HIGH PERFORMANCE
COMPUTING CINECA
ITALY

Parallel Programming with MPI

Part IV - Communicators and virtual topologies

HPC SCHOOL
COSTA RICA

Centro Nacional de Alta Tecnología

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

COMMUNICATORS
Definitions, group constructors, communicator constructors

VIRTUAL TOPOLOGY
Definitions and examples

THE CARTESIAN COMMUNICATOR
Cartesian topology and constructor, utilities, cartesian partitioning

KABRÉ
SUPERCOMPUTADORA

Communicators

What are communicators?

Many users are familiar with the mostly used communicator:

MPI_COMM_WORLD

A **communicator** can be thought as a handle to a group.

- a **group** is a ordered set of processes
- each process is associated with a rank
- ranks are contiguous and start from zero

Groups allow collective operations to be operated on a subset of processes

Definitions & properties

Intracommunicators are used for communications within a single group
Intercommunicators are used for communications between two disjoint groups

Group management:

- All group operations are local
- Groups are not initially associated with communicators
- Groups can only be used for message passing within a communicator
- We can access groups, construct groups, destroy groups

Group accessors:

`MPI_GROUP_SIZE`

This routine returns the number of processes in the group

`MPI_GROUP_RANK`

This routine returns the rank of the calling process inside a given group

Group constructors

Group constructors are used to create new groups from existing ones (initially from the group associated with MPI_COMM_WORLD; you can use *mpi_comm_group* to get this).

Group creation is a local operation: no communication is needed

After the creation of a group, no communicator has been associated to this group, and hence no communication is possible within the new group

Group constructors

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
```

- This routine returns the group associated with the communicator comm

```
int MPI_Group_union(MPI_Group group_a, MPI_Group
    group_b, MPI_Group *newgroup);
```

- This returns the ensemble union of group_a and group_b

```
int MPI_Group_intersection(MPI_Group group_a,
    MPI_Group group_b, MPI_Group *newgroup);
```

- This returns the ensemble intersection of group_a and group_b

Group constructors

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
      MPI_Group *newgroup);
```

- This routine creates a new group that consists of all the n processes with ranks ranks[0]... ranks[n-1]

```
int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
      MPI_Group *newgroup);
```

- This routine creates a new group that consists of all the n processes after removing ranks[0]... ranks[n-1]

Example MPI_Group_incl:

group = {a,b,c,d,e,f,g,h,i,j}
n = 5
ranks = {0,3,8,6,2}
newgroup = {a,d,i,g,c}

Example MPI_Group_excl:

group = {a,b,c,d,e,f,g,h,i,j}
n = 5
ranks = {0,3,8,6,2}
newgroup = {b,e,f,h,j}

Communicator management

Communicator access operations are local, not requiring interprocess communication

Communicator constructors are collective and may require interprocess communications

We will cover in depth only intracommunicators, giving only some notions about intercommunicators.

Communicator accessors

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- Returns the number of processes in the group associated with the comm

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- Returns the rank of the calling process within the group associated with the comm

```
int MPI_Comm_compare(MPI_Comm comm_a, MPI_Comm
    comm_b, int *result);
```

- Returns:
  - MPI_IDENT if comm1 and comm2 are the same handle
  - MPI_CONGRUENT if comm1 and comm2 have the same group attribute
  - MPI_SIMILAR if the groups associated with comm1 and comm2 have the same members but in different rank order
  - MPI_UNEQUAL otherwise

Communicator constructors

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
```

- This returns a communicator newcomm identical to the communicator comm.

```
int MPI_Comm_create(MPI_Comm comm, int *rank);
```

- This collective routine must be called by all the process involved in the group associated with comm. It returns a new communicator that is associated with the group. MPI_COMM_NULL is returned to processes not in the group.

- **Note that the new group must be a subset of the group associated with comm!**

Example

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
     int rank, new_rank, nprocs, sendbuf, recvbuf;
     int ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
     MPI_Group orig_group, new_group;
     MPI_Comm new_comm;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     sendbuf = rank;
     MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
     if (rank < nprocs/2)
        MPI_Group_incl(orig_group, nprocs/2, ranks1, &new_group);
     else MPI_Group_incl(orig_group, nprocs/2, ranks2, &new_group);
     MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
     MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
     MPI_Group_rank (new_group, &new_rank);
     printf("rank= %d newrank= %d recvbuf=%d\n",rank,new_rank,recvbuf);
     MPI_Finalize();
     return 0;
}
```

Hypothesis: nprocs = 8 credits: http://static.msi.umn.edu

Example

RESULTS:

```
rank= 0 newrank= 0 recvbuf= 6

rank= 1 newrank= 1 recvbuf= 6

rank= 2 newrank= 2 recvbuf= 6

rank= 3 newrank= 3 recvbuf= 6

rank= 4 newrank= 0 recvbuf= 22

rank= 5 newrank= 1 recvbuf= 22

rank= 6 newrank= 2 recvbuf= 22

rank= 7 newrank= 3 recvbuf= 22
```

Hypothesis: nprocs = 8 credits: http://static.msi.umn.edu

Destructors

The communicators and groups from a process’ viewpoint are just handles.

Like all handles, there is a limited number available: you could (in principle) run out!

```
int MPI_Group_free(MPI_Group *group);
```

```
int MPI_Comm_free(MPI_Comm *comm);
```

Remember to free your handles after they are no longer needed, it is always a good practice (like with allocatable arrays)

Intercommunicators

Intercommunicators are associated with 2 groups of disjoint processes.

Intercommunicators are associated with a remote group and a local group.

The target process (destination for send, source for receive) is its rank in the remote group

A communicator is either intra or inter, never both

Virtual topology

Virtual Topology

Topology:

- Extra, optional attribute that can be given to an intra-communicator; topologies cannot be added to inter-communicators.
- Can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

A process group in MPI is a collection of n processes:

- Each process in the group is assigned a rank between 0 and n-1.
- In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used).

