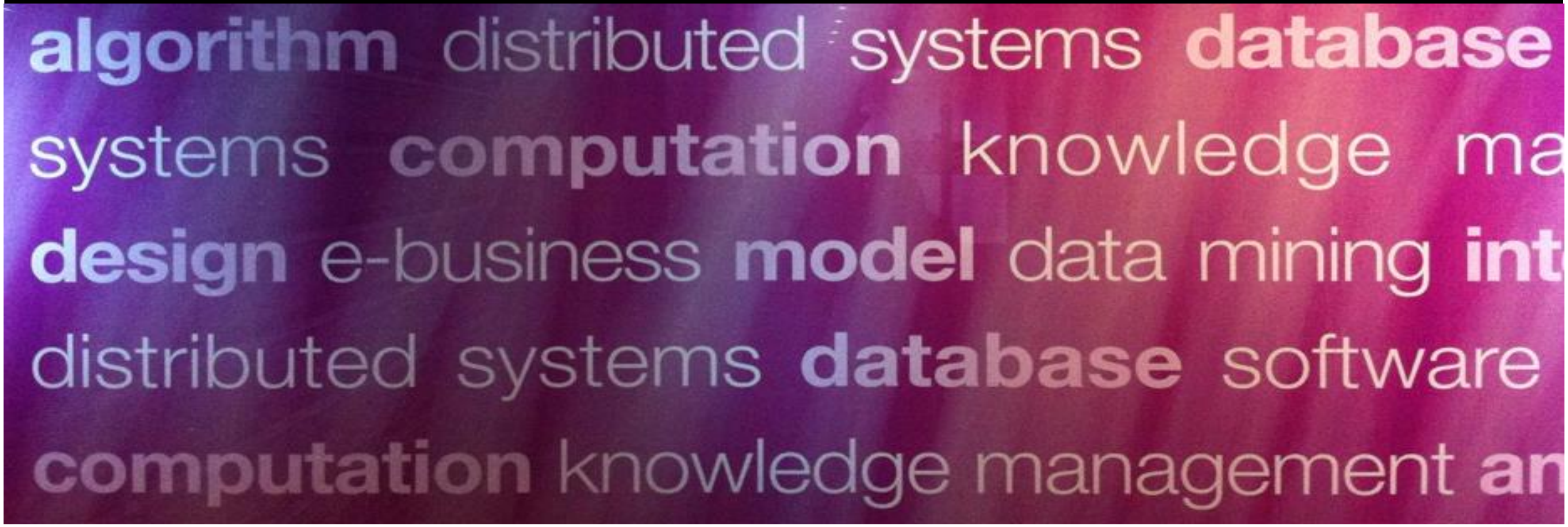
MONASH University

Information Technology

FIT3143 - LECTURE WEEK 9a

PARALLEL ALGORITHM DESIGN -
ADVANCED MPI TOPICS



Topic Overview

- Revisiting Collective Communications with MPI Scatter & Gather
- Introduction to MPI Virtual Topologies

A portion of the content in the following slides were adopted from:

a) Introduction to the
Message Passing Interface (MPI), Irish Centre for High-End Computing (ICHEC)
(www.ichec.ie)

Learning outcome(s) related to this topic

- Design and develop parallel algorithms for various parallel computing architectures (LO3)

Revisiting Collective Communications with MPI Scatter & Gather

Collective Communication

- Communications involving a group of processes.
- Must be called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Characteristics of Collective Communication

