lalltoallw
- Same rules as for other collective communications

Allgather PP-C2



```
int MPI_Allgather (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

} Arguments corresponding to
sent data

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

} Arguments corresponding to
received data

```
    MPI_Comm comm(in),
```

```
);
```

Neighborhood Allgather



```
int MPI_Neighborhood_allgather (
```

```
void *sendbuf(in),
```

```
int sendcount(in),
```

```
MPI_Datatype sendtype(in),
```

```
void *recvbuf(out),
```

```
int recvcount(in),
```

```
MPI_Datatype recvtype(in),
```

```
MPI_Comm comm(in),
```

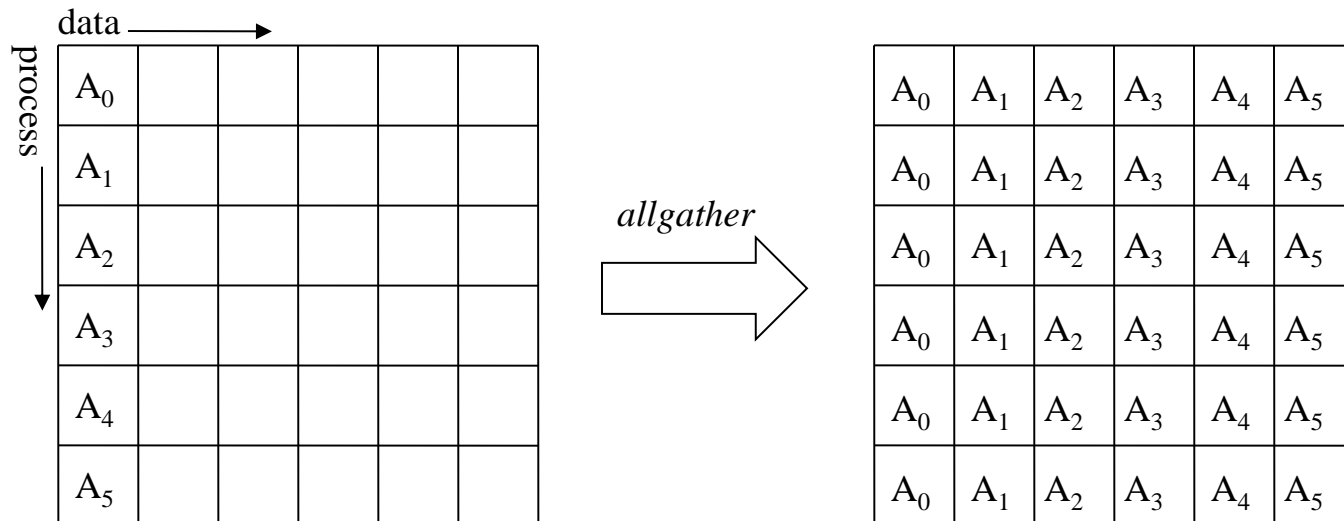
```
);
```

Arguments corresponding to
sent data

Arguments corresponding to
received data

Allgather PP-C2

- Equivalent to gather collective operation except that every process involved inside the communication receives the final result.
- Allgather \approx Gather + Broadcast





