ICHEC
Irish Centre for High-End Computing

PRACE Course: Intermediate MPI

9-11 November 2022

MPI Virtual Topologies

Topologies

- MPI process topologies allow for simple referencing scheme of processes.
  - Two main types: Cartesian and graph topologies
  - Process topology defines a new communicator

- MPI topologies are virtual – hence virtual topologies
  - Not necessarily related to the physical structure of the computer (connections between cores, chips, and nodes in the hardware)
  - Process mapping more natural only to the programmer
  - It can still be exploited by the system

Topologies

- Simplifies management of communication relationships
- Usually no performance benefits
  - BUT code more compact and readable as a result
  - Can allow MPI to optimize communications
    > such as adapting communication buffer resources
    > renumber processes according to the network/memory.

- The communication pattern of a set of processes can be represented by a graph.
  - the nodes represent processes, and
  - the edges connect processes that communicate with each other.
  - edge weights provide more information

*Christoph Niethammer and Rolf Rabenseifner, Topology aware Cartesian grid mapping with MPI, EuroMPI 2018.
*Christoph Niethammer, Rolf Rabenseifner, An MPI interface for application and hardware aware cartesian topology optimization, EuroMPI 2019

Virtual Topology

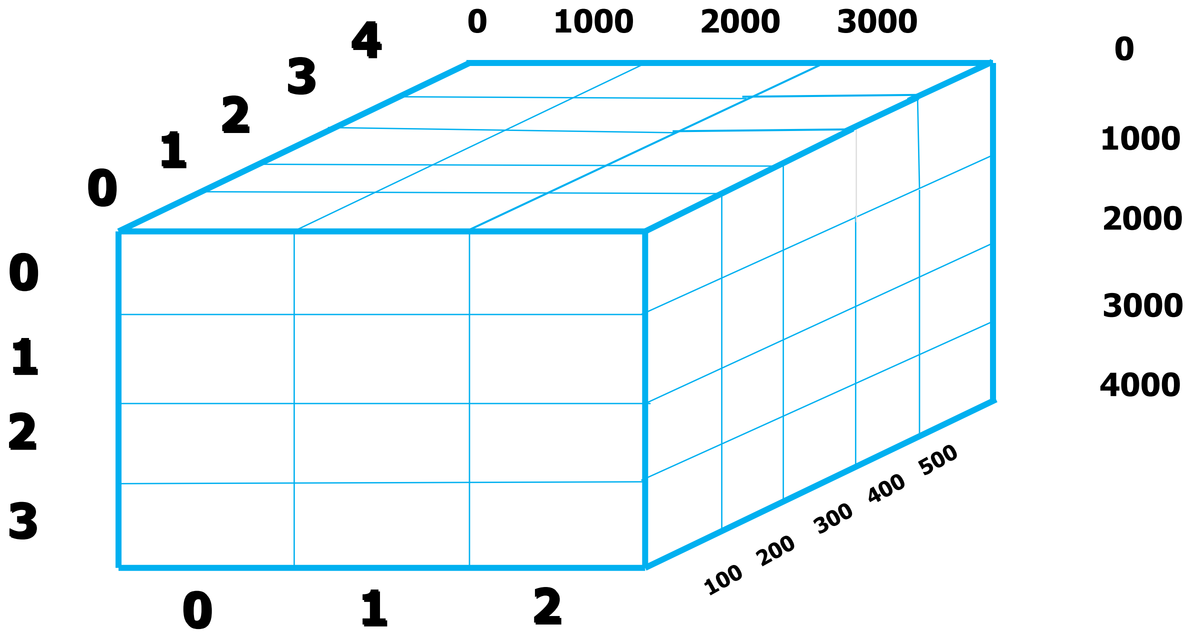
- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - from process ranks (old communicator), to those based on the topology naming scheme,
  - and vice versa.

- Naming scheme to fit the communication pattern
  - Array decomposition: handled by the application program
  - Process Coordinates: handled with virtual Cartesian topologies

- Allocation of processes to particular parts of a grid.