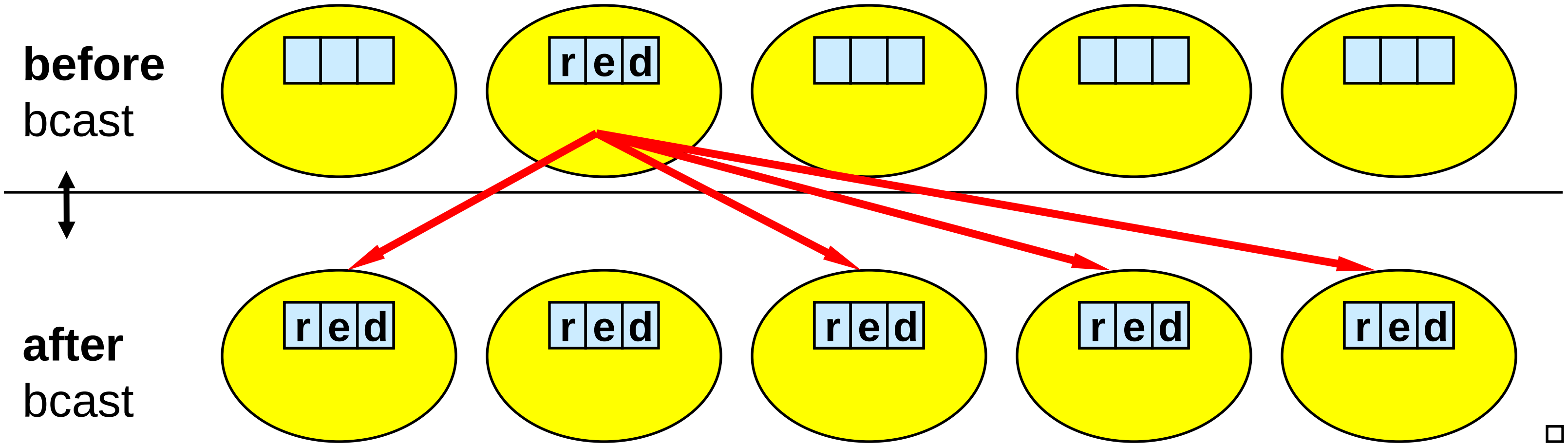
- Optimised Communication routines involving a group of processes
- Collective action over a communicator, i.e. all processes must call the collective routine.
- Synchronization may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must have exactly the same size as send buffers.

Barrier Synchronization

- C: int MPI_Barrier(MPI_Comm comm)
- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - a process cannot continue before it has the data that it needs.
  - if used for debugging:
    - please guarantee, that it is removed in production.

Broadcast

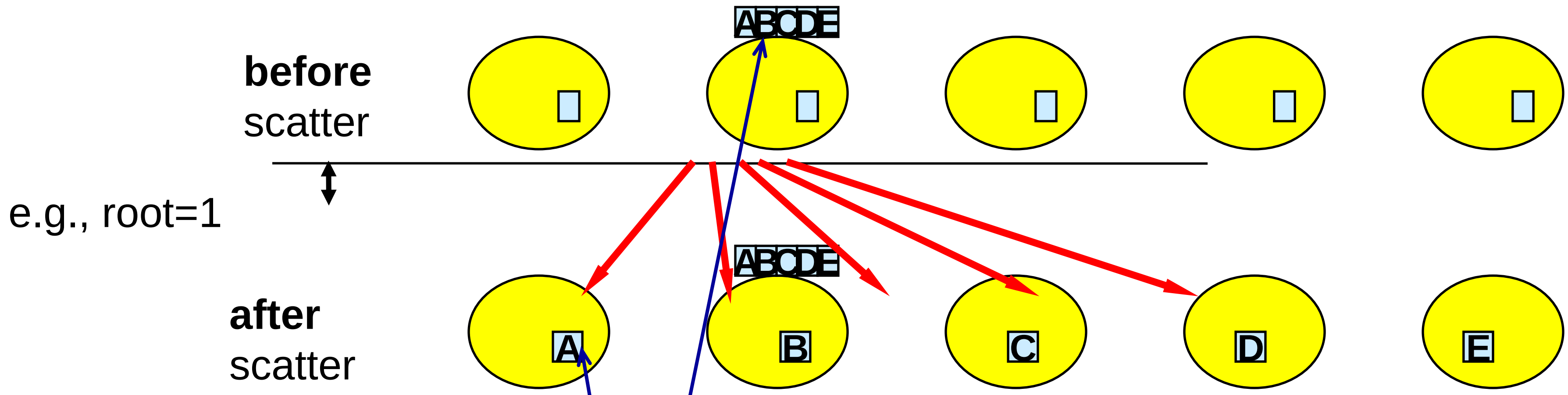
- C: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)



