Advanced virtual topologies

Advanced Message-Passing Programming (INFR11169)

Ludovic Capelli

EPCC

February 1, 2023

Individuals

The material presented in this lecture is inspired from content developed by:

- David Henty

Contributors

In this material:

- The slides are created using LaTeX *Beamer* available at `https://ctan.org/pkg/beamer`.

License - Creative Commons BY-NC-SA-4.0[1]

Table of Contents

Structured meshes

Definition

A *structured mesh* is a mesh where one can find a connectivity pattern that applies to every element in the mesh.

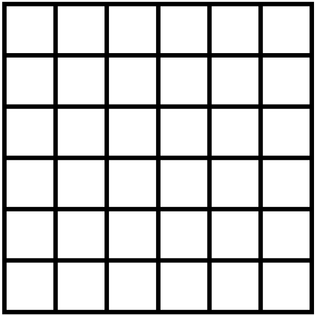
Structured meshes

Definition

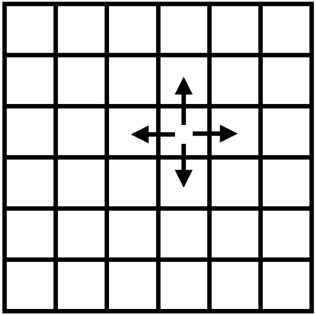
A *structured mesh* is a mesh where one can find a connectivity pattern that applies to every element in the mesh.

Note: the mesh itself might "grow" non-uniformly, to adapt to boundaries for instance, however its elements are still connected using the same pattern.

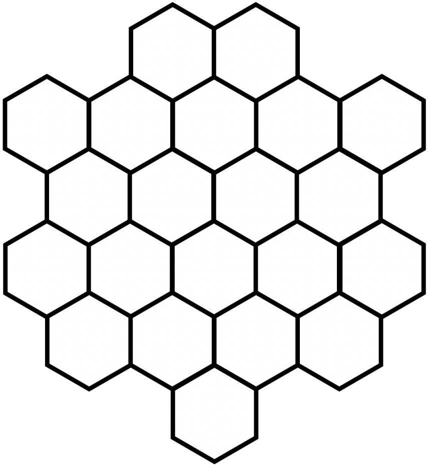
Example of a structured 2D mesh made of squares



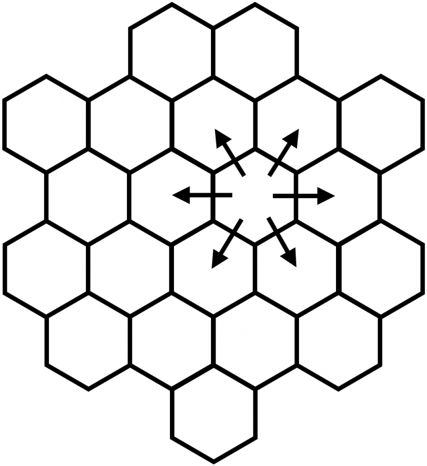
Typical communication pattern



Example of a structured 2D mesh made of hexagons



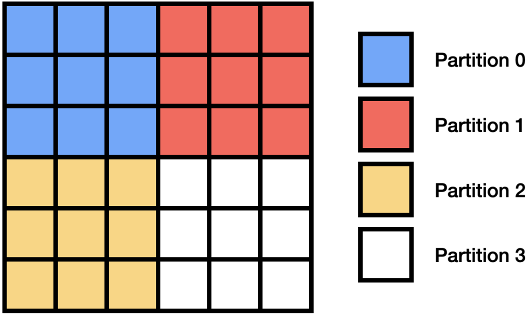
Typical communication pattern



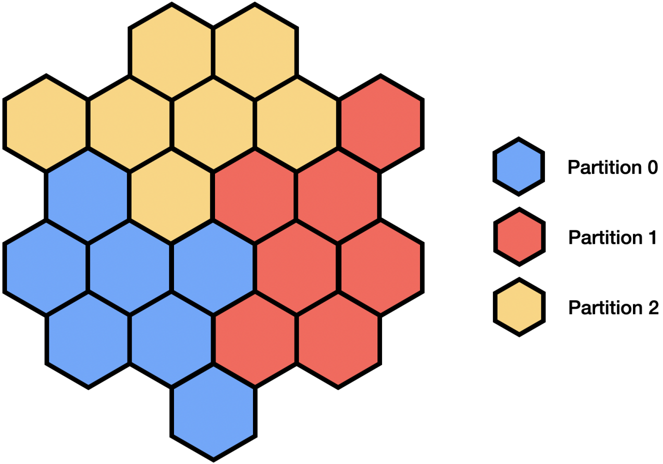
Load-balancing

In structured meshes, the load-balance typically comes down to balancing the number of elements per worker.

