



703650 VO Parallel Systems WS2019/2020

MPI Derived Datatypes and Virtual Topologies

Philipp Gschwandtner

Overview

- ▶ derived datatypes
 - ▶ allows to send user-specific datatypes
- ▶ virtual topologies
 - ▶ adds semantic position information to ranks
- ▶ tales from the proseminar
 - ▶ off-topic topics

Motivation

- ▶ we discussed using MPI for parallelization, but on a **very basic level**
 - ▶ we can only transfer contiguous ranges of arrays of the same element type
 - ▶ we need to manually compute rank numbers for talking to semantically significant and often-used ranks (e.g. left/right neighbor)
- ▶ what about
 - ▶ **transferring (nested) structs/classes, arrays of tuples, columns of a 2 D matrix, etc.**
 - ▶ ease of coding: **“send temperature to my left neighbor”** instead of **“send double to (myRank - 1 + numRanks) % numRanks”**



Derived Datatypes



Recap: MPI Datatypes

- ▶ several basic types predefined
 - ▶ MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, ...
- ▶ what about something like on the right?
 - ▶ struct with 4 members
 - ▶ 3x 8 bytes + 4 bytes

```
struct Particle {  
    double x;  
    double y;  
    double z;  
    int species;  
};
```

Issues With More Complex Data Structures

- ▶ MPI doesn't know how large a single element is
 - ▶ no predefined `MPI_(DATA_TYPE_THAT_DOESNT_EXIST_YET)`
 - ▶ what about nesting types? with differently-sized members?
 - ▶ sending individual elements blows up the code and causes performance overhead due to multiple messages
- ▶ issue of sending a single member of struct instances
 - ▶ bad solution: explicitly assemble send/receive buffers with single data type per message transfer
 - ▶ causes coding, memory footprint, and message overhead (at least one message per type)

Why not Just use MPI_BYTE/MPI_INT/MPI_... everywhere?

- ▶ adds strong typing to MPI library calls and allows automatic type handling
 - ▶ size of e.g. `int` is unknown (C standard only defines minimum requirements!)
 - ▶ `int` on machine A and `int` on machine B might have different size
 - ▶ machine A might be little-endian, machine B might be big-endian
 - ▶ saves a lot of explicit user-written `sizeof()` constructs
 - ▶ enables type-specific hardware optimizations for MPI
- ▶ using `MPI_BYTE/...` everywhere deprives you of all of the above

MPI Derived Datatypes

- ▶ composed of existing types
 - ▶ both basic and derived
- ▶ used to transfer high-level data structures
 - ▶ encodes more information in transfer, allows MPI to perform optimizations
 - ▶ more performance-efficient than individual transfer of data structure contents
 - ▶ less code, easier to read and maintain

MPI Derived Datatypes cont'd

- ▶ allow definition of new handles
 - ▶ e.g. MPI_FOOBAR
- ▶ require several steps
 - ▶ construction: declare and define new datatype
 - ▶ allocation / commit: needs to be done once by all ranks before using new datatype
 - ▶ usage (optional)
 - ▶ deallocation (optional): frees internal MPI storage, to be done once by all ranks

Selection of MPI Derived Datatype Facilities

- ▶ `MPI_Type_create_struct(...)`
 - ▶ specifies the data layout of user-defined structs (or classes)
- ▶ `MPI_Type_vector(...)`
 - ▶ specifies strided data, i.e. same-type data with missing elements
- ▶ `MPI_Type_create_subarray(...)`
 - ▶ specifies sub-ranges of multi-dimensional arrays
- ▶ `MPI_Type_contiguous(...)`
 - ▶ specifies a user-defined contiguous type comparable to C arrays

Structs

- ▶ `int MPI_Type_create_struct(int count, const int blocklengths[], const MPI_Aint displacements[], const MPI_Datatype types[], MPI_Datatype* newtype)`
 - ▶ `count`: number of blocks
 - ▶ `blocklengths`: number of elements per block (array)
 - ▶ `displacements`: starting address of first element of each block (array)
 - ▶ `types`: type of each block (array)
 - ▶ `newtype`: resulting derived datatype
- ▶ allows user-defined, aggregated types to be used in MPI communication directly