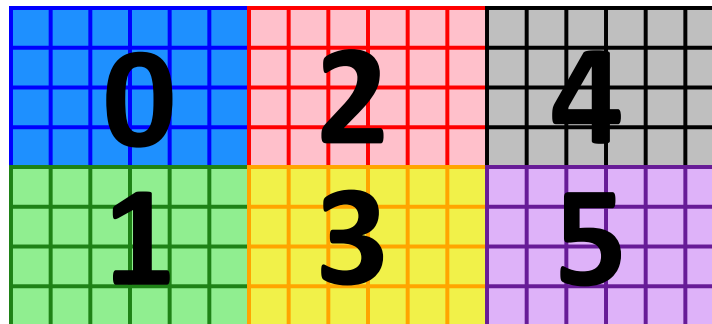
Neighborhood Allgather



- Equivalent to allgather collective operation except that every process involved inside the communication only receives data from incoming edges.

Neighborhood Allgather

- Equivalent to allgather collective operation except that every process involved inside the communication only receives data from incoming edges.
- Consider our cartesian example with bidirectional edges

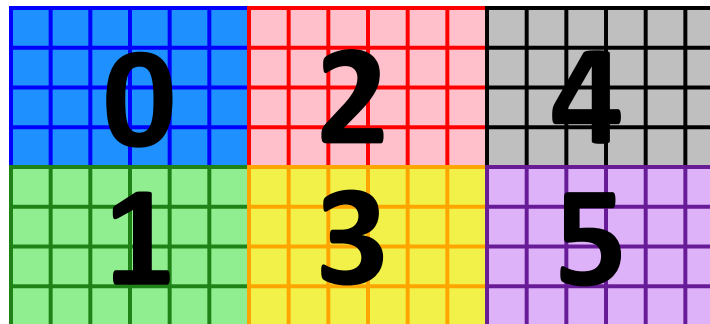