Virtual Topology

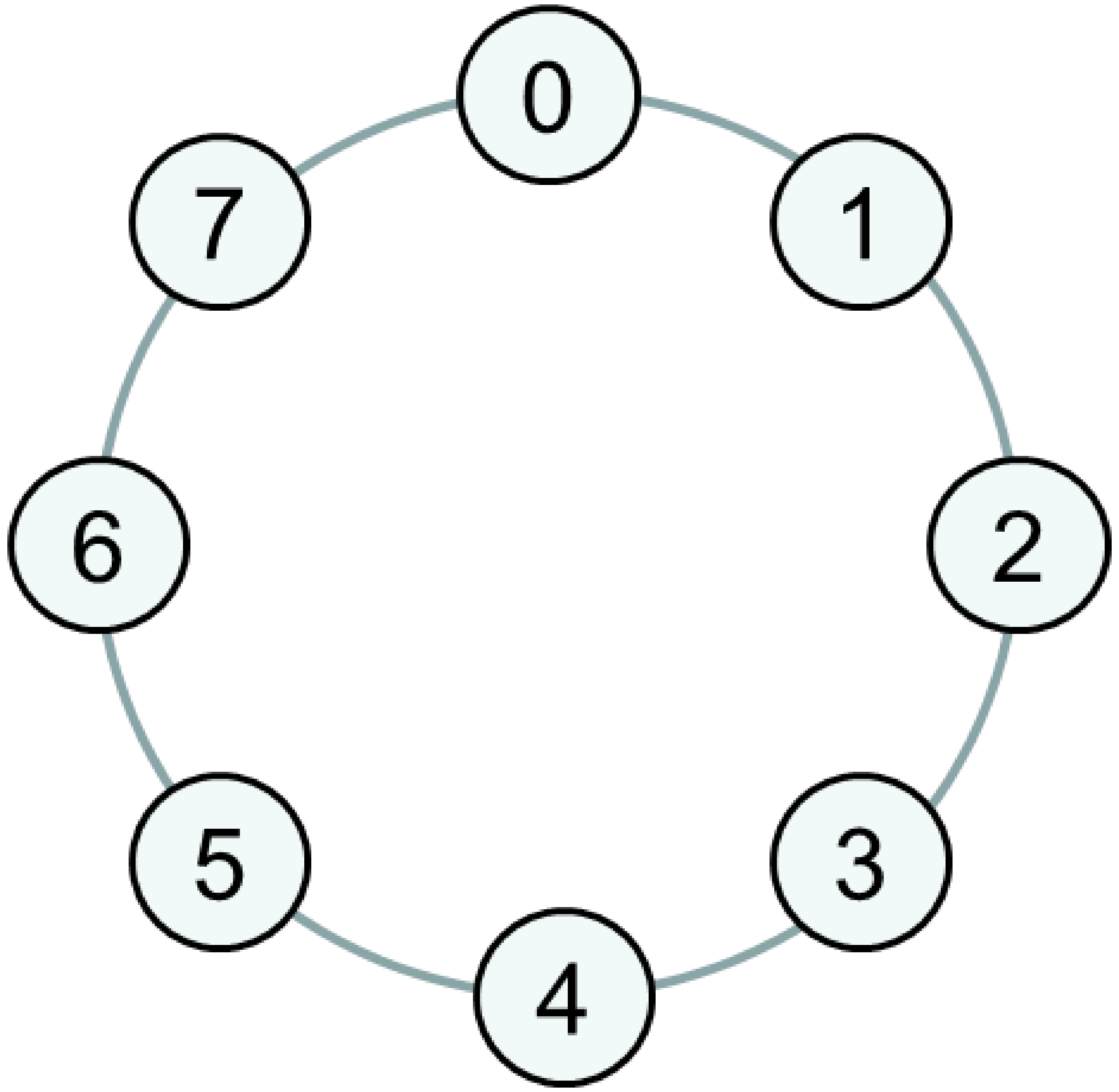
Virtual Topology:

- Logical process arrangement in topological patterns such as 2D or 3D grid; more generally, the logical process arrangement is described by a graph.