e.g., root=1

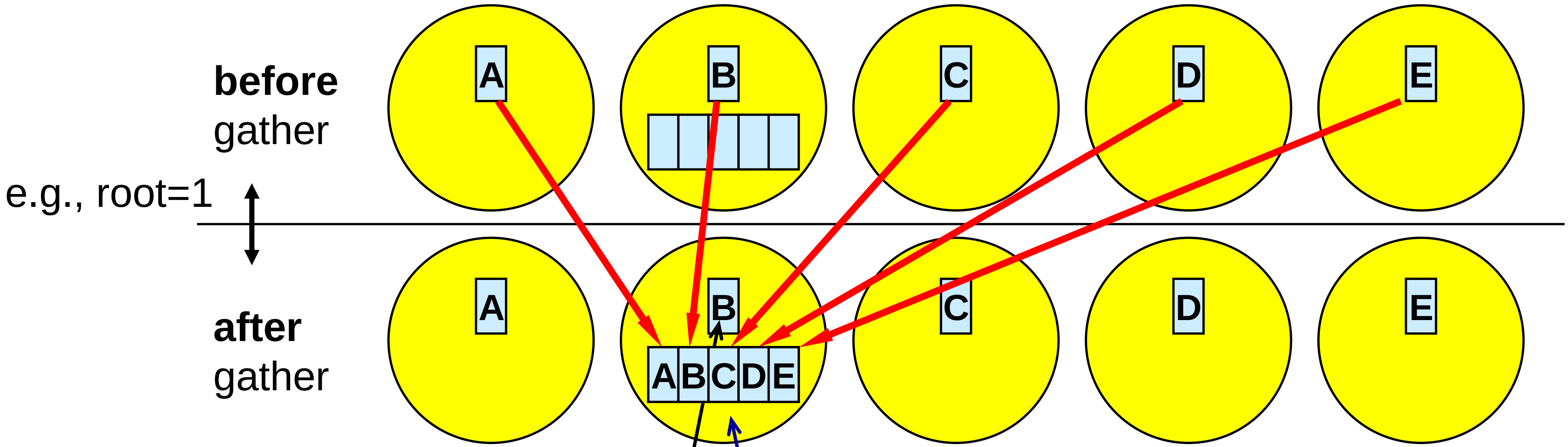
- **rank of the sending process (i.e., root process)**
- **must be given identically by all processes**

Scatter



- C: int MPI_Scatter(void ***sendbuf**, int sendcount, MPI_Datatype sendtype, void ****recvbuf***, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- C: int MPI_Scatterv(const void ***sendbuf**, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void ***recvbuf**, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Gather



- C: int MPI_Gather(void ***sendbuf**, int sendcount, MPI_Datatype sendtype,
  void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

- C: int MPI_Gatherv(const void ***sendbuf**, int sendcount, MPI_Datatype sendtype,
  void ***recvbuf**, const int *recvcounts, const int *displs, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

Click here for sample C code implementation of MPI Scatter & Gather

Introduction to MPI Virtual Topologies

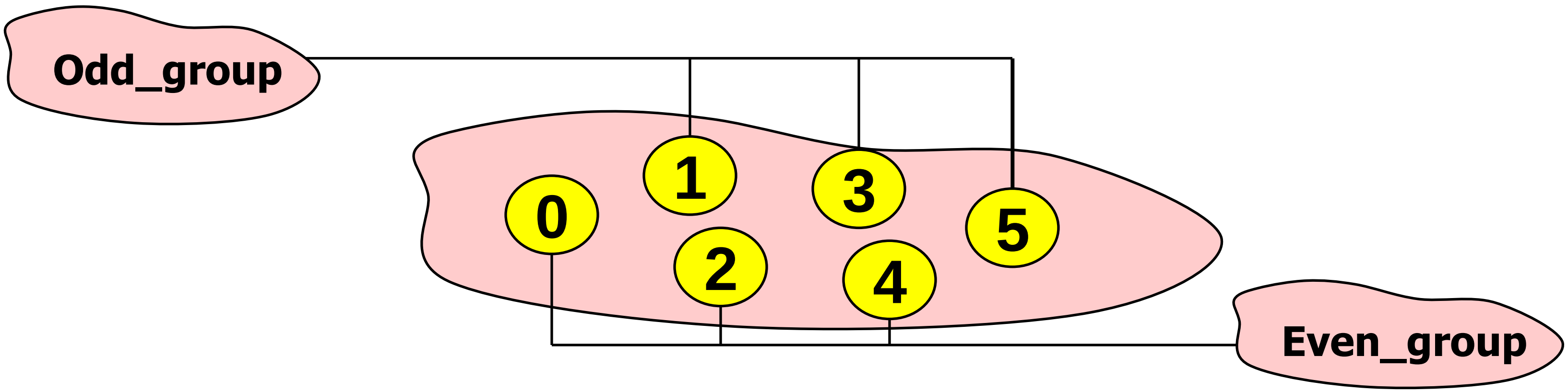
Topologies - Motivations

- Need to create sets of processes
  - For programming convenience
  - Make use of collectives routines
- Need to map the abstract topology onto the natural topology of the problem domain
  - For programming convenience
  - For performance