Structs: Block Lengths, Displacements and Types

```
struct Particle {  
    int posX;  
    int posY;  
    int posZ;  
    double magneticForceX;  
    double magneticForceY;  
    double magneticForceZ;  
};
```

} block no 0, starts at byte 0,
12 bytes long, type is integer

} block no 1, starts at byte 12,
24 bytes long, type is double

Careful with Displacements

- ▶ careful with manually specifying displacements
 - ▶ binary layout of your struct in memory is compiler-dependent!
 - ▶ e.g. struct members might be padded to multiples of 8 bytes
- ▶ use `offsetof()` instead!

```
MPI_Aint displacements[2] =  
    { 0,  
      12 };
```

```
// ===== vs =====
```

```
MPI_Aint displacements[2] =  
    { offsetof(Foo, posX),  
      offsetof(Foo, magneticForceX) };
```

Careful with Pointers

- ▶ don't transfer shallow copies of data
 - ▶ `double*` data might be not available or at a different address on node B
- ▶ try to avoid
 - ▶ otherwise, make a deep copy and ensure proper pointers

```
struct Particle {  
    int size;  
    double* data;  
};
```

Recommendation: Use C++ for more complex stuff, so you don't need to handle a lot of pointers.

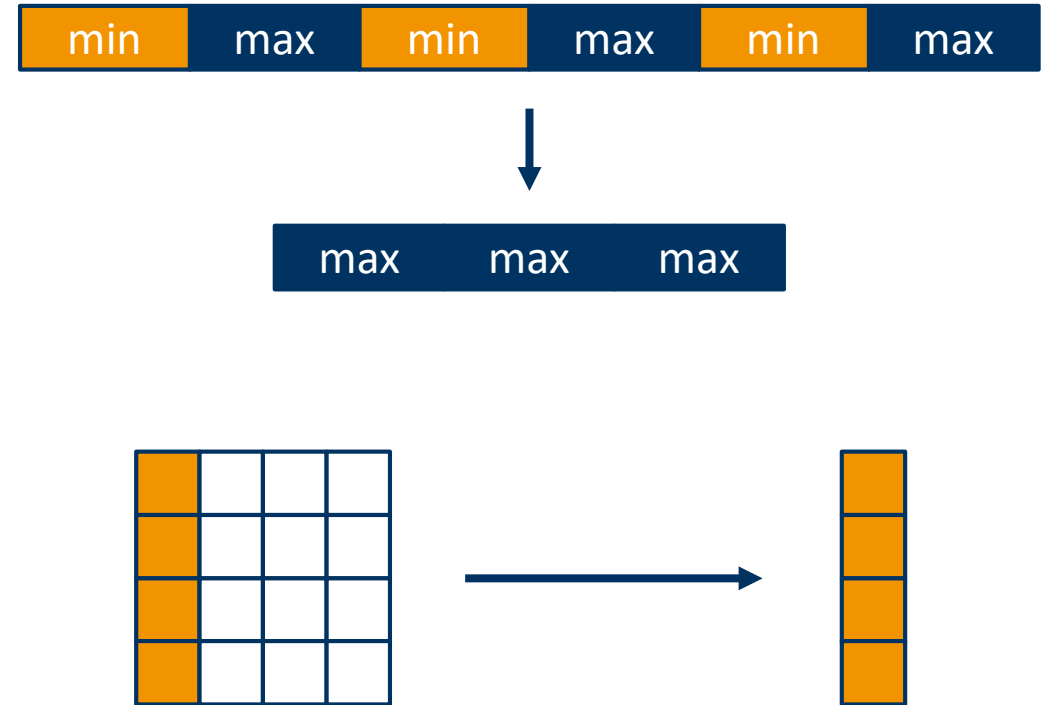
Struct Example

```
typedef struct {  
    int barInt;  
    double barDoubleA;  
    double barDoubleB;  
} Foo;  
MPI_Datatype myType;  
int blocklengths[2] = { 1, 2 };  
MPI_Aint displacements[2] =  
    { offsetof(Foo, barInt),  
      offsetof(Foo, barDoubleA) };  
MPI_Datatype datatypes[2] =  
    { MPI_INT, MPI_DOUBLE };
```

```
MPI_Type_create_struct(2, blocklengths,  
    displacements, datatypes, &myType);  
    commit needs to happen here  
if (myRank == 0) {  
    Foo data[2] = ...  
    MPI_Send(data, 2, myType, 1, 42,  
        MPI_COMM_WORLD);  
} else {  
    Foo data[2] = ...  
    MPI_Recv(data, 2, myType, 0, 42,  
        MPI_COMM_WORLD,  
        MPI_STATUS_IGNORE);  
}
```