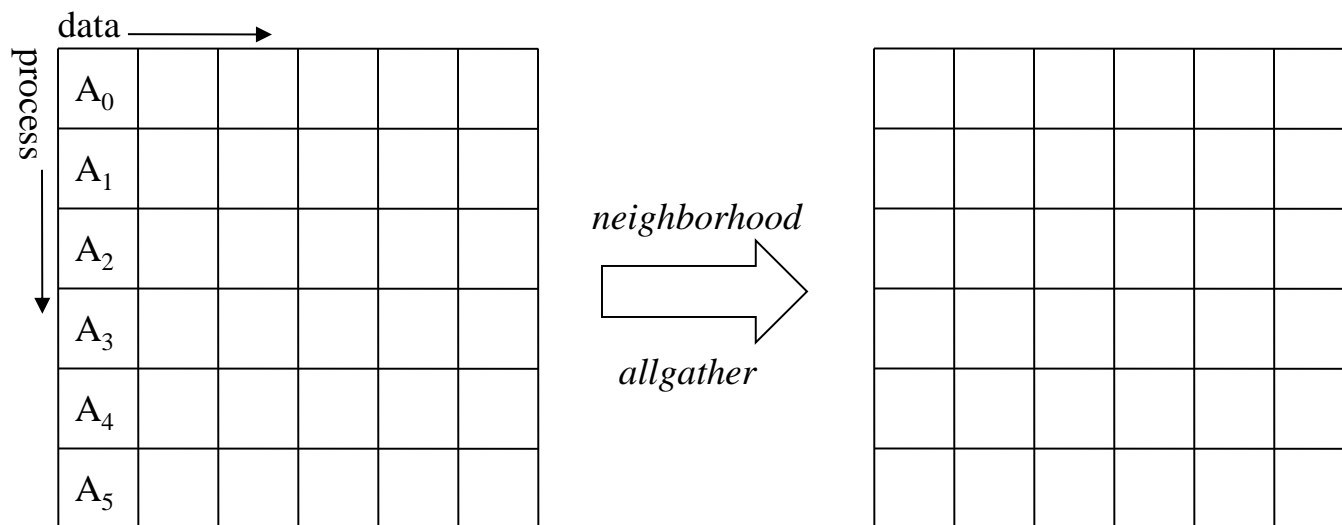
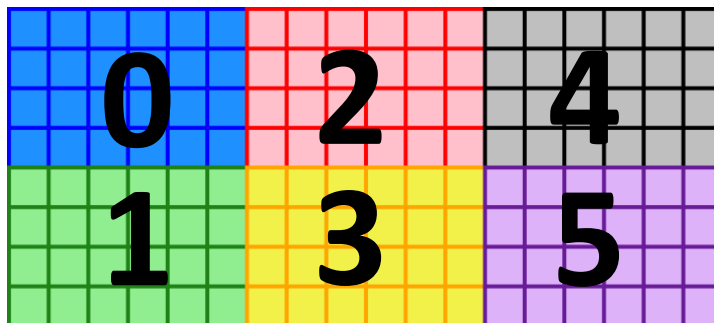
Neighborhood Allgather



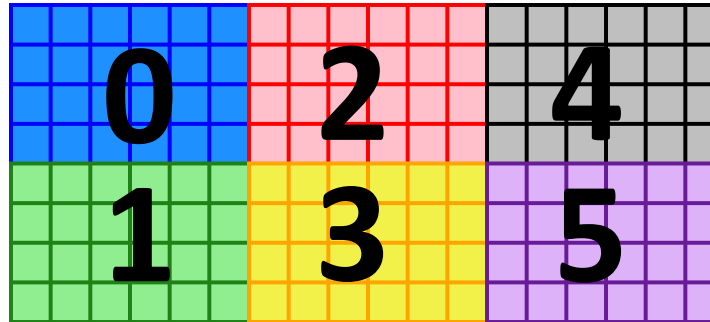
- Equivalent to allgather collective operation except that every process involved inside the communication only receives data from incoming edges.
- Consider our cartesian example with bidirectional edges
- $0 \leftarrow 1,2$ / $1 \leftarrow 0,3$ / $2 \leftarrow 0,3,4$ / $3 \leftarrow 1,2,5$ / $4 \leftarrow 2,5$ / $5 \leftarrow 3,4$