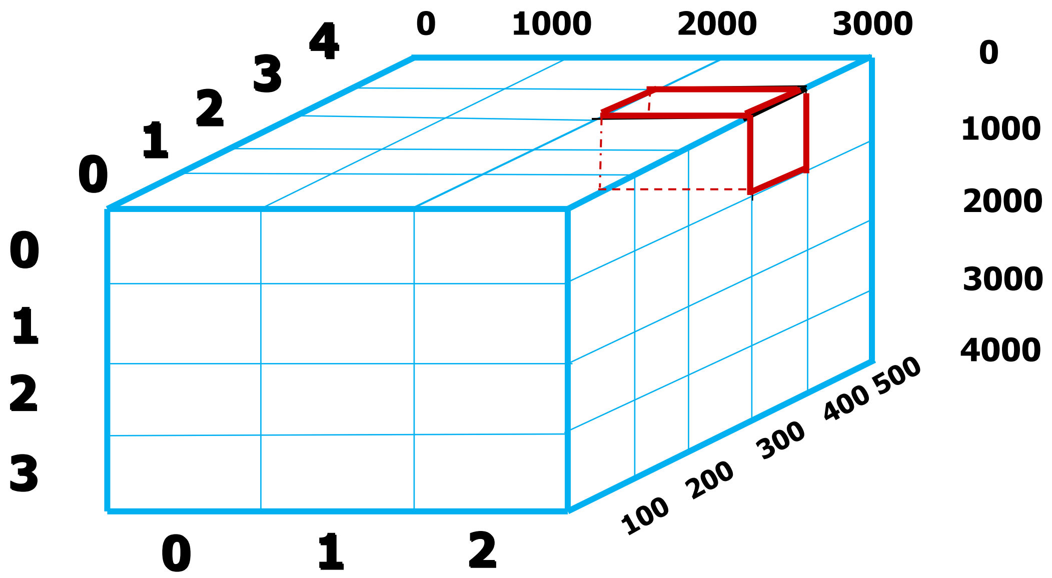
Virtual Topology

- Global array: A(1:3000, 1:4000, 1:500) = 6•109 units
- On 3 x 4 x 5 = 60 processors
- Process coordinates = (0:2), (0:3), (0:4)



Virtual Topology

- ic_0=2, ic_1=0, ic_2= 3 → (rank=43)
- A(2001:3000, 1:1000, 301:400) = 1•10^8 units



Topology Types

1. Cartesian Topologies – `MPI_Cart_create()`
   - regular communications patterns represented
   - create a virtual Cartesian process grid: 1d, 2d or higher
   - Specify the grid dimensions: number of rows and number of columns
   - each process is connected to its neighbor in a virtual grid
   - boundaries can be cyclic, or not
   - processes are identified by Cartesian coordinates
   - communication between any two processes is still allowed.

   - Applications: Matrix Computations, PDE Simulations: Heat equation

Topology Types

2. Graph Topologies – `MPI_Graph_create()`
   - irregular communications patterns represented
   - General graphs
   - No scalable to large communicators
   - `MPI_Dist_graph_create()`: Distributed graph topology.

     > doesn’t require the complete graph to be specified at each process

     > adjacent and general interface

   - Applications: 2d 5pt stencil Poisson, 2d 9pt sentencil Game of Life

William Group et al., Using Advanced MPI

Topology Functions

- MPI topologies: `MPI_CART, MPI_GRAPH, MPI_DIST_GRAPH`
- When is not defined: `MPI_UNDEFINED`

- `MPI_Topo_test()` returns the type of topology that is assigned to a communicator.

- Retrieve topology information: `MPI_Cart_get(), MPI_Cartdim_get(),MPI_Graph_get(), MPI_Cart_rank(), MPI_Cart_coords()`

- Constructors: `MPI_Cart_create MPI_Graph_create() MPI_Dist_graph_create()`

Cartesian Virtual Topology

- The number of dimensions (d)
- The size of each dimensions (s_1, s_2, …, s_d))
- Number of processes $s_1 * s_2 * \ldots * s_d$

- Process coordinates begin their numbering at 0
- Row-major numbering is always used for the processes

- Ex: d=2, $s_1=2$, $s_2=2$
  coord (0,0): rank 0
  coord (0,1): rank 1
  coord (1,0): rank 2
  coord (1,1): rank 3

(0,0)	(1,0)
(0,1)	(1,1)

2d Cartesian decomposition

Cartesian Virtual Topology

- C:

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

- Fortran:

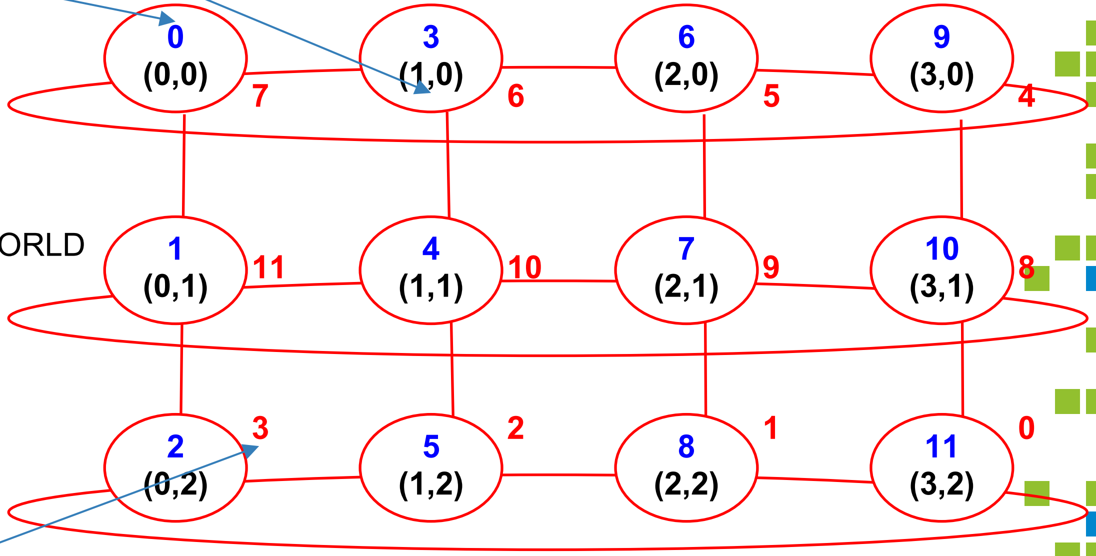
```
MPI_CART_CREATE(comm_old,ndims,dims,periods,reorder,,
    comm_cart, ierror)
    TYPE(MPI_Comm) :: comm_old, comm_cart
     INTEGER :: ndims, dims(*); INTEGER, OPTIONAL :: ierror
      LOGICAL :: periods(*), reorder
```