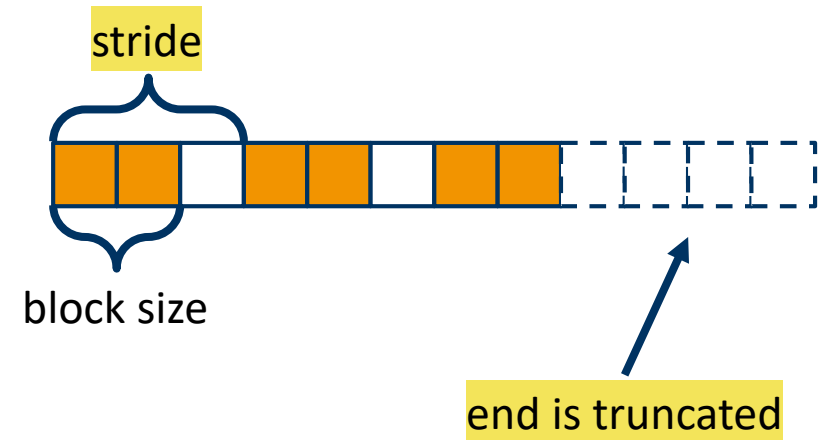
Non-Contiguous Data

- ▶ send all max values of an array of (min, max)-tuples to another rank
- ▶ send the column of a matrix
- ▶ do all of that without having to copy data to a contiguous buffer first!



Vectors

- ▶ Support strides (gaps in arrays)
 - ▶ e.g. take 2 elements, omit 1 element, repeat 3 times in total
 - ▶ useful for linear algebra



Vector Example

```
#define SIZE 20
#define STRIDE 3
#define COUNT 3
#define LENGTH 2

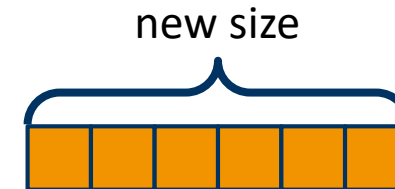
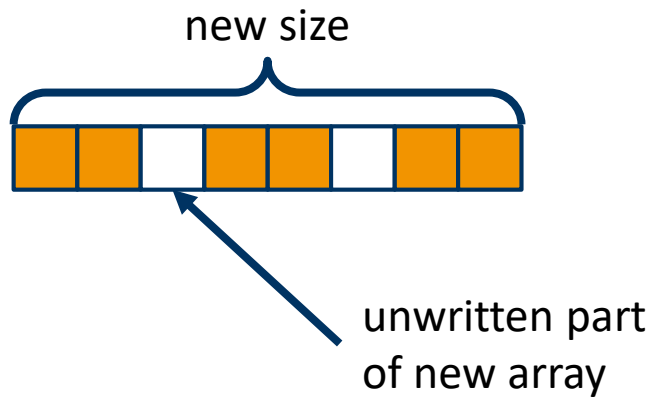
MPI_Datatype myType;
MPI_Type_vector(COUNT, LENGTH, STRIDE,
               MPI_CHAR, &myType);
MPI_Type_commit(&myType);
```

```
if (myRank == 0) {
    char data[SIZE] = ...;
    MPI_Send(data, 1, myType, 1, 42,
             MPI_COMM_WORLD);
} else {
    char data[SIZE];
    MPI_Recv(data, 1, myType, 0, 42,
             MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

Vector Variants