Groups & communicators

- A group is an ordered set of process identifiers
- Each process in a group is associated with an rank
- Usually one associates to groups communicators

Working with groups



- Select processes ranks to create groups
- Associate to these groups *new* communicators
- Use these new communicators as usual
- MPI_Comm_group(comm, group) returns in *group* the group associated to the communicator *comm*

For the previous example

- Odd_ranks={1, 3, 5}, Even_ranks={0, 2, 4}
  1. MPI_comm_group(MPI_COMM_WORLD, Old_group)
  2. MPI_Group_incl(Old_group, 3, Odd_ranks, &Odd_group)
  3. MPI_Group_incl(Old_group, 3, Even_ranks, &Even_group)
  - int MPI_Comm_create(MPI_COMM_WORLD, Odd_group, Odd_Comm)
  - int MPI_Comm_create(MPI_COMM_WORLD, Even_group, Even_Comm)

  - Alternatively…
  - color = modulo(myrank, 2)
  - MPI_Comm_split(MPI_COMM_WORLD, color, key, &newcomm)

Group Management

- Group Accessors
  - MPI_Group_size(…)
  - MPI_Group_rank(…)
  - …
- Group Constructors
  - MPI_COMM_GROUP(…)
  - MPI_GROUP_INCL(…)
  - MPI_GROUP_EXCL(…)
  - …
- Group Destructors
  - MPI_GROUP_FREE(group)

Communicator Management

- Communicator Accessors
  - MPI_COMM_SIZE(…)
  - MPI_COMM_RANK(…)
  - …
- Communicator Constructors
  - MPI_COMM_CREATE(…)
  - MPI_COMM_SPLIT(…)
- Communicator Destructors
  - MPI_COMM_FREE(comm)

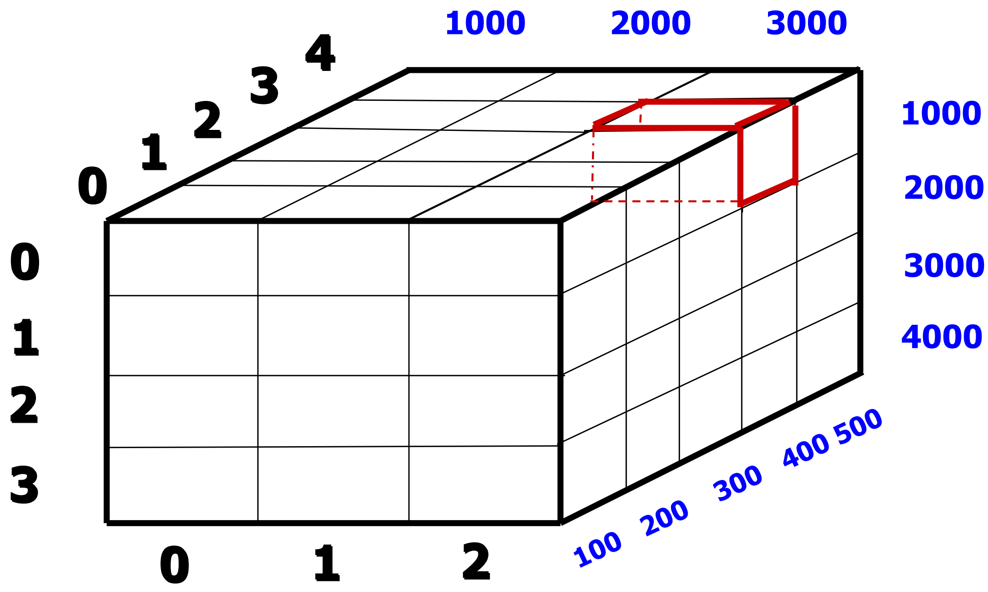
Virtual topology

- For more complex mapping, MPI routines are available
- Global array A(1:3000, 1:4000, 1:500) = $6 \cdot 10^9$ words
- on **3 x 4 x 5 = 60 processors**
- process coordinates **0..2, 0..3, 0..4**

- example:
  on process **$ic_0=2$, $ic_1=0$, $ic_2=3$ (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400) = $0.1 \cdot 10^9$ words

- **process coordinates:** handled with **virtual Cartesian topologies**
- Array decomposition: handled by the application program directly

Graphical representation



- Distribution of processes over the grid
- Distribution of the Global Array
- Coordinate (2, 0, 3) represents process number 43
- It is being assigned the cube A(2001:3000, 1:1000, 301:400)

Virtual Topologies

- Convenient process naming.
- Simplifies writing of code.
- Can allow MPI to optimize communications.

