

Virtual Topologies



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Virtual Topologies

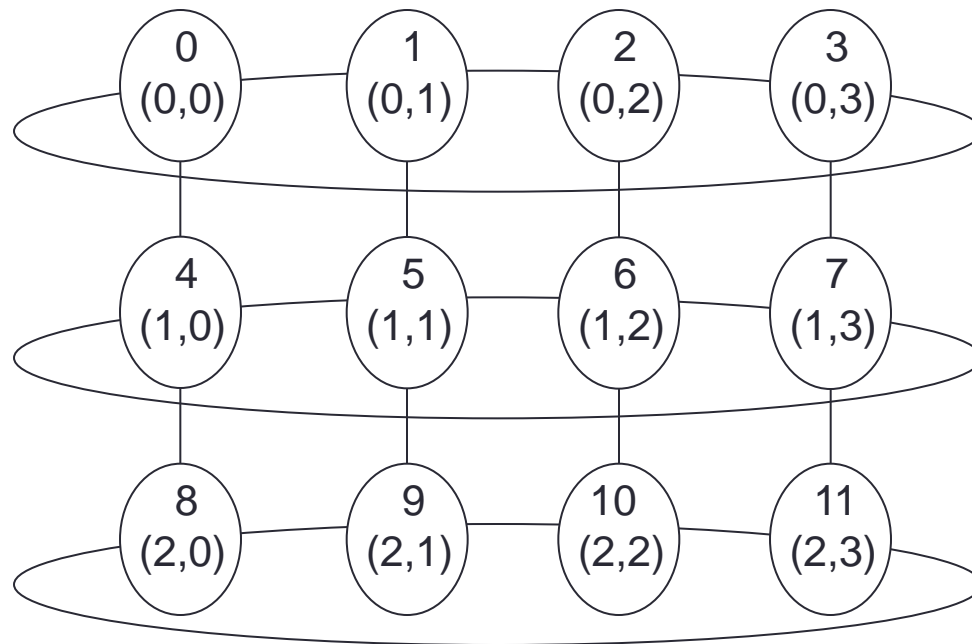
- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimise communications.

How to use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides “mapping functions”.
- Mapping functions compute processor ranks, based on the topology naming scheme.

Example

A 2-dimensional Cylinder



Topology types

- Cartesian topologies
 - each process is “connected” to its neighbours in a virtual grid.
 - boundaries can be cyclic, or not.
 - optionally re-order ranks to allow MPI implementation to optimise for underlying network interconnectivity.
 - processes are identified by cartesian coordinates.
- Graph topologies
 - general graphs
 - not covered here

Creating a 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)
```

```
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR  
LOGICAL PERIODS(*), REORDER
```

Balanced Processor Distribution

- C:

```
int MPI_Dims_create( int nnodes, int ndims,  
                    int *dims )
```

- Fortran:

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
```

```
INTEGER NNODES, NDIMS, DIMS(*), IERROR
```


MPI_Dims_create

- Call tries to set dimensions as close to each other as possible

dims before call	function call	dims on return
(0, 0)	<code>MPI_DIMS_CREATE(6, 2, dims)</code>	(3, 2)
(0, 0)	<code>MPI_DIMS_CREATE(7, 2, dims)</code>	(7, 1)
(0, 3, 0)	<code>MPI_DIMS_CREATE(6, 3, dims)</code>	(2, 3, 1)
(0, 3, 0)	<code>MPI_DIMS_CREATE(7, 3, dims)</code>	erroneous call

- Non zero values in dims sets the number of processors required in that direction
 - WARNING: make sure dims is set to zero before the call

Cartesian Mapping Functions

Mapping process grid coordinates to ranks

- C:

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

- Fortran:

```
MPI_CART_RANK (COMM, COORDS, RANK, IERROR)  
INTEGER COMM, COORDS (*), RANK, IERROR
```

Cartesian Mapping Functions

Mapping ranks to process grid coordinates

- C:

```
int MPI_Cart_coords(MPI_Comm comm, int rank,  
                   int maxdims, int *coords)
```

- Fortran:

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
```

```
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

Cartesian Mapping Functions

Computing ranks of my neighbouring processes
Following conventions of `MPI_SendRecv`

- C:

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

- Fortran:

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP,  
              RANK_SOURCE, RANK_DEST, IERROR)
```

```
INTEGER COMM, DIRECTION, DISP,  
        RANK_SOURCE, RANK_DEST, IERROR
```

Non-existent ranks

- What if you ask for the rank of a non-existent process?
 - or look off the edge of a non-periodic grid?
- MPI returns a NULL processor
 - rank is `MPI_PROC_NULL`
- **`MPI_PROC_NULL`** is a black hole
 - sends and receives complete immediately
 - send buffer disappears, receive buffer isn't touched
 - like UNIX `/dev/null`

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.
- **MPI_Cart_sub** and **MPI_CART_SUB** generate new communicators for the slices.
 - Use array to specify which dimensions should be retained in the new communicator.

Partitioning with MPI_CART_SUB

- C:

```
int MPI_Cart_sub ( MPI_Comm comm,  
                  int *remain_dims,  
                  MPI_Comm *newcomm)
```

- Fortran:

```
MPI_CART_SUB (COMM, REMAIN_DIMS,  
              NEWCOMM, IERROR)
```

```
INTEGER      COMM, NEWCOMM, IERROR  
LOGICAL      REMAIN_DIMS (*)
```

Exercise

- See Exercise 6 on the sheet
- Rewrite the exercise passing numbers round the ring using a one-dimensional ring topology.