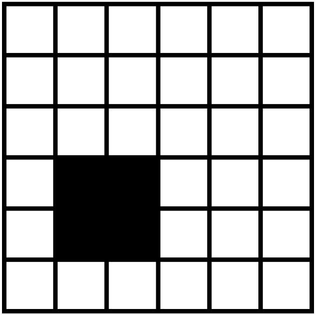
Load-balancing: 9 elements per MPI process



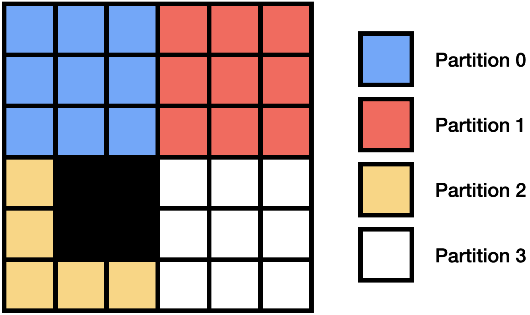
Load-balancing: 6 elements per MPI process



Mesh with a "hole"



Mesh with a "hole": basic partitioning



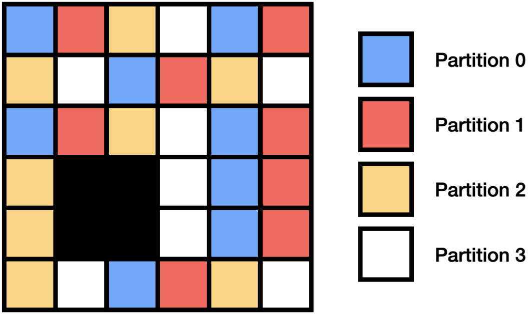
Mesh with a "hole": basic partitioning

- Pros
  - No need to alter the underlying decomposition.

Mesh with a "hole": basic partitioning

- Pros
  - No need to alter the underlying decomposition.
- Cons
  - Load-imbalance present: 3 MPI processes have 9 elements, whereas 1 MPI process has 5 elements.

Mesh with a hole: cyclic partitioning



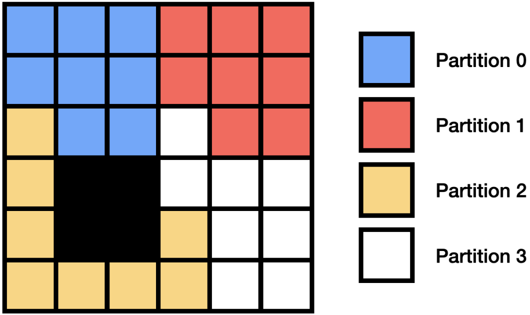
Mesh with a hole: cyclic partitioning

- Pros
  - Every MPI process is assigned 8 elements.

Mesh with a hole: cyclic partitioning

- Pros
  - Every MPI process is assigned 8 elements.
- Cons
  - Hindered data locality as elements are rarely consecutive.

Mesh with a hole: more advanced partitioning



Mesh with a hole: more advanced partitioning

- Pros
  - Every MPI process has been assigned an identical number of elements (8).
  - Most elements are consecutive.

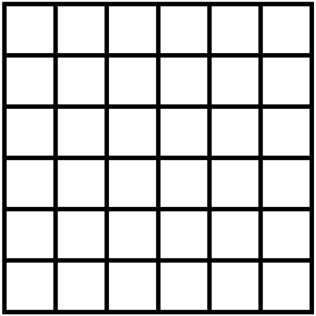
Mesh with a hole: more advanced partitioning

- Pros
  - Every MPI process has been assigned an identical number of elements (8).
  - Most elements are consecutive.
- Cons
  - More complex.
  - Takes more time to complete.