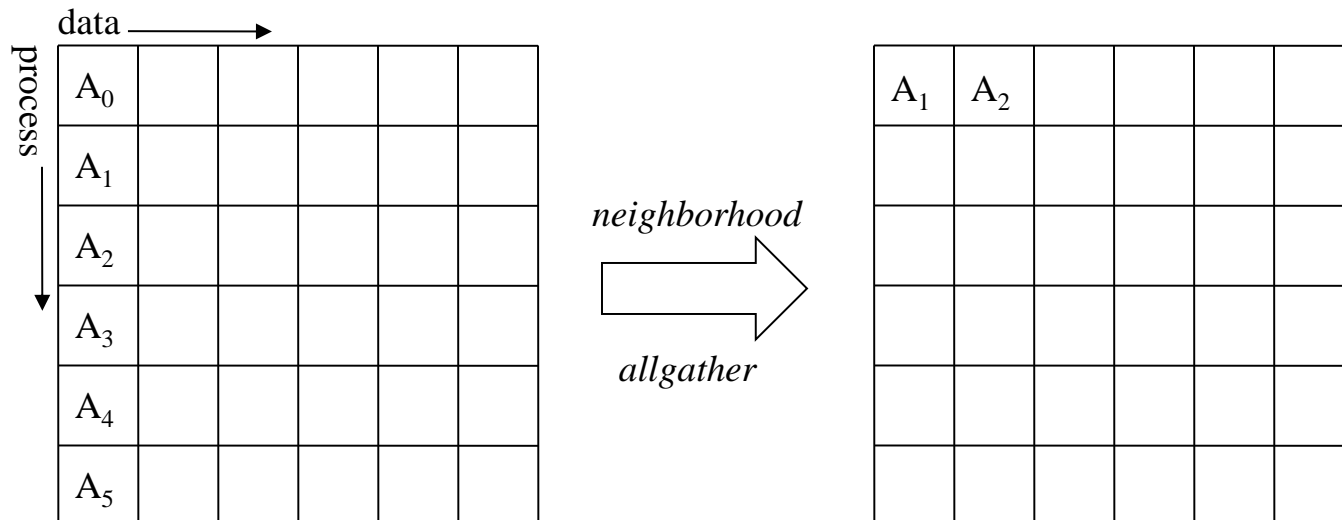
Neighborhood Allgather



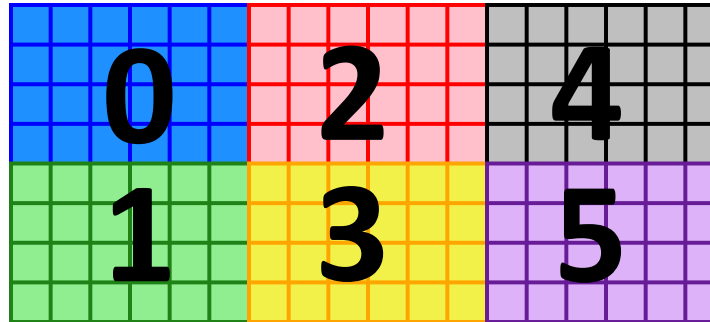
Neighborhood Allgather



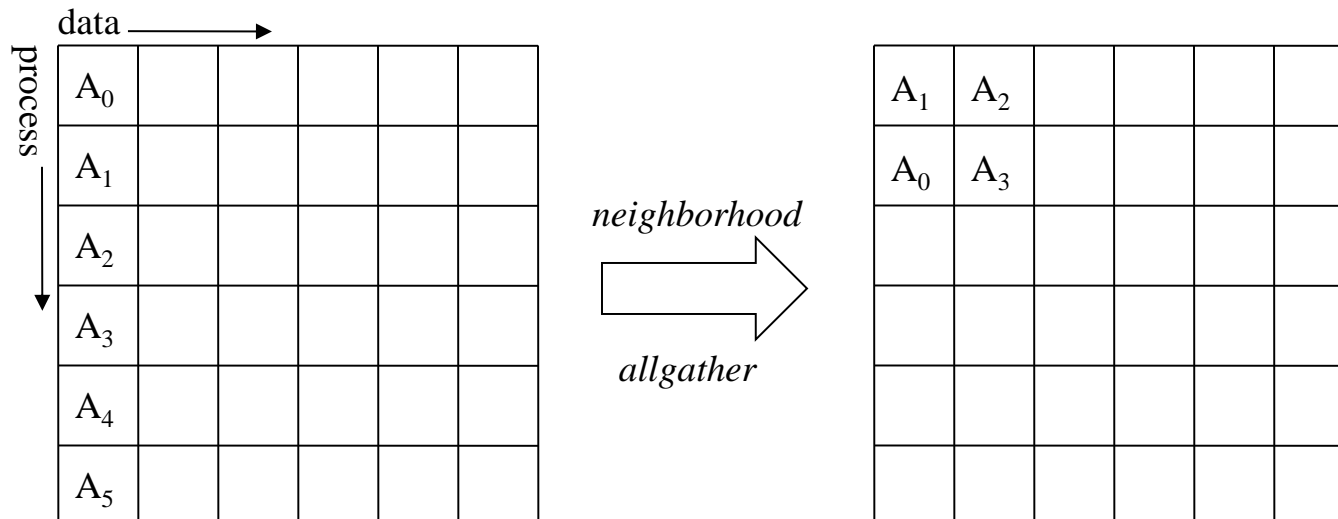
■ 0 < -1, 2



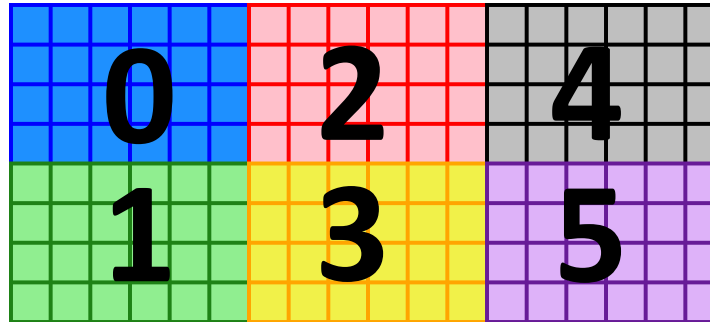
Neighborhood Allgather



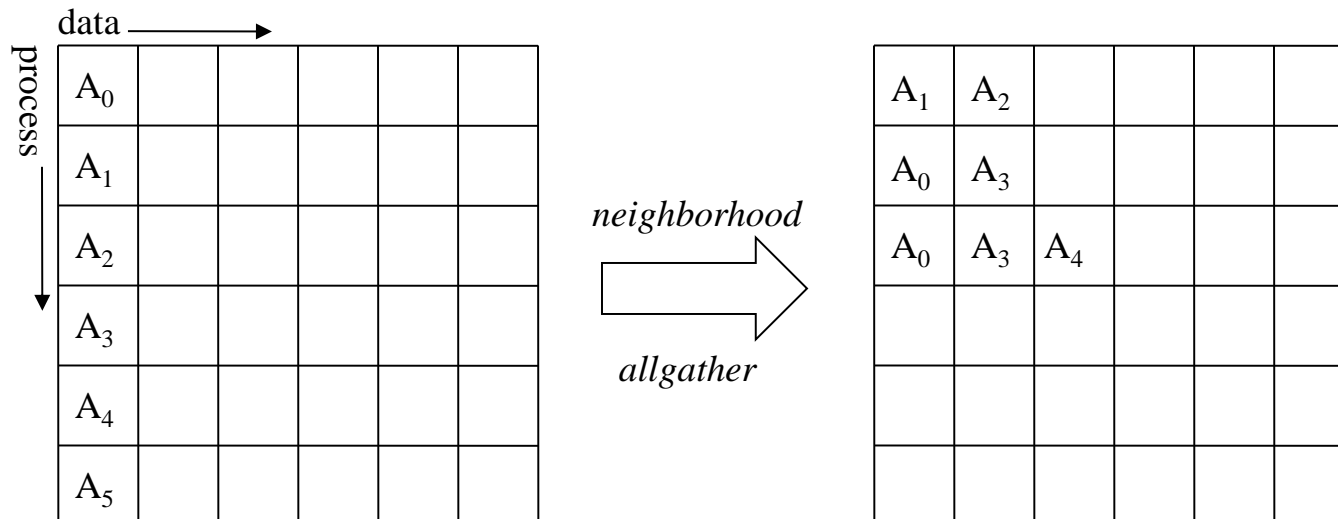
■ $0 < -1, 2 / \underline{1} < -0, 3$



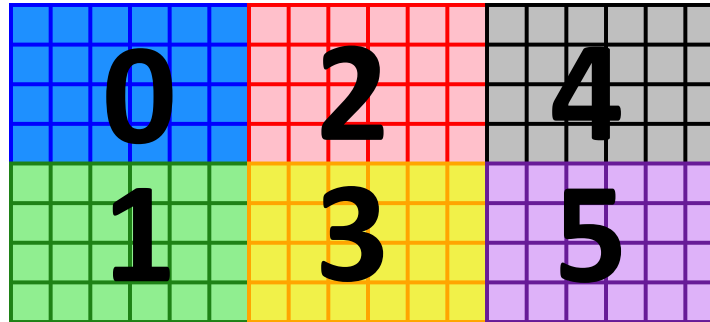
Neighborhood Allgather



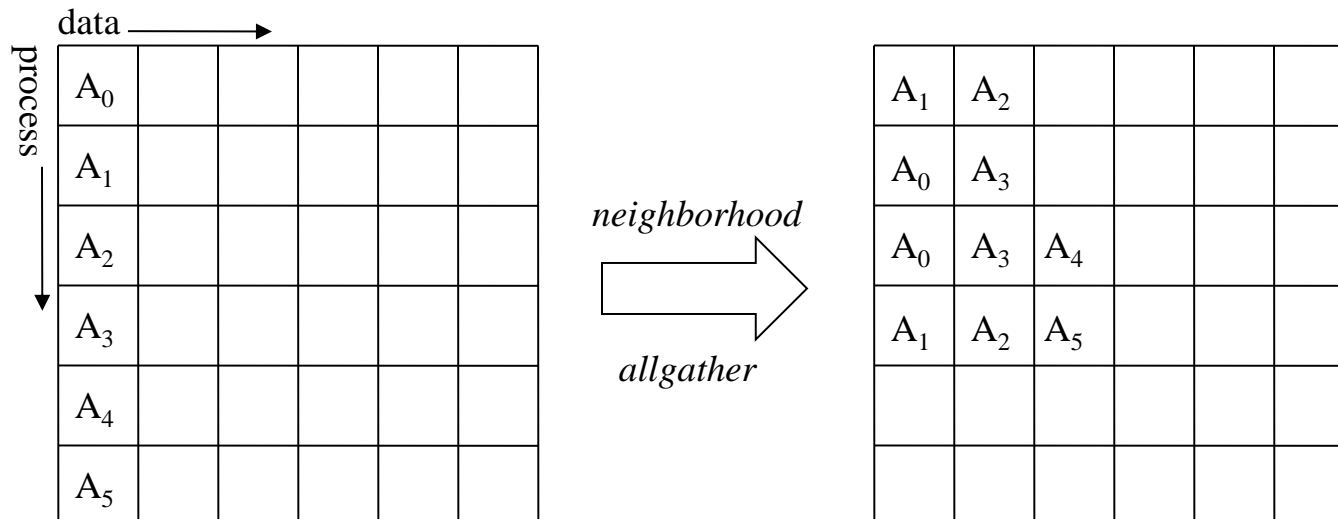