How to use a Virtual Topology

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

Topology Types

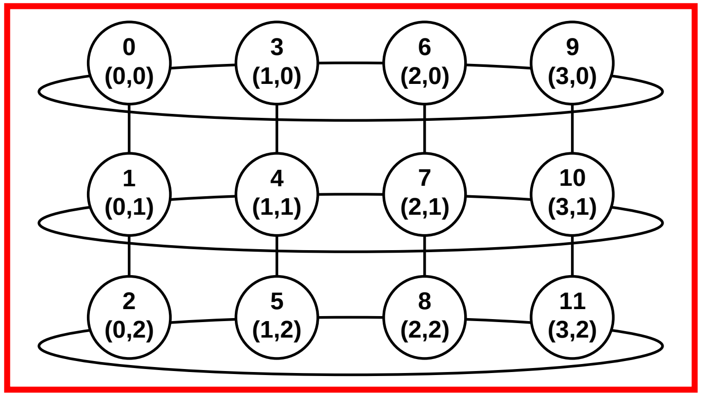
- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course,
    communication between any two processes is still allowed.

- Graph Topologies
  - general graphs,
  - not covered here.

Creating a Cartesian Virtual Topology

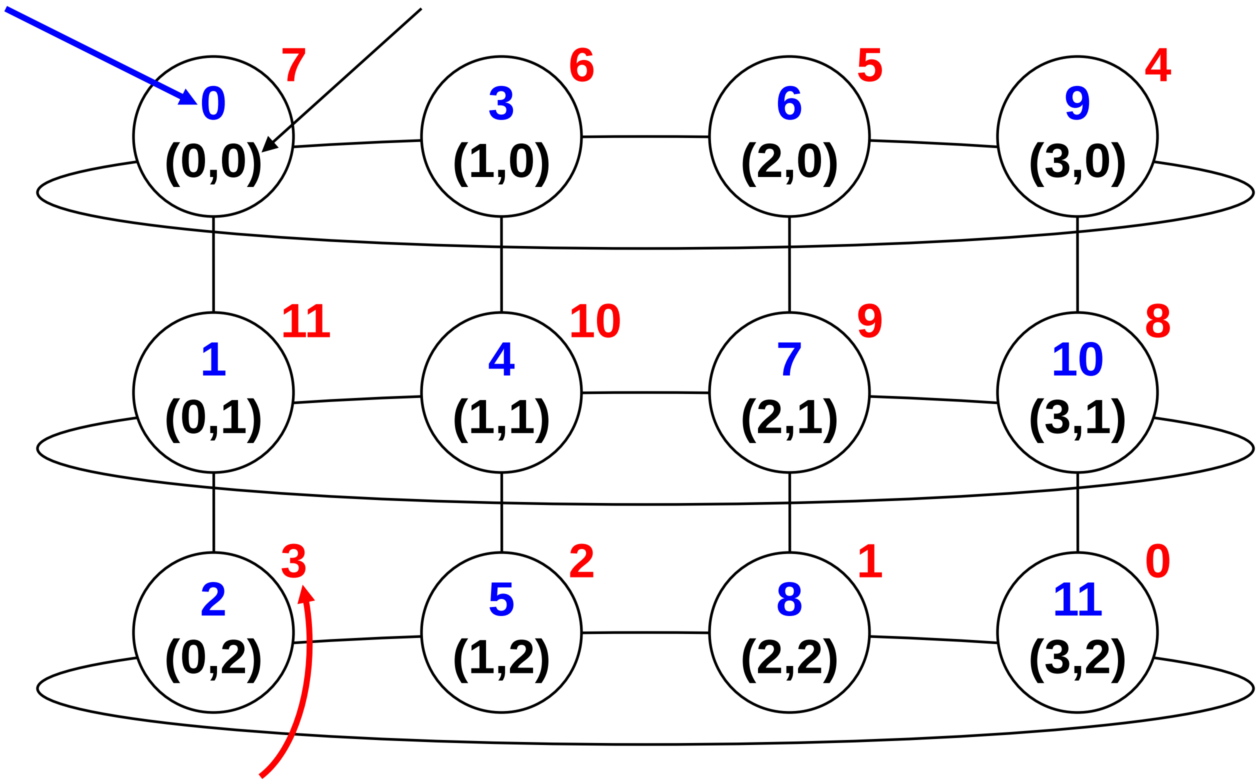
- int MPI_Cart_create(MPI_Comm comm_old, int ndims,
  int *dims, int *periods, int reorder,
  MPI_Comm **comm_cart*)