- Creates a new communicator newcomm from oldcomm, that represents an ndim dimensional grid with sizes dims.
- periodic in coordinate direction i if periodic[i] is true.
- Ranks are reordered (to better match the physical topology) if reorder is true
- `MPI_Cart_get()` to find the neighbours
- `MPI_Cart_create_weighted()` from MPI-4.1 – Hardware topology aware

Example – A 2-dimensional Cylinder

- Ranks and Cartesian process coordinates in comm_cart
- 2d torus of 12 processes in 4x3 grid

comm_old = MPI_COMM_WORLD
ndims = 2
dims = (4, 3)
periods = (1, 0)
reorder = 1



- Ranks in comm and comm_cart may differ, if reorder = 1 or .TRUE.
- This reordering can allow MPI to optimize communications

Cartesian Mapping Functions

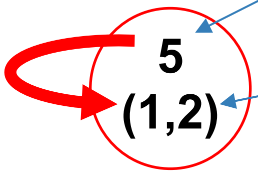
- C:

```
int MPI_Cart_coords(MPI_Comm comm_cart, int old_rank, int
                    maxdims, int *coords)
```

- Fortran:

```
MPI_CART_COORDS(comm_cart, old_rank, maxdims, coords, ierror)
    TYPE(MPI_Comm) :: comm_cart
    INTEGER :: old_rank, maxdims, cords(*)
    INTEGER, OPTIONAL :: ierror
```

- Mapping ranks to virtual process grid coordinates:



Cartesian Mapping Functions

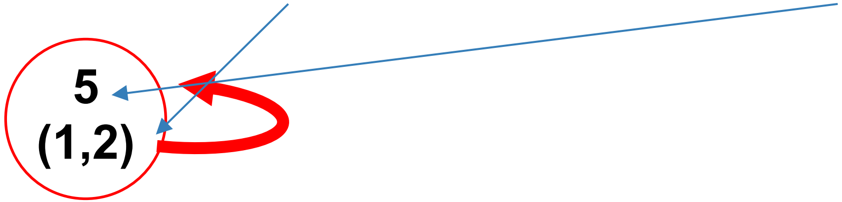
- C:

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords,
                                     int *rank)
```

- Fortran:

```
MPI_CART_RANK(comm_cart, coords, rank, ierror)
        TYPE(MPI_Comm) :: comm_cart
        INTEGER :: cords(*), rank
         INTEGER, OPTIONAL :: ierror
```

- Mapping process grid coordinates to ranks:



Example - Cartesian Mapping

- The processes in comm_cart are ranked in row-major order. Thus it may be advantageous to change the relative ranking of the processes in `MPI_COMM_WORLD`.

```
int coordinates[2];
int my_grid_rank;
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2, coordinates);
```

- Each process gets its own coordinates with:
  - If reorder = 1, call `MPI_Comm_rank()`
  - Call `MPI_Cart_coords()`
- Out-of-range coordinates are erroneous for non-periodic dimensions

Cartesian Mapping Functions

- C:
  `int MPI_Dims_create(int nnodes, int ndims, int *dims)`
- Fortran:

```
MPI_DIMS_CREATE(nnodes, ndims, dims, ierror)
     INTEGER :: nnodes, ndims, dims(*)
     INTEGER, OPTIONAL :: ierror
```

- Fill in the dims array such that the product of dims[i] for i=0 to ndim-1 equals nnodes.
- Any value of dims[i] that is 0 on input will be replaced; values that are > 0 will not be changed
- negative value of dims[i] are erroneous
- `MPI_Dims_create_weighted()` from MPI-4.1

Example: 2d cylinder / Cart_create

```
int myRank, uniSize, ierror;
int ndims, dims[2], periods[2], reorder;
int coords[2], coordRank;
MPI_Comm comm_cart;

ierror=MPI_Init(&argc,&argv);
ierror=MPI_Comm_size(MPI_COMM_WORLD,&uniSize);
ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

if(uniSize==12){
  ndims=2;
  dims[0]=4; dims[1]=3;
  periods[0]=1; periods[1]=0;
  reorder=1;
  ierror=MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
  if (myRank == 5){
      MPI_Cart_coords(comm_cart, myRank, ndims, coords);
      printf("Rank %d coordinates are %d %d\n", myRank, coords[0], coords[1]);
   }
   if(myRank==0){
      coords[0]=3; coords[1]=1;
      MPI_Cart_rank(comm_cart, coords, &coordRank);
      printf("The processor at position (%d, %d) has rank %d\n", coords[0], coords[1], coordRank);
   }
}
```

2d cylinder of 12 processes in a 4x3 cartesian grid

Run with 12 processes

Cartesian Mapping Functions

- C:

```
int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int
                   disp, int *rank_source, int *rank_dest)
```

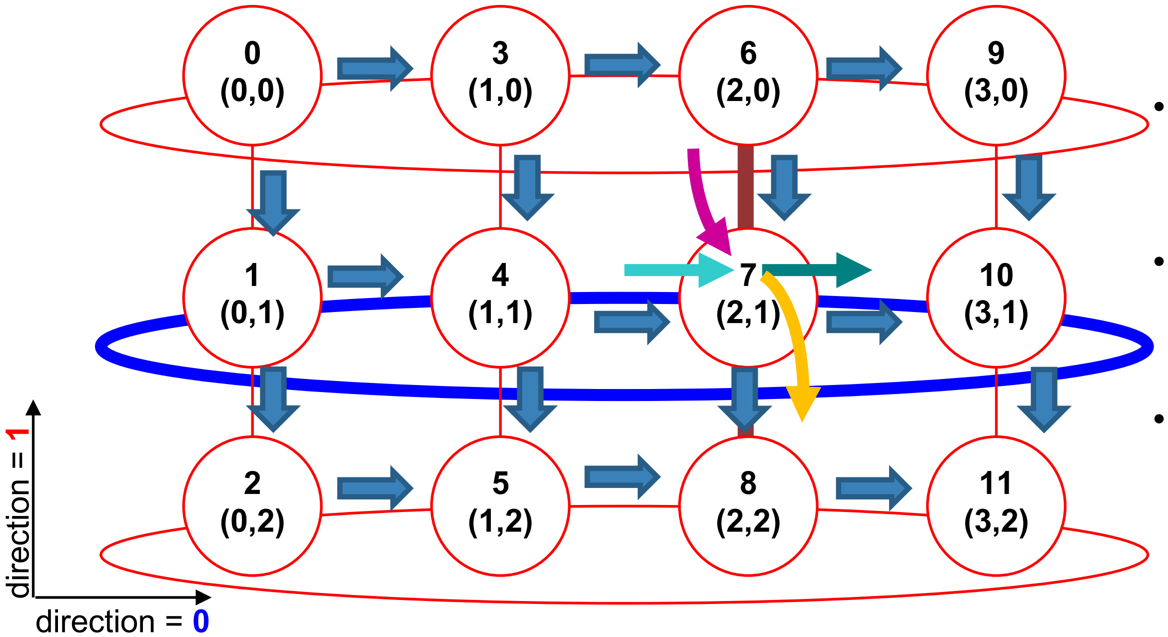
- Fortran:

```
MPI_CART_SHIFT(comm_cart, direction, disp, rank_source,
               rank_dest, ierror)
    INTEGER :: direction, disp, rank_source, rank_dest
    INTEGER, OPTIONAL :: ierror; Type(MPI_Comm) :: comm_cart
```

- Computing ranks of neighboring processes
- Returns the shifted source and destination ranks, given a shift direction and amount; MPI_PROC_NULL if there is no neighbor.
- direction[in]: coordinate dimension of shift
- displ[in]: displacement (> 0: upwards shift, < 0: downwards shift)

Example - MPI_Cart_shift

```
MPI_Cart_shift(cart, direction, displace, &rank_prev, &rank_next)
MPI_Cart_shift(cart, direction, displace, rank_prev, rank_next, ierror)
   example on             0          +1          4       10
   process rank=7         1          +1          6        8
```



- Shift by one in both directions.
- Find neighbours to the left and right along 0
- Find neighbours to the up and down along 1
- What are the rightand bottom neighbours of (3,2)?

Example – ring / shift / Sendrecv

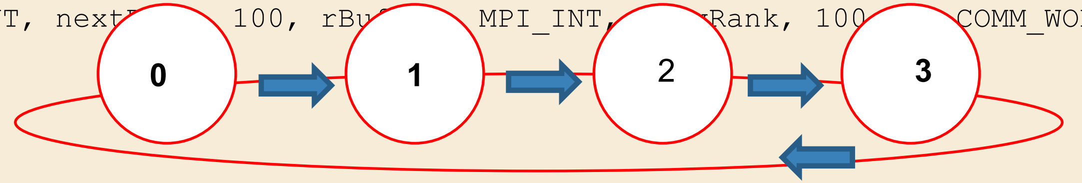
```
  int ndims, dims[1], periods[1], reorder;
  MPI_Comm comm_cart;
  int sendbuf, recvbuf;

  ndims=1;
  dims[0]=uniSize;
  periods[0]=1;
  reorder=1;
  ierror=MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
  ierror=MPI_Comm_rank(comm_cart, &commRank);

  ierror=MPI_Cart_shift(comm_cart, 0, 1, &prevRank, &nextRank);
  printf("Rank %d: rank prev = %d, rank next = %d\n", commRank, prevRank, nextRank);

  int sBuf[2] = {commRank, 0};
  int rBuf[2] = {-1, 0};