Limitations

- Structured meshes have interesting properties that make their storage in memory and indexing, among other aspects, convenient.
- However, real world meshes are more complex and do not decompose into a structured mesh.

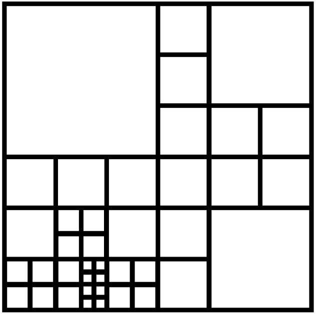
Original example



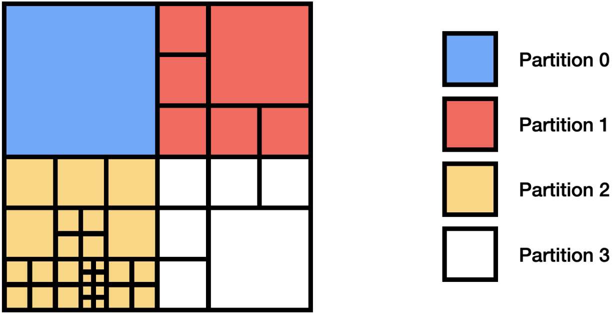
Applications

- Computational fluid dynamics: modelling a volume surrounding a formula 1 car. Volume units closer to the car need to be smaller to increase accuracy.
- Spatial simulations: not all zones of space contain the same number of bodies. Zones with galaxies for instance mechanically involve more interactions to simulate.
- Video games: certain zones are more populated than others. Think of the main city compared to the middle of a forest.

Unstructured meshes: variable cell sizes



Unstructured meshes: variable cell sizes



Mesh partitioning

What is the fitness function?

- The number of cells per partition?
- The connectivity of the partitions?
- How adjacent cells are?
- Any "hole" to deal with?
- Any disjoint partition to handle?

Mesh partitioning

What is the fitness function?

- The number of cells per partition?
- The connectivity of the partitions?
- How adjacent cells are?
- Any "hole" to deal with?
- Any disjoint partition to handle?

In addition to all the above, one must also consider the communication patterns, the execution flow, the variability of load-balancing at runtime...

Partitioning: hard.

- There exists multiple more advanced partitioning strategies.
- Beyond the scope of this module.
- ParMETIS is the solution commonly used.
- All involve a time against quality compromise.

The mapping from data to hardware, through MPI processes

- First, you find how to decompose your data across your MPI processes.
- Then, you find the best mapping of MPI processes to the underlying hardware.

Table of Contents

Virtual topology

Definition

A *virtual topology* is the arrangement of MPI processes.[2]

[2] See Section 8.1 in MPI standard version 4.0

Purpose

Question

Why do we use virtual topologies?

Purpose

Question

Why do we use virtual topologies?

Reason 1

They provide the developer with convenient and more intuitive constructions, such as finding ranks with shifts in a Cartesian virtual topology.

Purpose

Question

Why do we use virtual topologies?

Reason 1

They provide the developer with convenient and more intuitive constructions, such as finding ranks with shifts in a Cartesian virtual topology.

Reason 2

They inform the MPI runtime of the patterns of communications that will be in place. This information can be leveraged by the underlying runtime to improve the placement of MPI processes to hardware.

Multiple factors to take in consideration

- An MPI process contains one thread[3], which is eventually run by a hardware thread.
- Certain hardware threads may be on the same core.
- Certain caches may be shared by multiple cores (typically, L3).
- Certain cores may belong to the same processor and thus access the same memory module(s) (c.f.: NUMA region).
- Certain processors may be on different nodes.
- Certain nodes may be on different racks.

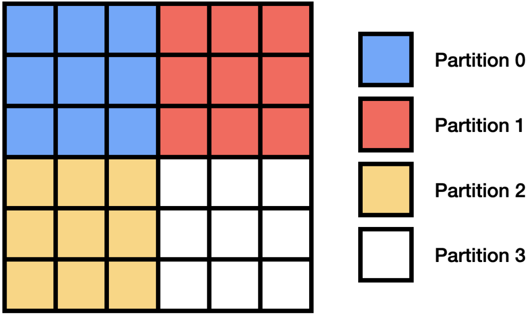
By declaring a virtual topology, the MPI runtime environment has extra information to leverage for a more efficient placement.