- $0 < -1, 2$ / $1 < -0, 3$ / $2 < -0, 3, 4$



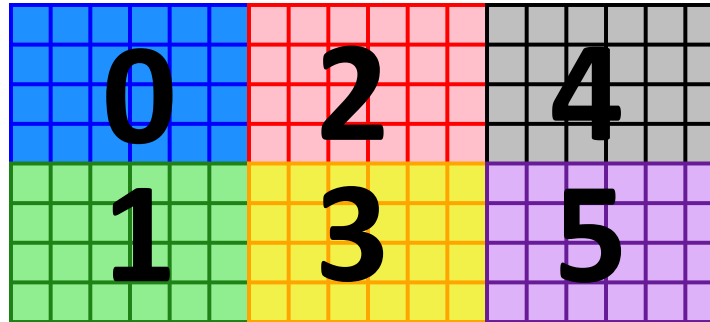
Neighborhood Allgather



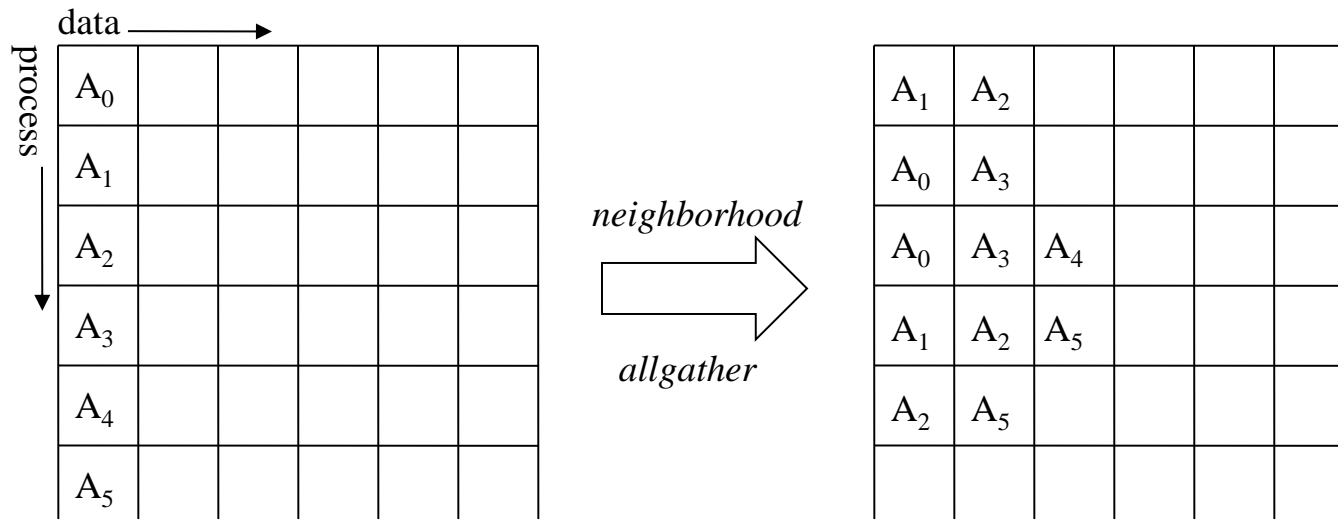
- 0 < -1,2 / 1 < -0,3 / 2 < -0,3,4 / 3 < -1,2,5



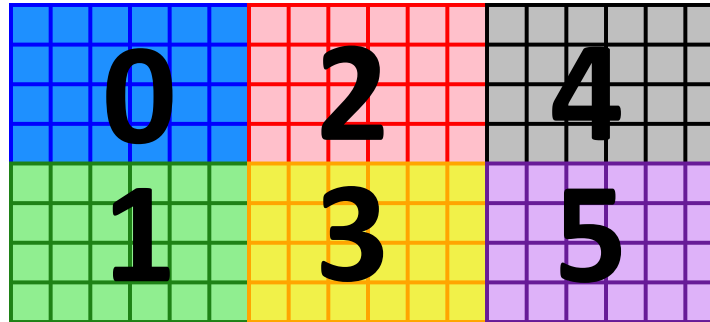
Neighborhood Allgather



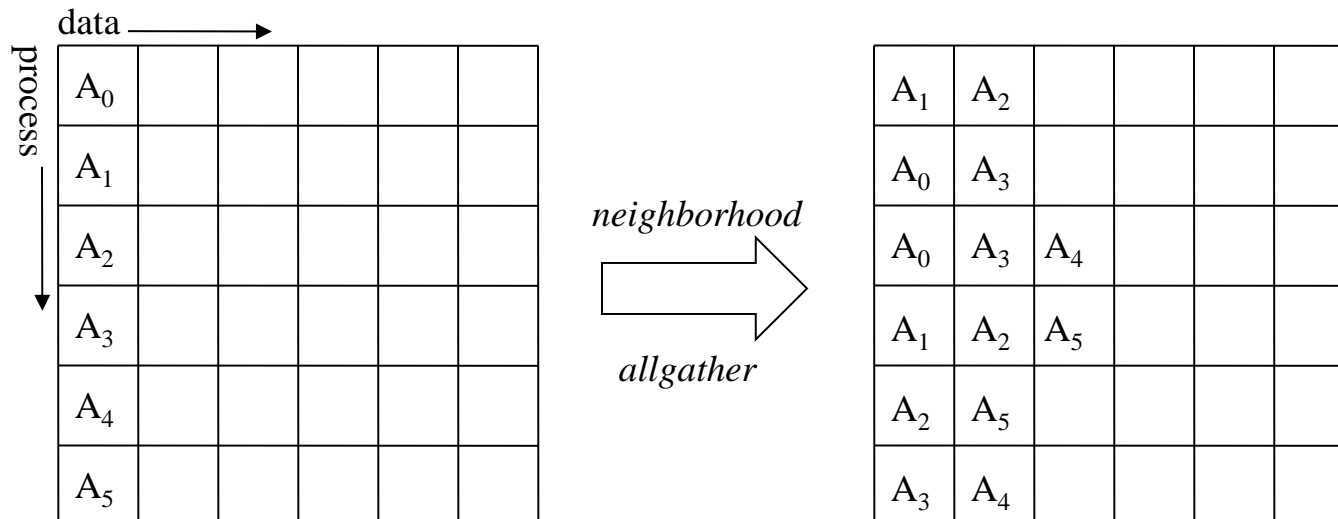
- 0 < -1,2 / 1 < -0,3 / 2 < -0,3,4 / 3 < -1,2,5 / 4 < -2,5



Neighborhood Allgather



- 0 < -1,2 / 1 < -0,3 / 2 < -0,3,4 / 3 < -1,2,5 / 4 < -2,5 / 5 < -3,4



Alltoall PP-C2