    while ( rBuf[0] != commRank ) {
      MPI_Sendrecv(sBuf, 2, MPI_INT, next      100, rBu      MPI_INT,      Rank, 100      COMM_WORLD,
&status);
        sBuf[1] += rBuf[0];
        sBuf[0] = rBuf[0];
    }
```

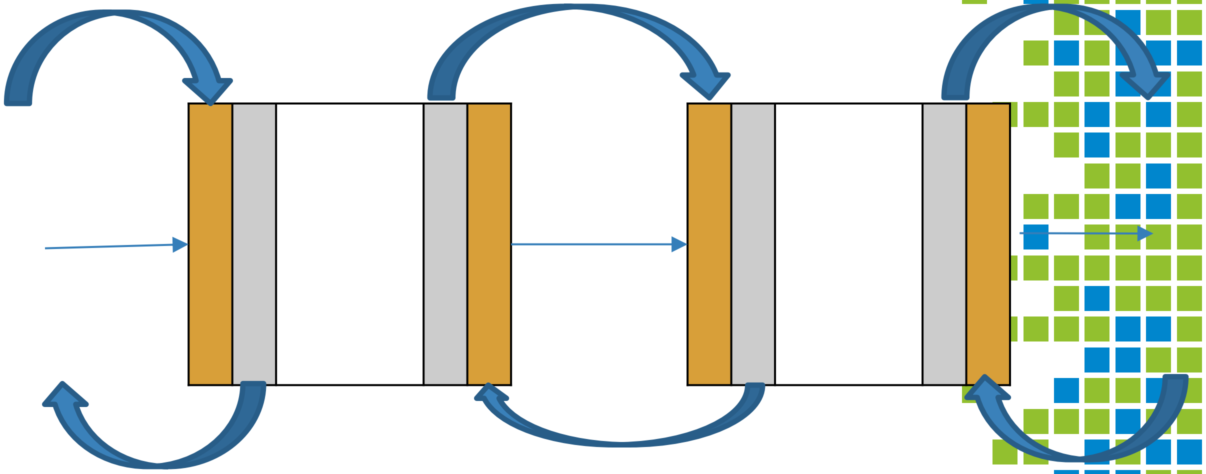


Halo Swapping

- Computational domain is split across processes.
- Each process computes over its local domain
- Each process needs to send and receive from its neighbours
- Neighbour data is stored in the ghost (halo) column
- Data is being shifted right from one process to another.

Halo exchange:

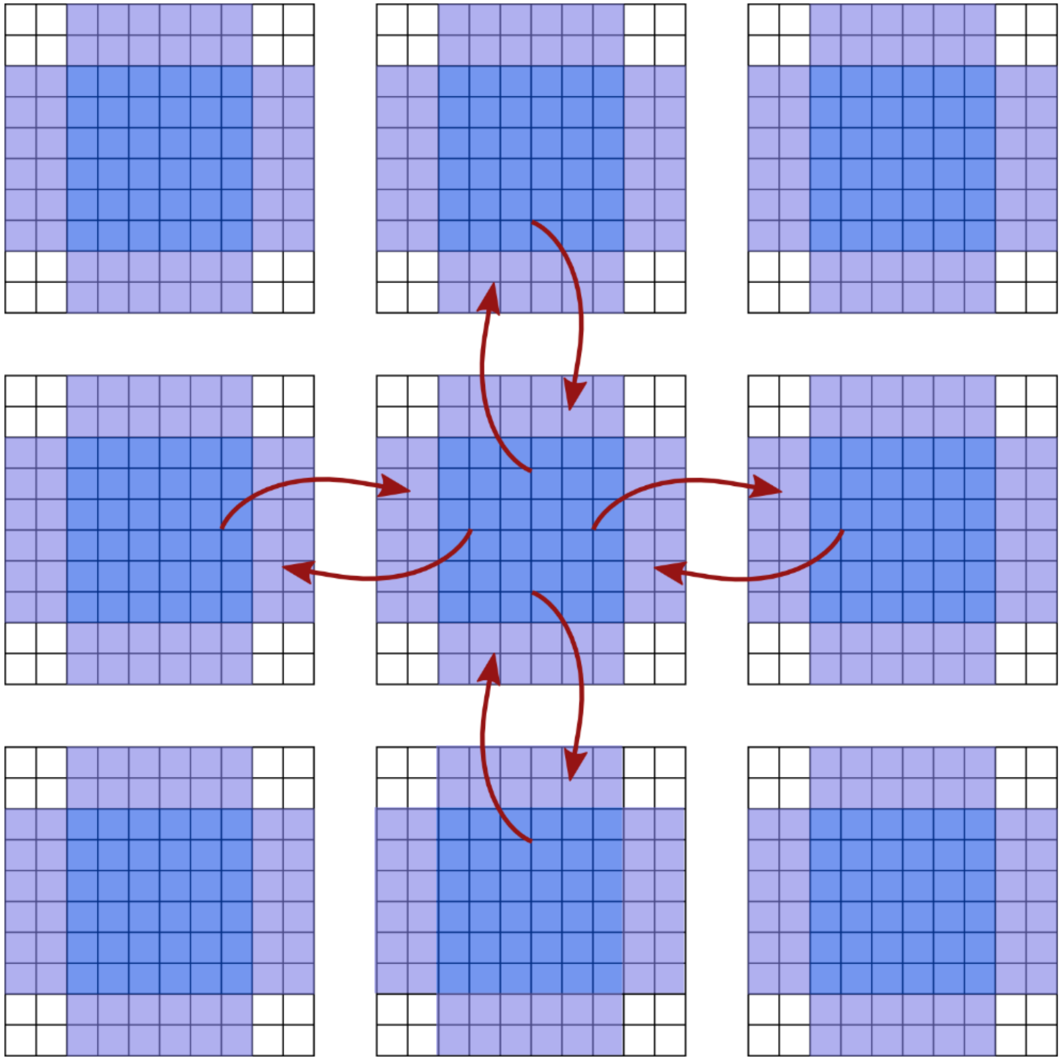
- Copy of the rightmost column from one process to the left ghost column of the process right to it.
- And viceversa



Halo exchange:

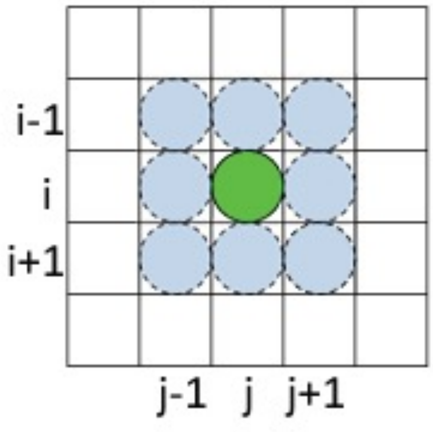
1. Horizontal swap in the direction east
2. Horizontal swap in the direction west
3. Vertical swap in the direction south
4. Vertical swap in the direction north

- Each implemented with a single MPI call - MPI_Sendrecv.



Use case: Conway's Game of Life

- n by n 2d array of cells
- Matrix values for each cell is initialised as 1(live) or 0 (dead)
- Each cell is surrounded by eight others
- At each time step, calculate each new cell state based on previous cell state



- Decompose the domain:
  - Divide domain left-right (break with vertical line)
  - Split domain into equally sized blocks n both i and j directions.
- Each process is assigned a single block to work on.
- How to send the data in ghost cells?

PARTITIONING OF CARTESIAN STRUCTURES

- C:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims,
                         MPI_Comm *comm_slice)
```