[3] We do not consider hybrid programming for now

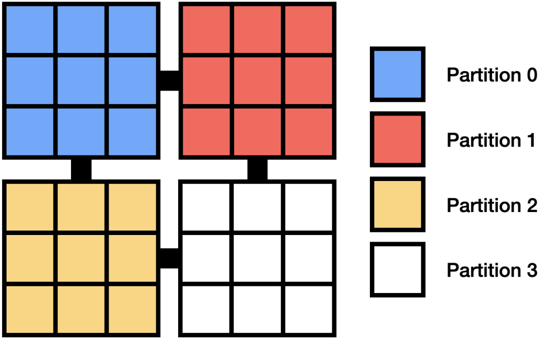
Virtual topologies for structured meshes

For structured meshes, Cartesian topologies work pretty well. These are the ones you have seen and practiced in MPP last semester.

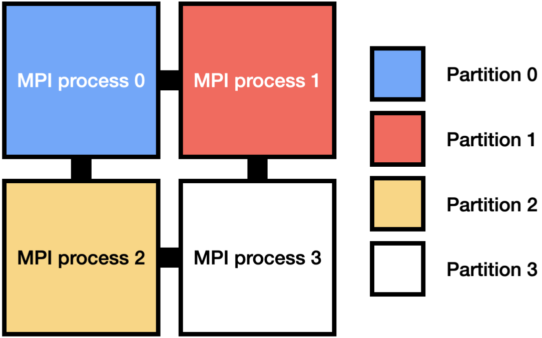
From data to virtual topology: partition data



From data to virtual topology: analyse connectivity



From data to virtual topology: create vitual topology



MPI_Cart_create

```c
int MPI_Cart_create(MPI_Comm old_communicator,
                    int dims_count,
                    const int dims[],
                    const int periods[],
                    int reorder,
                    MPI_Comm new_communicator);
```

MPI_Cart_create

```
PROCEDURE MPI_Cart_create(old_communicator, dims_count, dims, periods, reorder, new_communicator, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: old_communicator
    INTEGER, INTENT(IN) :: dims_count
    INTEGER, INTENT(IN) :: dims(dims_count)(dims_count)
    LOGICAL, INTENT(IN) :: periods(dims_count)(dims_count)
    INTEGER, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: new_communicator
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Reordering

Can have an impact on performance.

MPI_Dims_create: finding dimensions

```
int MPI_Dims_create(int process_number,
                    int dims_count,
                    int dims[]);
```

MPI_Dims_create: finding dimensions

```
PROCEDURE MPI_Dims_create(process_number, dims_count,
    dims, ierror)
    INTEGER, INTENT(IN) :: process_number
    INTEGER, INTENT(IN) :: dims_count
    INTEGER, INTENT(INOUT) :: dims(dims_count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

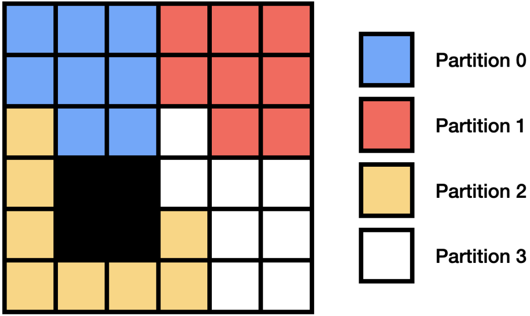
MPI_Cart_shift: finding neighbours

```
int MPI_Cart_shift(MPI_Comm communicator,
                   int direction,
                   int displacement,
                   int* source,
                   int* destination);
```