```
char data[SIZE];  
MPI_Recv(data, 1, myType,  
         0, 42, MPI_COMM_WORLD,  
         MPI_STATUS_IGNORE);
```

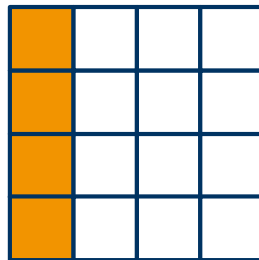
```
char data[COUNT*LENGTH];  
MPI_Recv(data, COUNT*LENGTH, MPI_CHAR,  
         0, 42, MPI_COMM_WORLD,  
         MPI_STATUS_IGNORE);
```



Use Case: Data Transposition

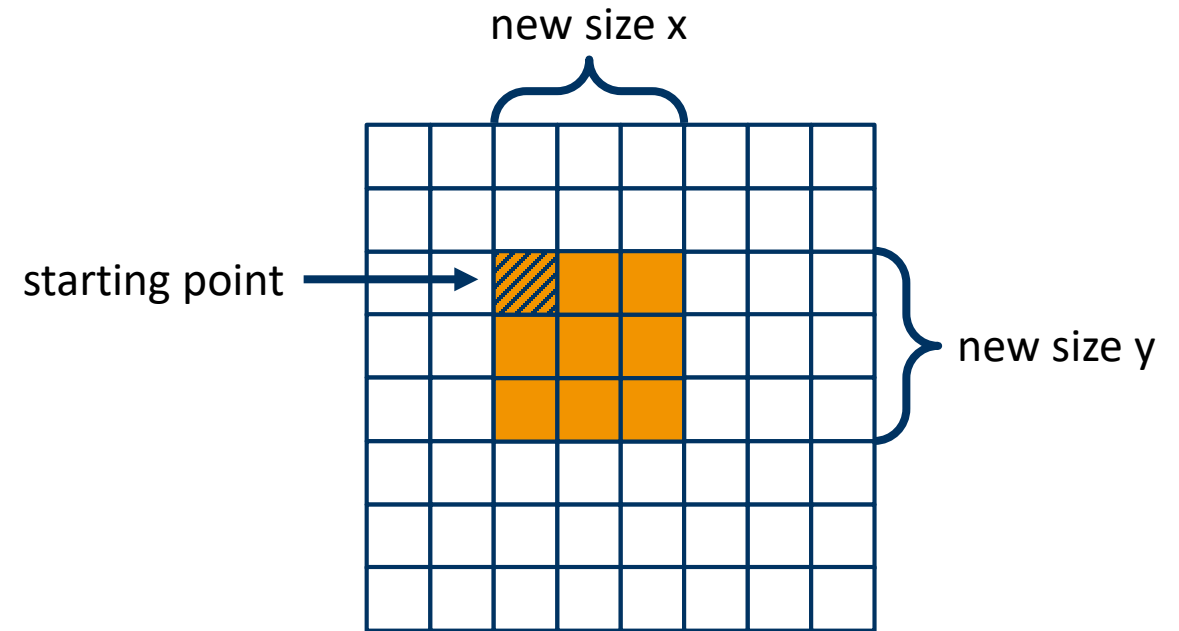
```
int data[SIZE][SIZE];  
MPI_Type_vector(SIZE, 1, SIZE, MPI_INT,  
                &myType);  
MPI_Send(data, 1, myType,  
          1, 42,  
          MPI_COMM_WORLD);
```

```
int data[SIZE];  
MPI_Recv(data, SIZE, MPI_INT,  
          0, 42,  
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Subarrays