- Fortran:

```
MPI_CART_SUB(comm_cart, remain_dims, comm_slice, ierror)
    LOGICAL :: remain_dims(*)
     INTEGER, OPTIONAL :: ierror;
     Type(MPI_Comm) :: comm_cart, comm_slice
```

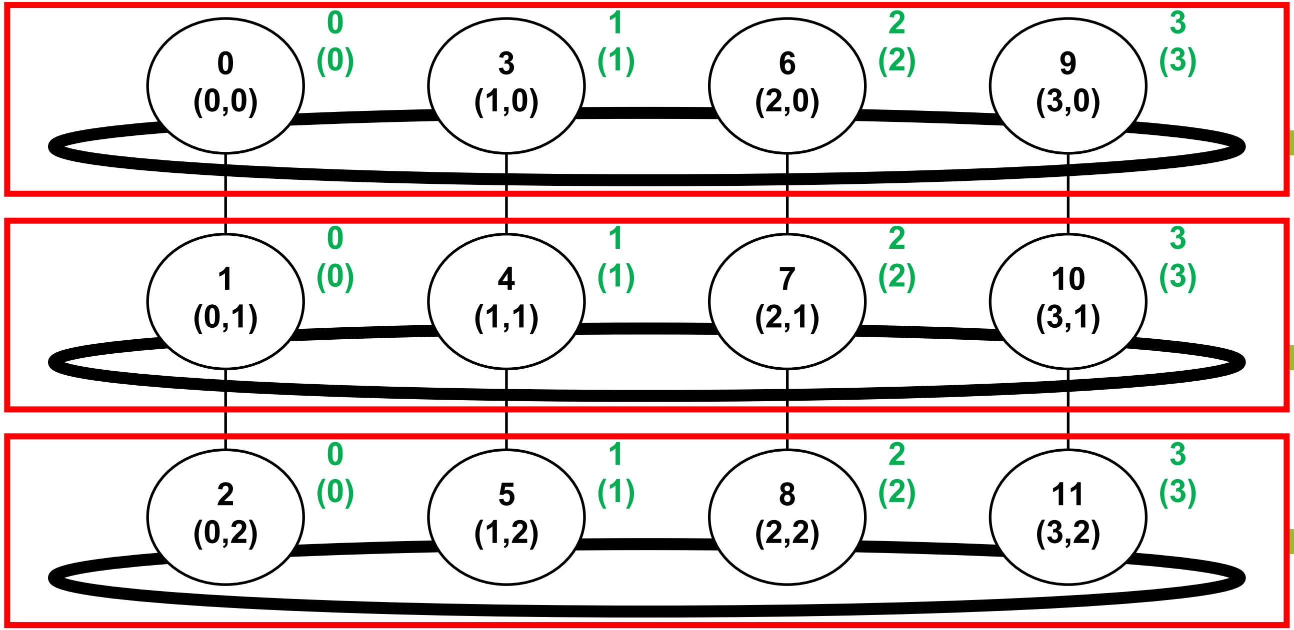
- creates new communicators for subgrids of up to (n-1) dimensions from an n-dimensional Cartesian grid.
- remain_dims[in]: the ith entry of remain_dims specifies whether the ith dimension is kept in the subgrid or is dropped
- newcomm[out]: communicator containing the subgrid that includes the calling process