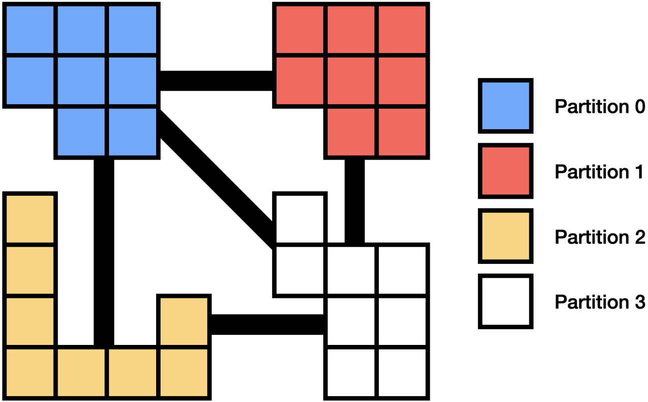
MPI_Cart_shift: finding neighbours

```
PROCEDURE MPI_Cart_shift(communicator, direction,
    displacement, source, destination, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: communicator
    INTEGER, INTENT(IN) :: direction
    INTEGER, INTENT(IN) :: displacement
    INTEGER, INTENT(OUT) :: source
    INTEGER, INTENT(OUT) :: destination
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

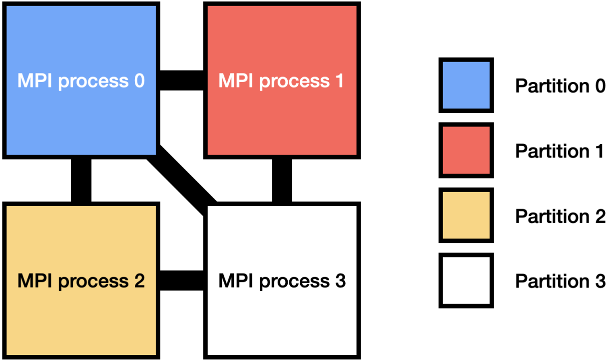
Slightly less Cartesian topologies



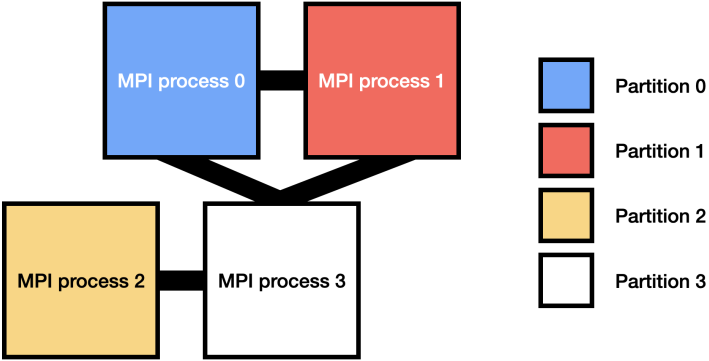
Slightly less Cartesian topologies



Slightly less Cartesian topologies



Slightly less Cartesian topologies



Non-Cartesian virtual topologies

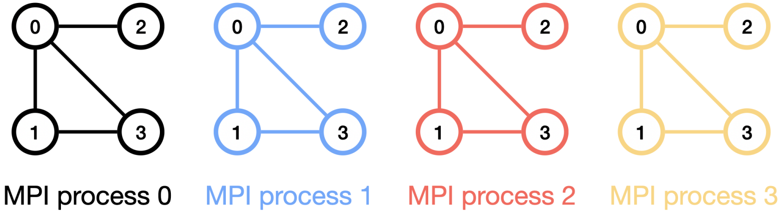
- The same way MPI provides datatype constructors with different levels of flexibility, it provides virtual
- When a Cartesian topology does not apply, one can use a graph topology, which is the most generic: you can create any topology using graphs.topology constructors with different levels of flexibility.

Creating graph virtual topologies

There are three ways to create a graph virtual topology:

- `MPI_Graph_create`
- `MPI_Dist_graph_create`
- `MPI_Dist_graph_create_adjacent`

MPI_Graph_create



When creating a graph topology using `MPI_Graph_create`, every MPI process pass the entirety of the graph topology.

MPI_Graph_create

```c
int MPI_Graph_create(MPI_Comm old_comm,
                     int number_of_nodes,
                     const int indexes[],
                     const int edges[],
                     int reorder,
                     MPI_Comm* new_comm);
```

MPI_Graph_create

```
PROCEDURE MPI_Graph_create(old_comm, number_of_nodes,
    index, edges, reorder, new_comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: old_comm
    INTEGER, INTENT(IN) :: number_of_nodes
    INTEGER, INTENT(IN) :: indexes(number_of_nodes)
    INTEGER, INTENT(IN) :: edges(*)
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: new_comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_Graph_create: the indexes parameter

indexes This parameter contains the **total** number of neighbours[4] of all MPI processes with a rank lesser than, or equal to, the position considered in `indexes`.