comm_old	**=**	**MPI_COMM_WORLD**
ndims	**=**	**2**
dims	**=**	**(4, 3)**
periods	**=**	**(1/.true., 0/.false.)**
reorder	**=**	***see next slide***

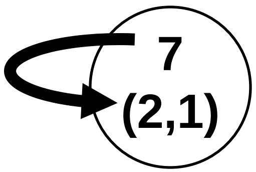


Example – A 2-dimensional Cylinder

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



Cartesian Mapping Functions



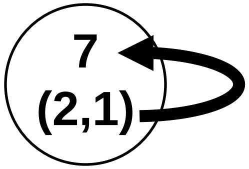
- Mapping
  ranks to
  process grid coordinates

- int MPI_Cart_coords(MPI_Comm comm_cart,
  int rank, int maxdims, int **coords*)

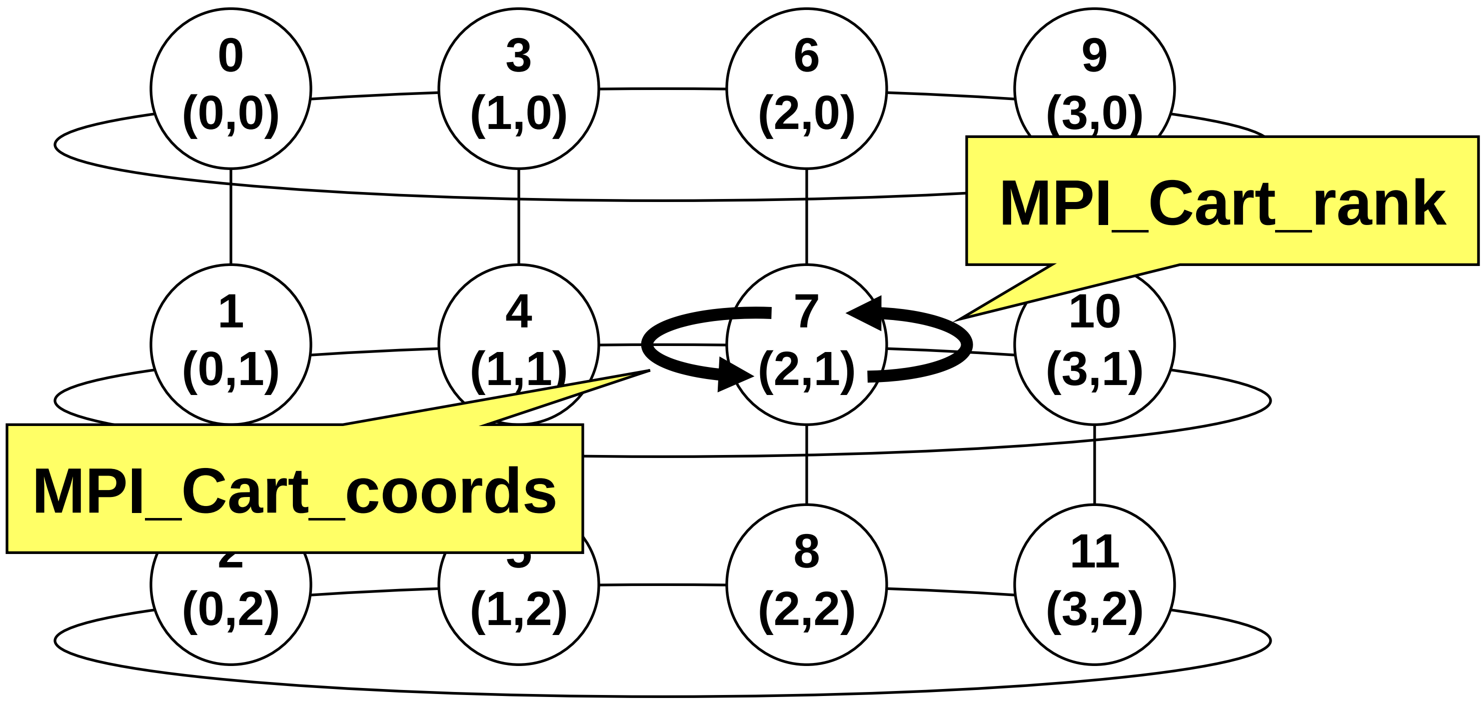
Cartesian Mapping Functions

- Mapping process grid coordinates to ranks

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



Own coordinates

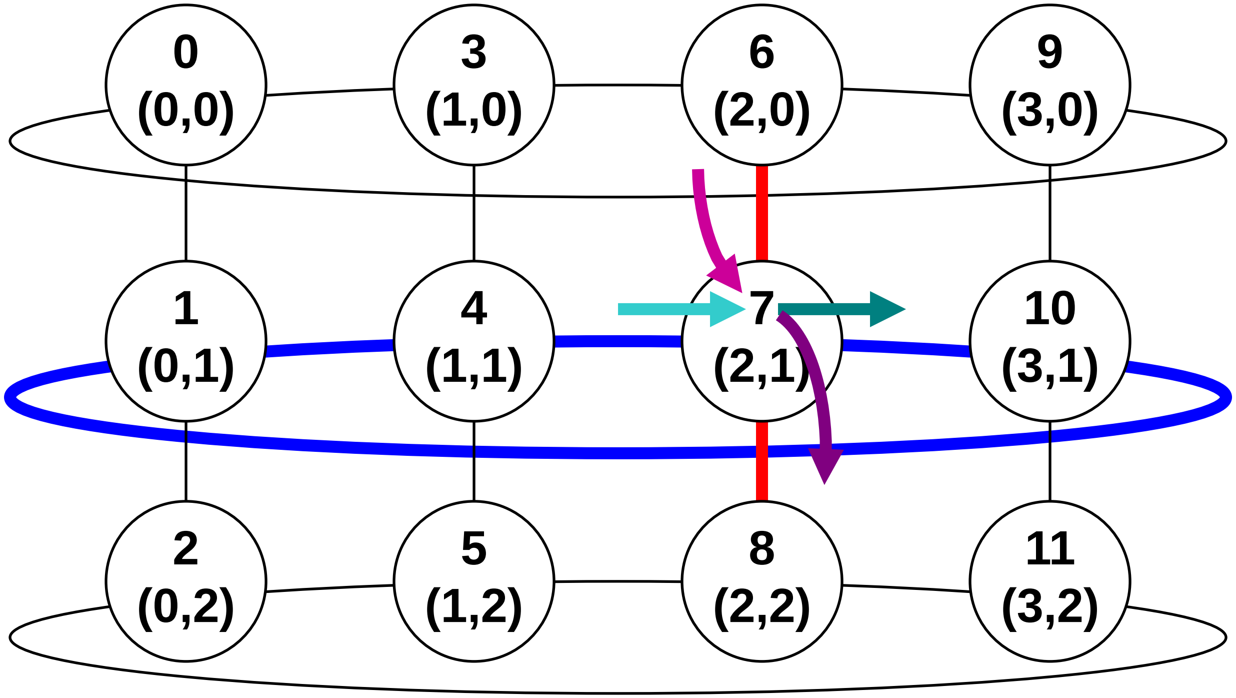


- Each process gets its own coordinates with
  MPI_Comm_rank(comm_cart, ***my_rank***, *ierror*)
  MPI_Cart_coords(comm_cart, **my_rank**, maxdims, ***my_coords***, *ierror*)

Cartesian Mapping Functions?