```
int MPI_Alltoall (
```

```
    void *sendbuf(in),
```

```
    int sendcount(in),
```

```
    MPI_Datatype sendtype(in),
```

```
    void *recvbuf(out),
```

```
    int recvcount(in),
```

```
    MPI_Datatype recvtype(in),
```

```
    MPI_Comm comm(in),
```

```
);
```

} Arguments corresponding to
sent data

} Arguments corresponding to
received data

Neighborhood Alltoall



```
int MPI_Neighborhood_Alltoall (
```

```
void *sendbuf(in),
```

```
int sendcount(in),
```

```
MPI_Datatype sendtype(in),
```

```
void *recvbuf(out),
```

```
int recvcount(in),
```

```
MPI_Datatype recvtype(in),
```

```
MPI_Comm comm(in),
```

```
);
```

} Arguments corresponding to
sent data

} Arguments corresponding to
received data



Neighborhood Alltoall



- Equivalent to `MPI_Neighborhood_allgather` collective operation except that every process sends different data to each destination.
- `Neighborhood_allgatherv` and `Neighborhood_alltoallv` are doing the same communication pattern, except that you can specify different displacement in the receive buffer
 - Like usual `allgather` and `alltoall`
- `Neighborhood_alltoallv` is doing the same communication pattern, except that you can specify different displacement and different types for each send/recv
 - Like usual `alltoallw`