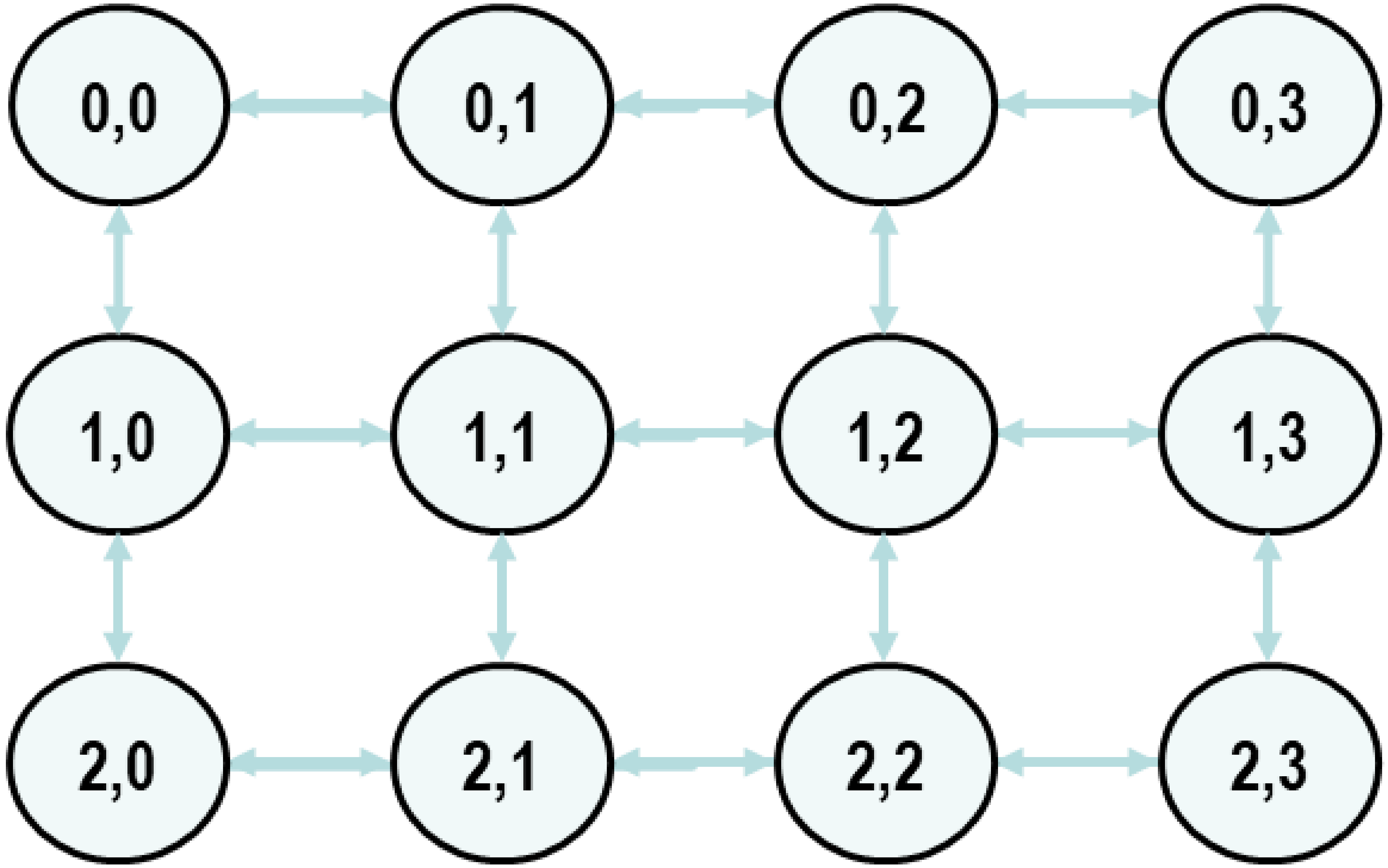
Virtual process topology .vs. topology of the underlying, physical hardware:

- Virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine.
- The description of the virtual topology depends only on the application, and is machine-independent.