- Computing ranks of neighboring processes
- int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
  int **rank_prev*, int **rank_next*)
- Returns MPI_PROC_NULL if there is no neighbor.
- MPI_PROC_NULL can be used as source or destination rank in each communication

MPI_Cart_shift – Example



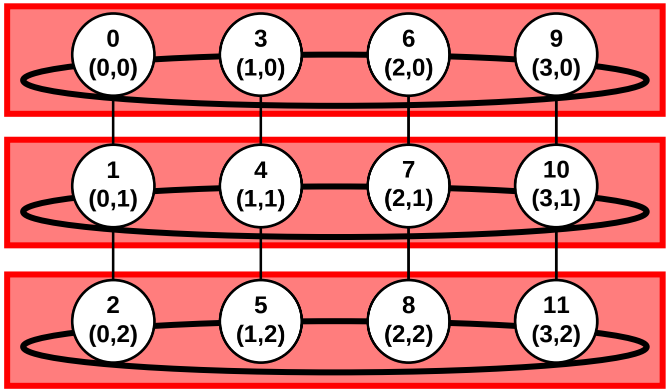
invisible input argument: **my_rank** in cart

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

Cartesian Partitioning

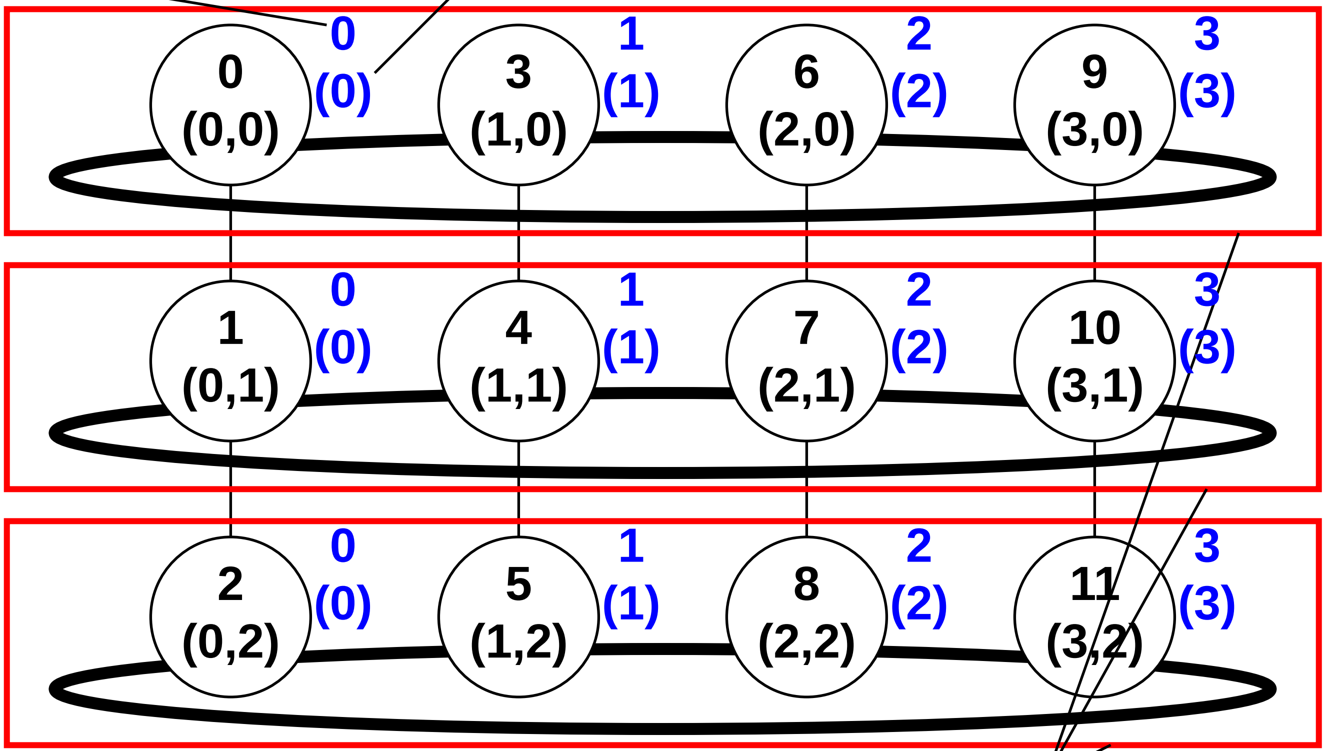
- Cut a grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

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



MPI_Cart_sub – Example

- **Ranks** and **Cartesian process coordinates** in **comm_sub**



- MPI_Cart_sub(comm_cart, remain_dims, ***comm_sub***, *ierror*)

(true, false)