PARTITIONING OF CARTESIAN STRUCTURES

- Ranks and Cartesian process coordinates in comm_slice

```
MPI_Cart_sub(comm_cart, remain_dims, comm_sub)

MPI_Cart_sub(comm_cart, remain_dims, comm_sub, ierror)
```



Example - MPI_Cart_sub

- Create communicators for the row of the grid
- Each new communicator consists of the processes obtained by fixing the row coordinates and letting the column coordinates vary.

```
int remain_dims[2];
MPI_Comm rowcomm;
remain_dims[0]=0;
remain_dims[1]=1;  // this dimension belongs to subgrid
MPI_Cart_sub(old_comm, remain_dims, &rowcomm);
```

- `MPI_Comm_split` is more general than `MPI_Cart_sub`
- `MPI_Comm_split` creates logical grid and is referred to by its linear rank number; `MPI_Cart_sub` creates cartesian grid and rank can be referred to by cartesian coordinates.

Neighbourhood Collectives

- Many applications and libraries exhibit sparse communication patterns
- Collective communications over topologies
- All communication only traditional collectives (except reduce) can be expressed as neighbourhood collectives (not recommended)
- Support for widely used patterns and better scalability

- 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)

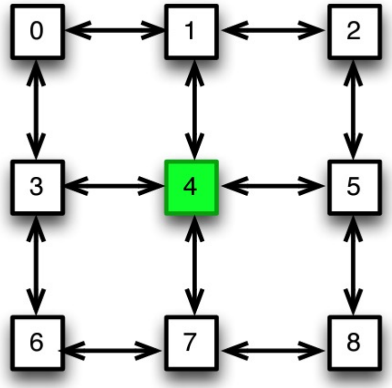
Neighbourhood Collectives

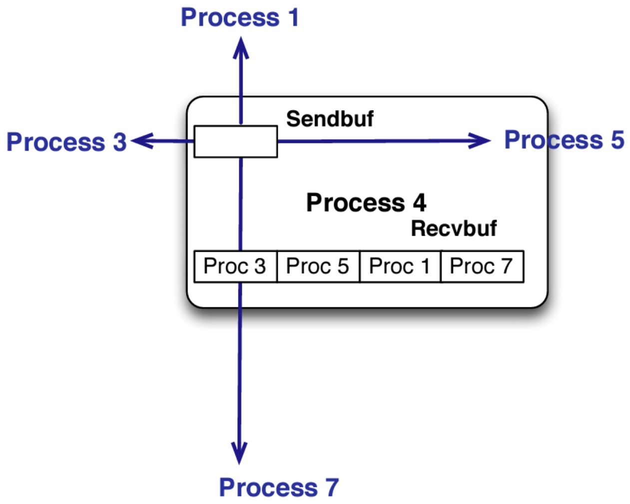
- Number of sources and destinations are equal to 2 * ndims dimensions
- The order of neighbors in buffers is in dimension order, and in each dimension first negative neighbor, and then positive neighbor
- For MPI_PROC_NULL, buffer not updated or communicated

- Main calls: `MPI_Neighbor_allgather()`, `MPI_Neighbor_alltoall()`, `MPI_Neighbor_alltoallw()`
- Full support for all nonblocking neighborhood collectives
  - Same collective invocation requirement
  - Matching will be done in order of the collective post for each collective

Neighbourhood Collectives

- MPI_[N|In]eighbor_allgather[v]
  - Send one piece of data to all neighbours
  - Gather one piece of data from each neighbour





- MPI_[N|In]eighbor_alltoall[v|w]
  - Send different data to each neighbour
  - Receive different data from each neighbour

A Preview of MPI 3.0: The Shape of Things to Come, OLCF

MPI_Neighbor_allgather

- C:

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

- Fortran:

```
MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER:: sendcount, recvcount
    TYPE(MPI_Datatype):: sendtype, recvtype
    TYPE(MPI_Comm):: comm
    INTEGER, OPTIONAL) :: ierror
```

- Same send buffer for each outgoing neighbour
- Contiguous chunks in receive buffer from each incoming neighbour
- Similar to MPI_gather where each process is the root on the neighborhood

MPI_Neighbor_alltoall

- C:

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

- Fortran:

```
 MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm, ierror)
     TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
     INTEGER:: sendcount, recvcount
     TYPE(MPI_Datatype):: sendtype, recvtype
     TYPE(MPI_Comm):: comm
     INTEGER, OPTIONAL) :: ierror
```