Examples



RING

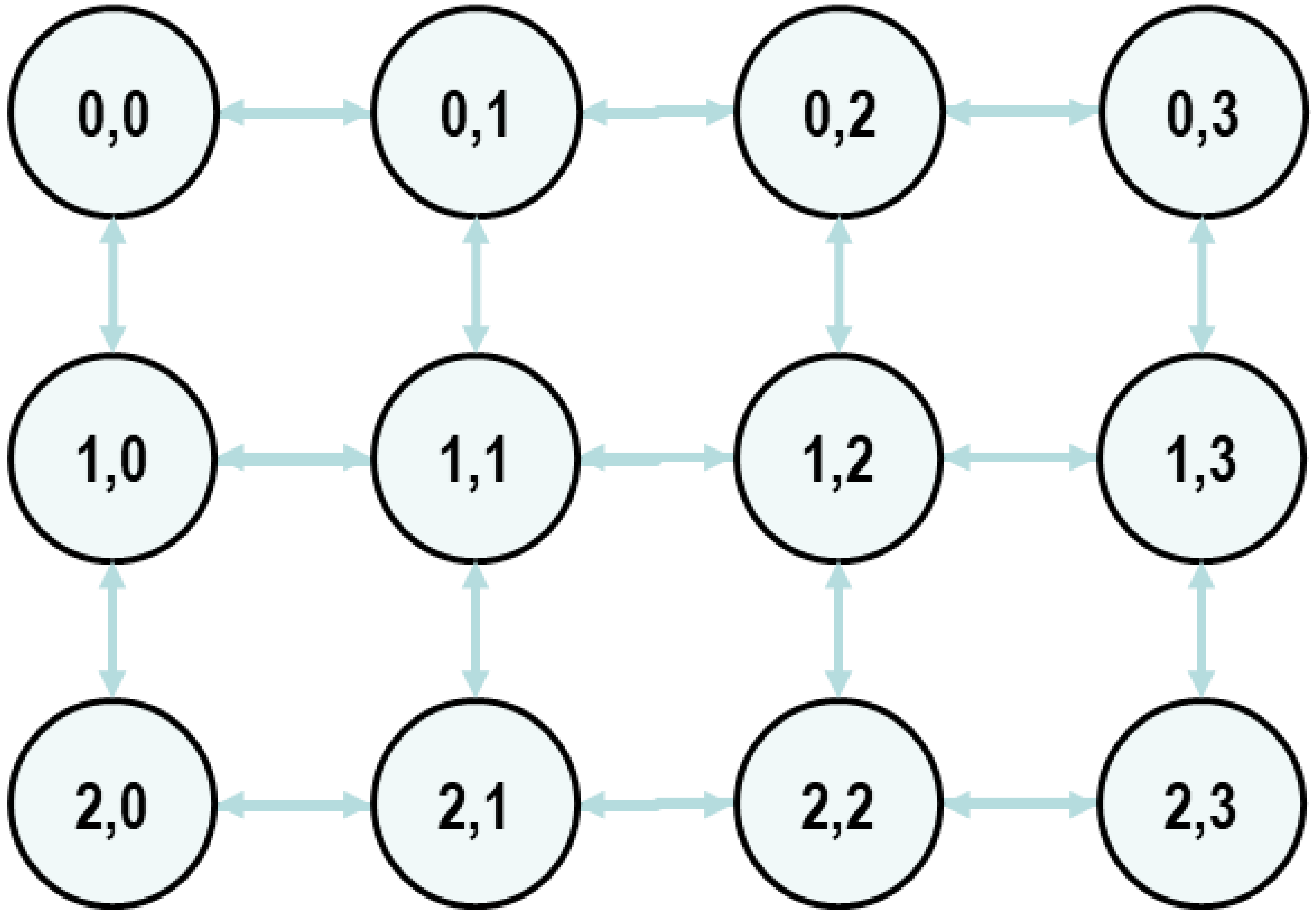


2D-GRID

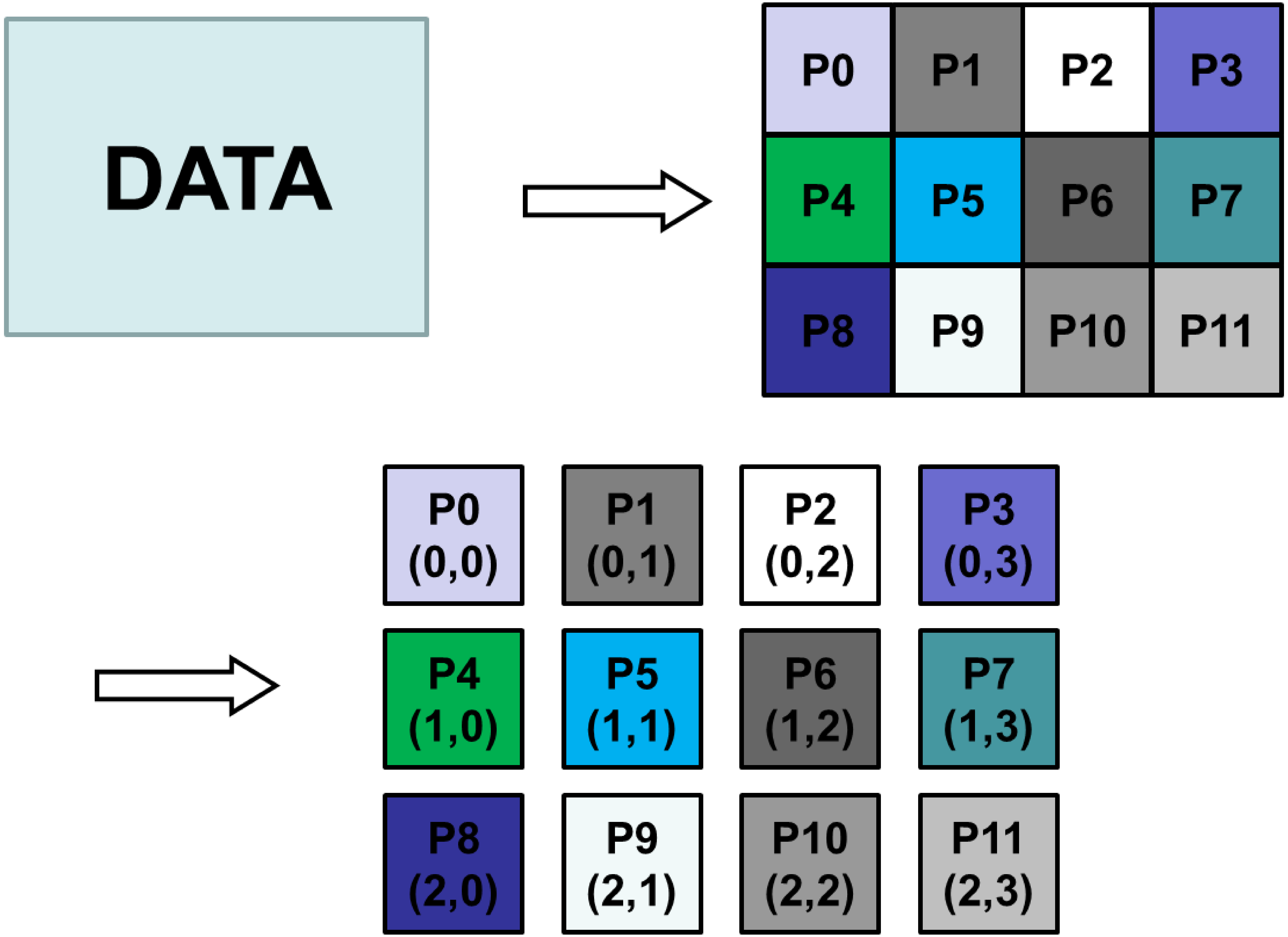
The cartesian communicator

Cartesian topology

A grid of processes is easily described with a **cartesian topology**:
- Each process can be identified by **cartesian coordinates**
- **Periodicity** can be selected for each direction
- Communications are performed along **grid dimensions** only