- ▶ Allows to address a multi-dimensional sub-range of array elements



Subarray Example in 2 D

```
#define SIZE 8
#define SUBSIZE 3

MPI_Datatype myType;
int size[2] = { SIZE, SIZE };
int subSize[2] = { SUBSIZE, SUBSIZE };
int start[2] = { 2, 2 };

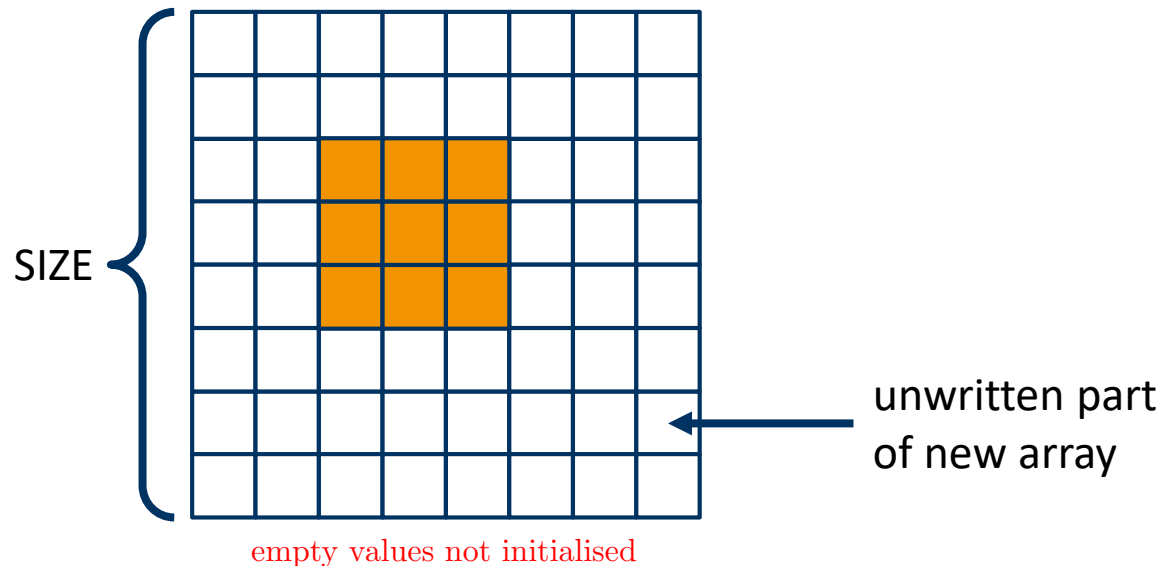
MPI_Type_create_subarray(2, size,
    subSize, start, MPI_ORDER_C, MPI_INT,
    &myType);
MPI_Type_commit(&myType);
```

```
if (myRank == 0) {
    int data[SIZE][SIZE] = ...;
    MPI_Send(data, 1, myType, 1, 42,
        MPI_COMM_WORLD);
} else {
    int subData[SUBSIZE][SUBSIZE];
    MPI_Recv(subData, SUBSIZE*SUBSIZE,
        MPI_INT, 0, 42, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}
```

Subarray Receive Variants

```
int data[SIZE][SIZE];  
MPI_Recv(data, 1,  
         myType, 0, 42, MPI_COMM_WORLD,  
         MPI_STATUS_IGNORE);
```

```
int subData[SUBSIZE][SUBSIZE];  
MPI_Recv(subData, SUBSIZE*SUBSIZE,  
         MPI_INT, 0, 42, MPI_COMM_WORLD,  
         MPI_STATUS_IGNORE);
```



Multiple Ways of Distributing Rows

- ▶ Allocate as a 1D array, use linearized indices
 - ▶ use 1D MPI vector with stride
 - ▶ (use nD MPI subarray with 1 dimension)
 - ▶ (use nD MPI darray with 1 dimension)
- ▶ Allocate as a nD array
 - ▶ use nested 1D MPI vectors
 - ▶ use nD MPI subarray
 - ▶ use nD MPI darray
- ▶ Same functional result for all of the above, but performance might differ
 - ▶ remember, MPI doesn't guarantee performance portability

Contiguous Derived Datatypes

- ▶ allows to aggregate same-type arrays into a single-count datatype
- ▶ has certain advantages
 - ▶ sending more than e.g. $2^{32}-1$ elements (count parameter type in MPI_Send/Recv/... is only int!)
 - ▶ allows semantic grouping and naming of data

```
MPI_Datatype myType;
MPI_Type_contiguous(SIZE, MPI_CHAR, &myType);
MPI_Type_commit(&myType);

char data[SIZE] = { 0 };

if(myRank == 0) {
    MPI_Send(data, 1, myType, 1, 42,
             MPI_COMM_WORLD);
} else {
    MPI_Recv(data, 1, myType, 0, 42,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

MPI_Type_free(&myType);
```

Packing/Unpacking

- ▶ MPI also offers `MPI_Pack(...)` and `MPI_Unpack(...)` functions
 - ▶ “Packs a datatype into contiguous memory” (MPICH documentation)
 - ▶ prefer this over derived datatypes? (hint: no)
- ▶ requires explicit copy of data from non-contiguous, user-defined form into a contiguous buffer to be sent with MPI
 - ▶ mostly superseded by MPI functions presented thus far, which directly access user-defined structures (no copy required)
 - ▶ pack/unpack still mostly offered for compatibility reasons, only very few edge cases

Free the Datatypes!