Example

The value at the fourth position, `indexes[3]` in C or `indexes(4)` in FORTRAN[5], must be equal to the total number of neighbours of the fourth first MPI processes altogether: namely 0, 1, 2 and 3.

[4] This design allows MPI to know the total number of entries in the `edges` parameter.

[5] If you use the 1-index applied by default.

`MPI_Graph_create`: the `edges` parameter

edges This parameter contains all edges of the graph, sorted by the source identifier. Edges are undirected so you do not have to specify both edges for each connection. Redundant and self edges are also allowed.

Example

Assuming MPI process 0 is connected to MPI processes 1, 3 and 4, and MPI process 1 is connected to MPI process 2 and 4, `edges` would contain: 1, 3, 4, 2, 4.

MPI_Graph_create

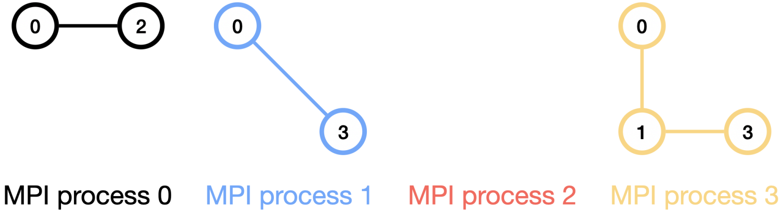
Since every MPI process has to specify the entire graph, this approach is not scalable, imagine the following with a graph of 100 billion edges, taking roughly 1TB of RAM:

- Every MPI process must store the entire graph.
- Every MPI process passes the entire graph over the network.

MPI_Graph_create

- Using MPI_Graph_create, since it does not distribute, even taking all 700,000+ cores from the ARCHER2 supercomputer will not help, because every MPI process would still need 1TB of RAM regardless.
- The second approach is to distribute this virtual topology construction, by having each MPI process specify only a portion of the entire graph, using MPI_Dist_graph_create

MPI_Dist_graph_create[6]



Every MPI process may specify 0, 1 or more edges. The edges specified do not have to contain the MPI process that passes them.

[6] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create[7]

```
int 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)
```

[7] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create[8]

```
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old,
    INTEGER, INTENT(IN) :: n
    INTEGER, INTENT(IN) :: sources(n)
    INTEGER, INTENT(IN) :: degrees(n)
    INTEGER, INTENT(IN) :: destinations(*)
    INTEGER, INTENT(IN) :: weights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

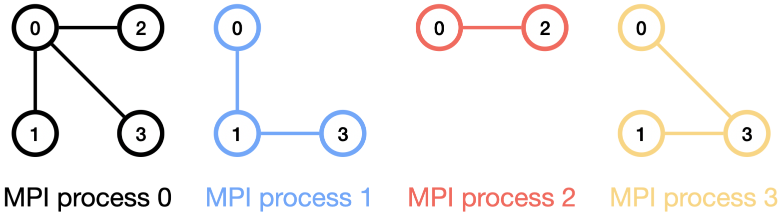
[8] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create_adjacent[9]

- MPI_Dist_graph_create allows MPI processes to pass any edge of the graph, which offers the best flexibility.
- However, MPI processes included in each edge specified must be somehow informed of this topological detail, which results in communication.
- The third approach to create a graph virtual topology is MPI_Dist_graph_create_adjacent, where each MPI process specifies all edges in which it is part of, and solely such edges.

[9] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create_adjacent[10]



[10] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create_adjacent[11]

```c
int 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);
```

[11] See Section 8.5.4 in MPI standard version 4.0

MPI_Dist_graph_create_adjacent[12]

```
MPI_Dist_graph_create_adjacent(comm_old, indegree,
    sources, sourceweights, outdegree, destinations,
    destweights, info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: indegree
    INTEGER, INTENT(IN) :: sources(indegree)
    INTEGER, INTENT(IN) :: sourceweights(*),
    INTEGER, INTENT(IN) :: outdegree
    INTEGER, INTENT(IN) :: destinations(outdegree)
    INTEGER, INTENT(IN) :: destweights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

[12] See Section 8.5.4 in MPI standard version 4.0