Example: 2D domain decomposition



Cartesian constructor

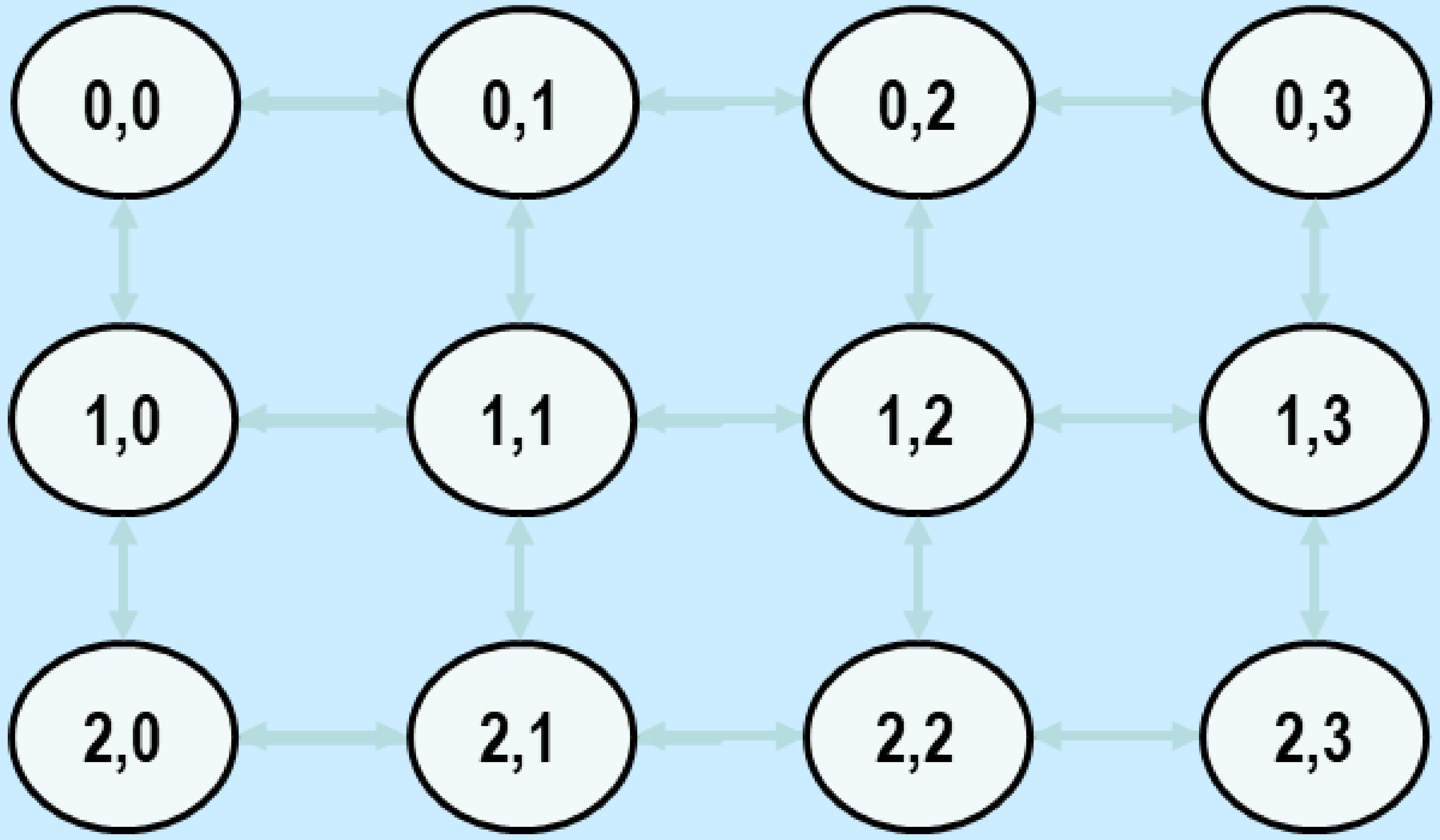
```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int
 dims[], const int periods[], int reorder, MPI_Comm *comm_cart)
```

comm_old	input communicator
ndims	number of dimensions of Cartesian grid
dims	integer array of size ndims specifying the number of processes in each dimension
periods	integer array of size ndims specifying whether the grid is periodic (1) or not (0) in each dimension
reorder	ranking may be reordered (1) or not (0)
comm_cart	communicator with new Cartesian topology

- Returns a handle to a new communicator to which the Cartesian topology information is attached.
- Reorder:
  - 0 (false): the rank of each process in the new group is identical to its rank in the old group.
  - 1 (true): the processes may be reordered, possibly so as to choose a good embedding of the virtual topology onto physical machine.
- If cart has less processes than the starting communicator, leftover processes have MPI_COMM_NULL as return value

Example

```
#include <mpi.h>

int main(int argc, char *argv[])
{

  MPI_Comm cart_comm;
  int dim[] = {4, 3};
  int period[] = {1, 0};
  int reorder = 0;

  MPI_Init(&argc, &argv);

  MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &cart_comm);
  ...
}
```



MPI_Dims_create

```
int MPI_Dims_create(int nodes, int ndims, int dims)
```

nnodes	number of nodes in a grid
ndims	number of Cartesian dimensions
dims	integer array of size ndims specifying the number of nodes in each dimension (output)

- Helps user to select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user
- If *dims[i]* is set to a positive number, the routine will not modify the number of nodes in that *i* dimension
- Negative value of *dims[i]* are erroneous

IN/OUT of dims

```
int MPI_Dims_create(int nodes, int ndims, int dims)
```

nnodes number of nodes in a grid
ndims number of Cartesian dimensions
dims integer array of size ndims specifying the number of nodes in each dimension (output)

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, 2, dims)	erroneous call

Using MPI_Dims_create

```
int dim[3], period[3], reorder, nprocs;
MPI_Comm cube_comm;
…
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

dim[0]=0; /* Let MPI arrange */
dim[1]=0; /* Let MPI arrange */
dim[2]=3; /* I want exactly 3 planes */

MPI_Dims_create(nprocs, 3, dim);

if (dim[0]*dim[1]*dim[2] < nprocs) {
  printf("Warning: some processes are not in use!\n");

period=(1,1,0);
reorder=0;

MPI_Cart_create(MPI_COMM_WORLD, 3, dim, period, reorder, &cube_comm);
…
```

From coordinate to rank

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[],
    int *rank)
```

comm communicator with Cartesian structure

coords integer array (of size ndims) specifying the Cartesian coordinates of a process

rank rank of specified process

- Translation of the logical process coordinates to process ranks as they are used by the point-to-point routines
- If dimension *i* is periodic, when *i*-th coordinate is out of range, it is shifted back to the interval 0 < *coords[i]* < *dims[i]* automatically
- Out-of-range coordinates are erroneous for non-periodic dimensions

From rank to coordinate

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int
 maxdim, int coords[])
```

comm communicator with Cartesian structure
rank rank of process within group of comm
maxdim length of vector “coords” in the calling program
coords integer array (of size ndims) specifying the Cartesian coordinates of specified process