- ▶ call `MPI_Type_free(...)` once you no longer need the type
 - ▶ frees MPI-internal data storage for your custom type
 - ▶ reduces memory footprint for large numbers of datatypes
 - ▶ facilitates debugging
 - ▶ note: any pending communication using this type will continue and complete normally
 - ▶ omitted in most source code examples for obvious space reasons

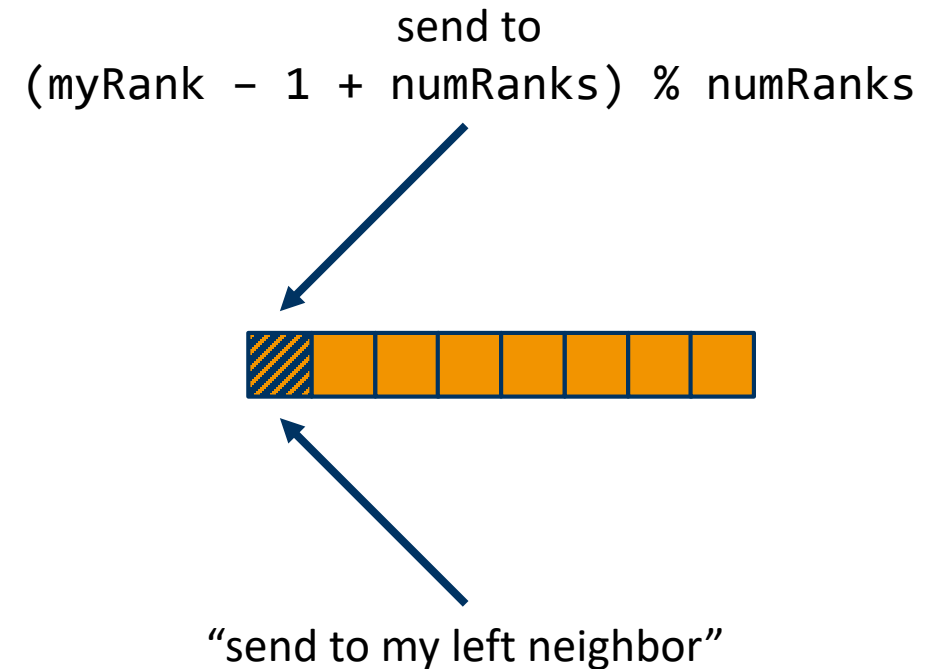


Virtual Topologies



Virtual Topologies

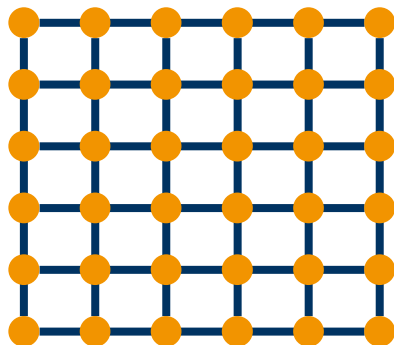
- ▶ allows to “name” MPI ranks and provide addresses with semantics
 - ▶ high-level view of MPI ranks
 - ▶ simplifies implementation of complex algorithms
 - ▶ called “virtual” because it doesn’t necessarily match hardware topology
- ▶ naming scheme should fit communication pattern
 - ▶ and reflect the real-world topological relationship of parts of your problem
 - ▶ enables MPI to perform optimizations



There are two Types of Topologies (According to MPI)