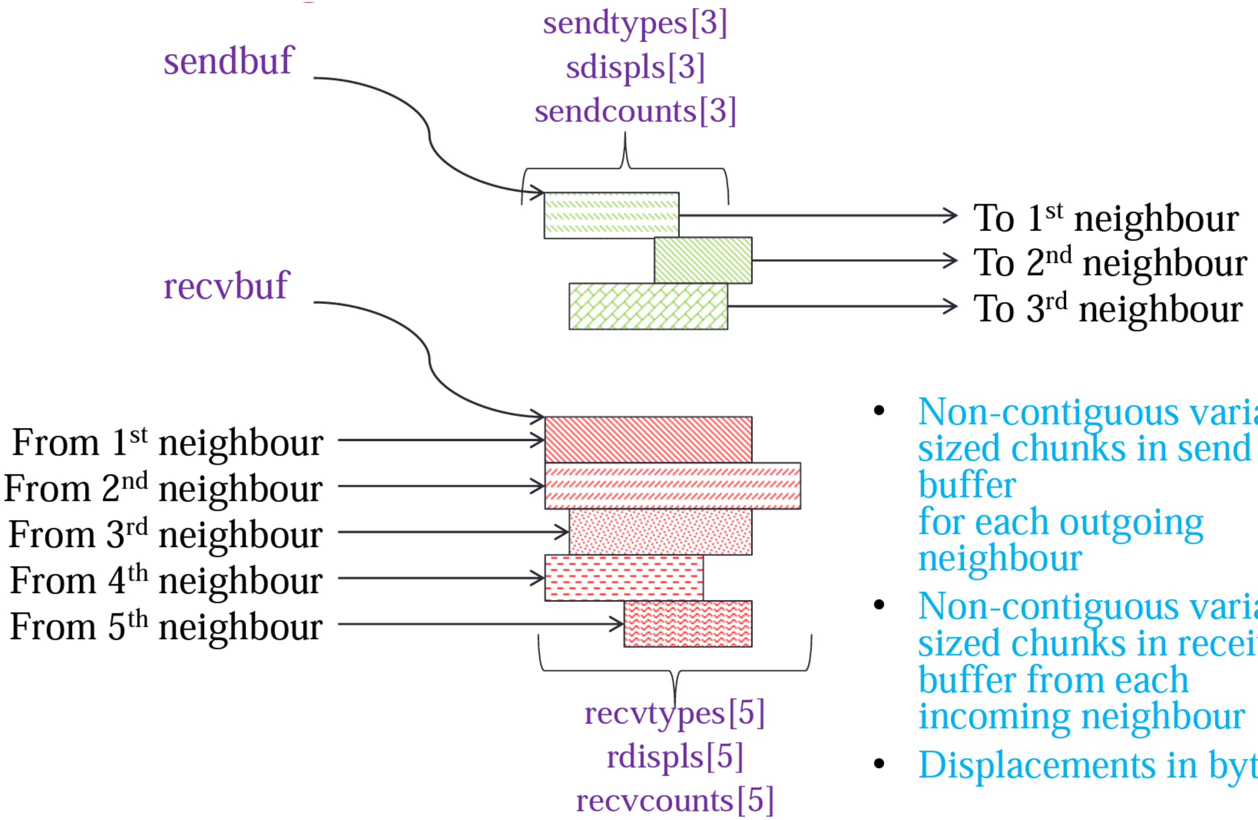
- Send a distinct data element to all neighbor process
- Receive a distinct data element from each of the neighbor
- Type signature of sendtype/recvtype must be same at the corresponding processes

Neighbourhood Collectives

- Vector Neighbourhood Collectives: `MPI_Neighbor_allgatherv()`, `MPI_Neighbor_alltoallv()`
  - specify different number of elements of the same type.
  - size and location specified as integer arrays: recvcounts[] and displs[]

- `MPI_Neighbor_alltoallw()`: Enables to specify different datatypes for each incoming or outgoing neighbour.
  - Non-contiguous variable-sized chunks in send buffer for each outgoing neighbour
  - Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour

MPI_Neighbor_alltoallw



- Non-contiguous variable-sized chunks in send buffer for each outgoing neighbour
- Non-contiguous variable-sized chunks in receive buffer from each incoming neighbour
- Displacements in bytes

Example – ring / shift / Alltoall

```
int uniSize, ierror, i, sum=0, commRank, prevRank, nextRank;
int ndims, dims[1], periods[1], reorder;
MPI_Comm comm_cart; MPI_Status recv_status, wait_status; MPI_Request request;
int sendbuf[2], recvbuf[2];

ierror=MPI_Init(&argc,&argv);
ierror=MPI_Comm_size(MPI_COMM_WORLD,&uniSize);

ndims=1;dims[0]=uniSize; periods[0]=1; reorder=1;
ierror=MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
ierror=MPI_Comm_rank(comm_cart, &commRank);

ierror=MPI_Cart_shift(comm_cart, 0, 1, &prevRank, &nextRank);
printf("Rank %d: rank prev = %d, rank next = %d\n", commRank, prevRank, nextRank);

sendbuf[1] = commRank;
for(i=0;i<uniSize;i++){
  ierror=MPI_Neighbor_alltoall(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT, comm_cart);
  sendbuf[1]=recvbuf[0];
  sum+=recvbuf[0];
}
```