- For each MPI process in Cartesian communicator, the coordinate whitin the cartesian topology are returned

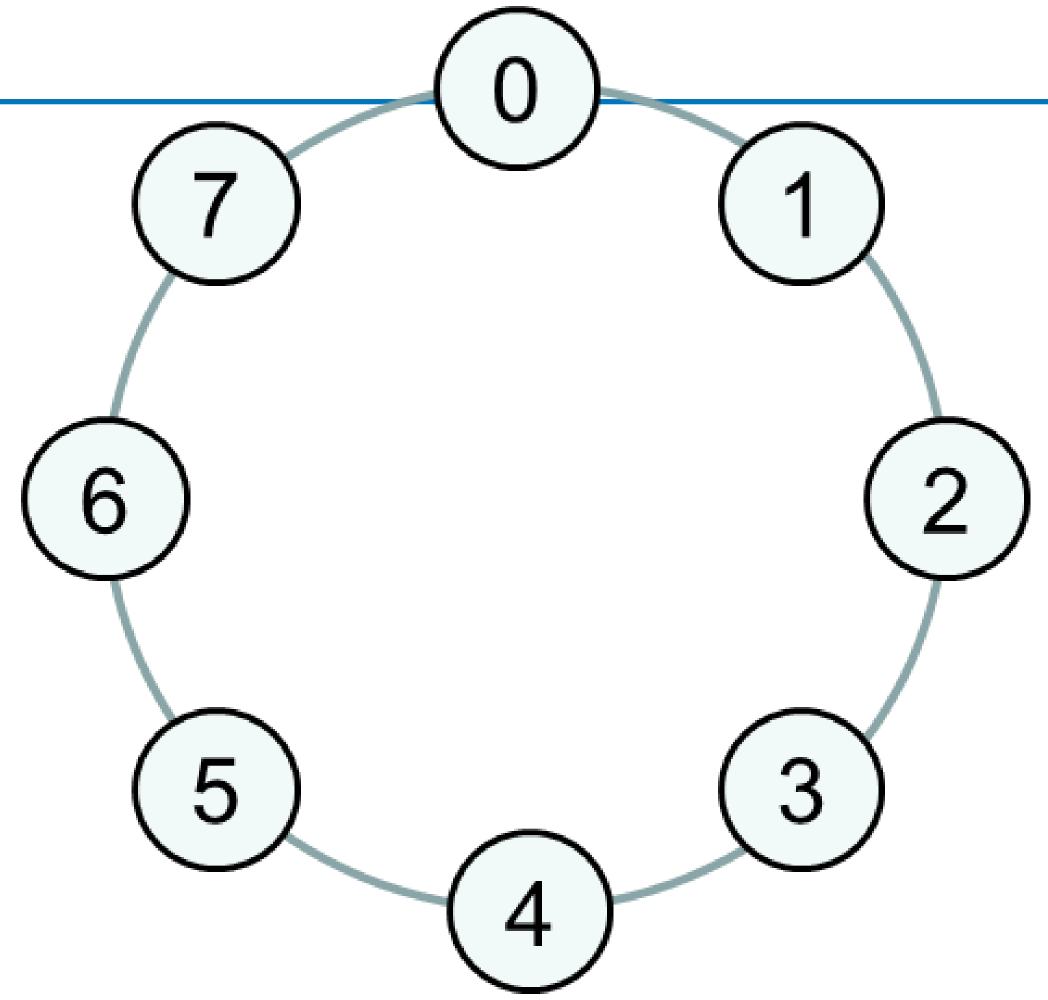
Mapping of coordinates

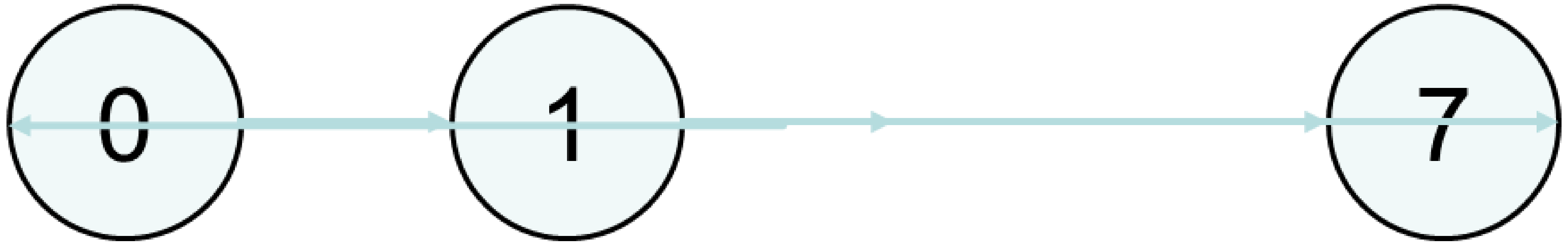
```
int cart_rank;
MPI_Comm_rank(cart_comm, &cart_rank);

int coords[2];
MPI_Cart_coords(cart_comm, cart_rank, 2, coords);

// set linear boundary values on top/left-hand domain
if (coords[0] == 0 || coords[1] == 0) {
  SetBoundary( linear(min, max), domain);
}

// set sinusoidal boundary values along lower domain
if (coords[0] == dim[0]) {
  SetBoundary( sinusoid(), domain);
}

// set polynomial boundary values along right-hand of domain
if (coords[1] == dim[1]) {
  SetBoundary( polynomial(order, params), domain);
}
```

Cartesian shift: a 1-D topology

Circular shift is another tipical MPI communication pattern:
- each process communicates only with its neighbours along one direction
- periodic boundary conditions can be set for letting first and last processes participate in the communication

such a pattern is nothing more than a **1D cartesian grid topology** with optional periodicity





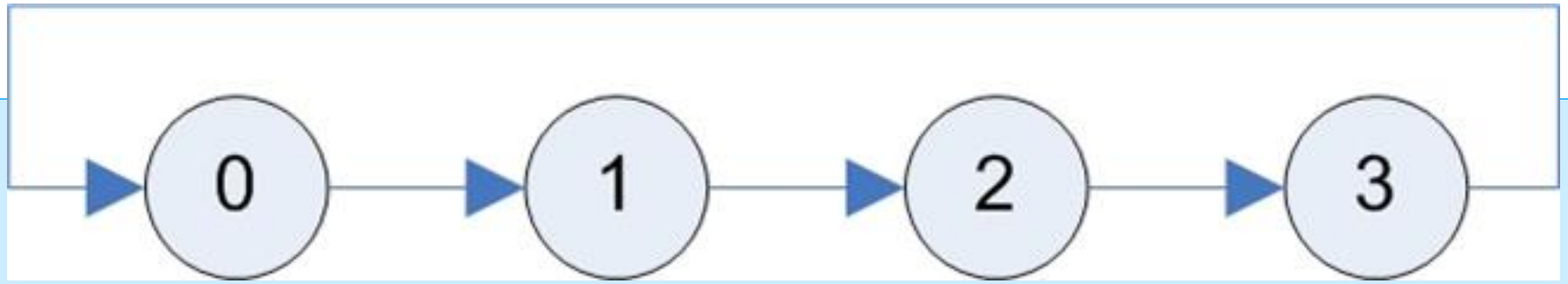
MPI_Cart_shift

```
int MPI_Cart_create(MPI_Comm comm, int direction, int disp,
    int *rank_source, int *rank_dest);
```

comm	input communicator
direction	coordinate dimension of shift
disp	>0: upwards shift; <0: downwards shift
rank_source	rank of source process
rank_dest	rank of destination process

- Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-o shift.
- In the case of an end-o shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.
- Provides the calling process the ranks of source and destination processes for an *MPI_Sendrecv* with respect to a specified coordinate direction and step size of the shift.

Example




```
int dim=nprocs;
int period=1;
int source, dest;
MPI_Status status;
MPI_Comm ring_comm;

MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &ring_comm);

MPI_Cart_shift(ring_comm, 0, 1, source, dest);

…
MPI_Sendrecv(&right_boundary, n, MPI_INT, dest, rtag, left_boundary, n, MPI_INT,
source, ltag, ring_comm, status);
…

MPI_Comm_free(ring_comm);
```

Partitioning of cartesian structure

It is often useful to partition a cartesian communicator into subgroups that form lower dimensional **cartesian subgrids**:

- new communicators are derived;
- lower dimensional communicators cannot communicate among them (unless inter-communicators are used).

MPI_Cart_sub

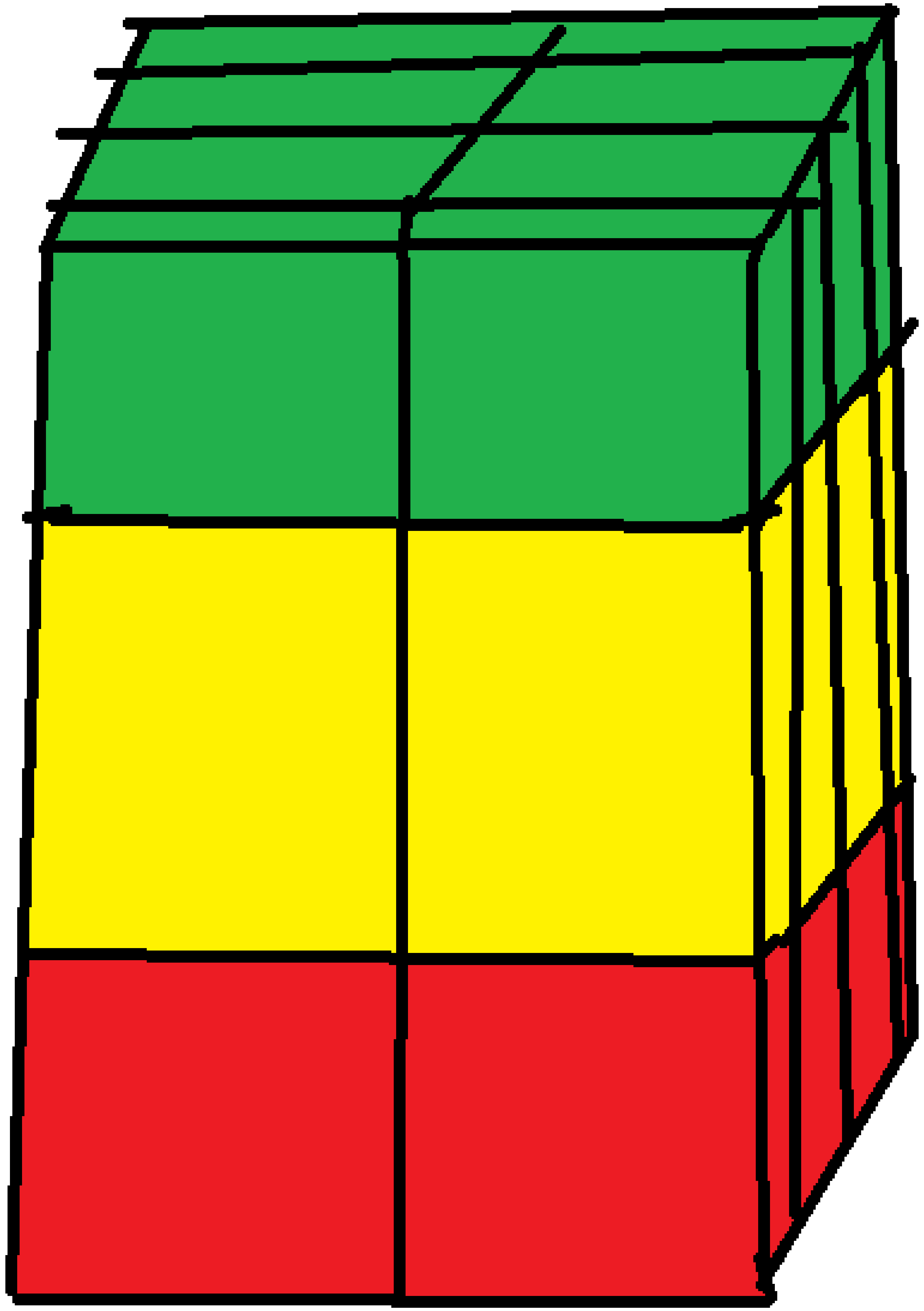
```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims,
     MPI_Comm newcomm);
```

comm	communicator with Cartesian structure
remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (1) or is dropped (0)
newcomm	communicator containing the subgrid that includes the calling process

```
int dim[] = {2, 3, 4};

int remain_dims[] = {1, 0, 1}; // 3 comm with 2x4 processes 2D grid
...
int remain_dims[] = {0, 0, 1}; // 6 comm with 4 processes 1D topology
```

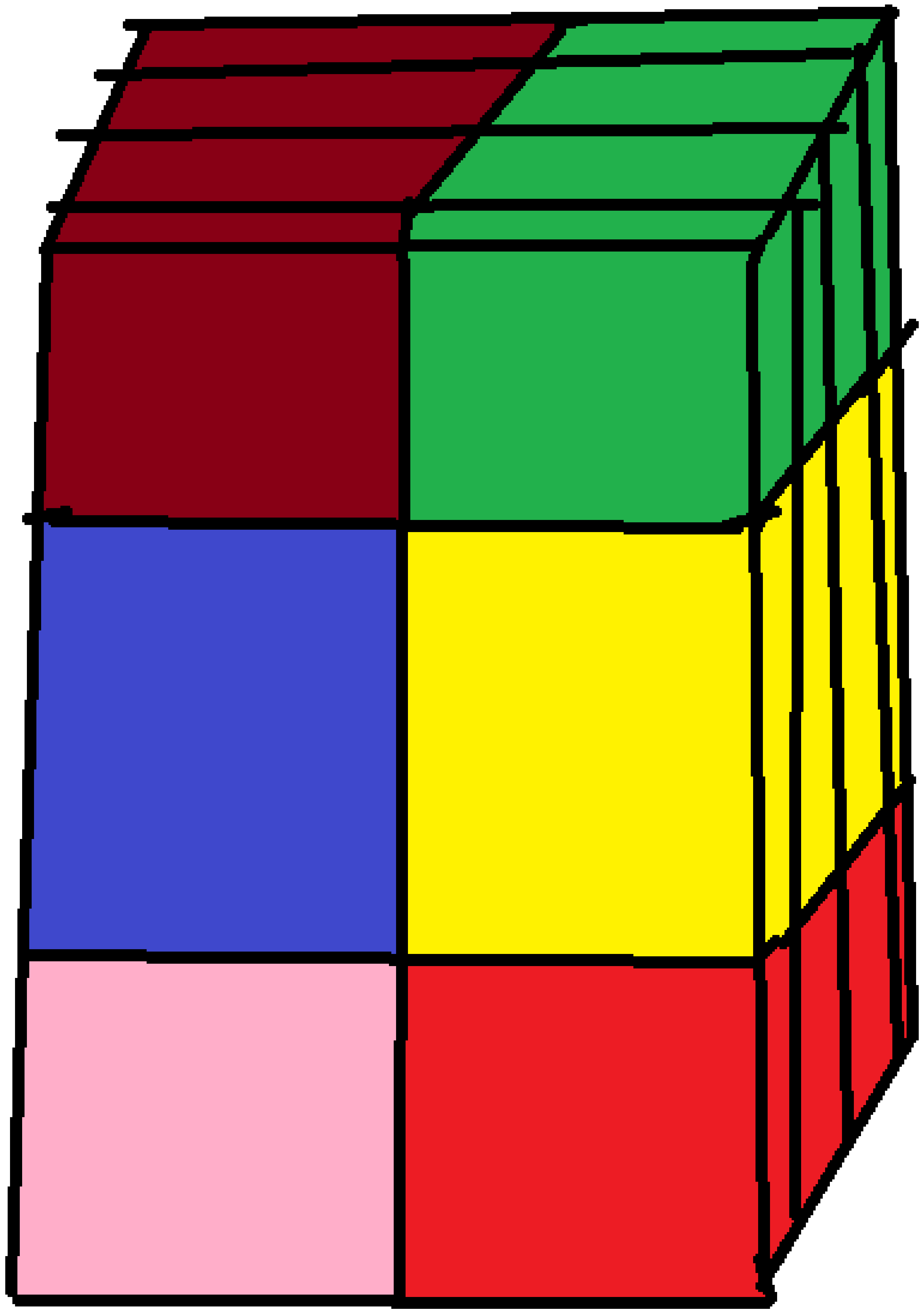
MPI_Cart_sub example

```
int dim[] = {2, 3, 4};

int remain_dims[] = {1, 0, 1};
// 3 comm with 2x4 processes
2D grid
```



MPI_Cart_sub example

```
int dim[] = {2, 3, 4};

int remain_dims[] = {0, 0, 1};
// 6 comm with 4 processes 1D
topology
```



Current MPI Standard to be found at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
Credits to A. Emerson and many other colleagues from CINECA for the slides