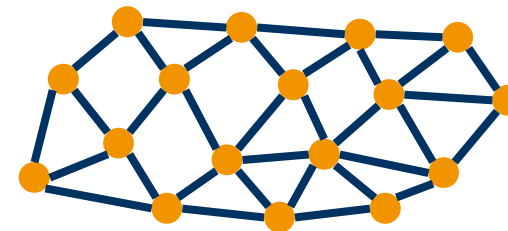
▶ Cartesian topologies

- ▶ regular grids of squares/cubes/...
- ▶ each rank is a node on the grid and connected to its neighbors
- ▶ boundaries can be periodic
- ▶ ranks can be identified via Cartesian coordinates instead of single index

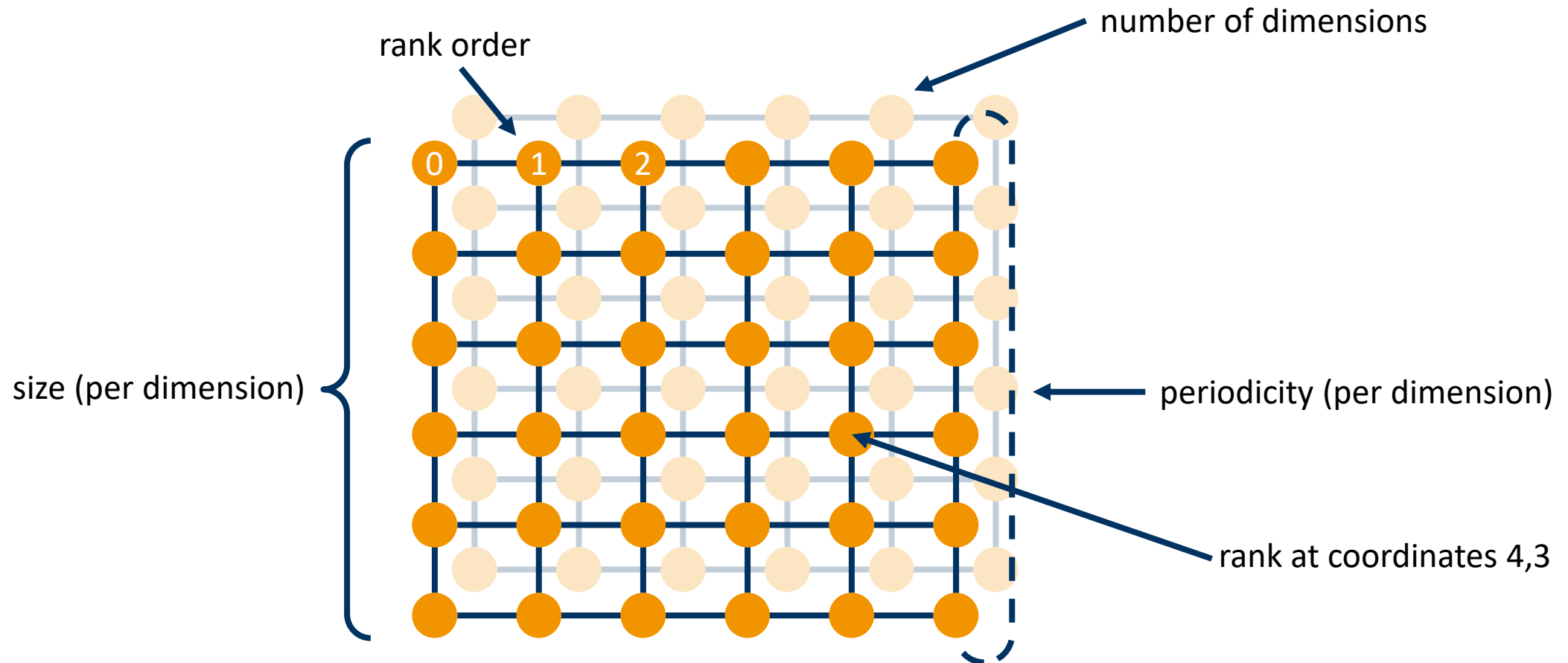


▶ graph topologies

- ▶ general graphs
- ▶ each rank is a vertex in the graph
- ▶ edges represent neighbor relationship
- ▶ edge weights specify communication intensity (facilitates optimization)
- ▶ not covered here



Properties of Cartesian Topologies



Working with Cartesian Topologies

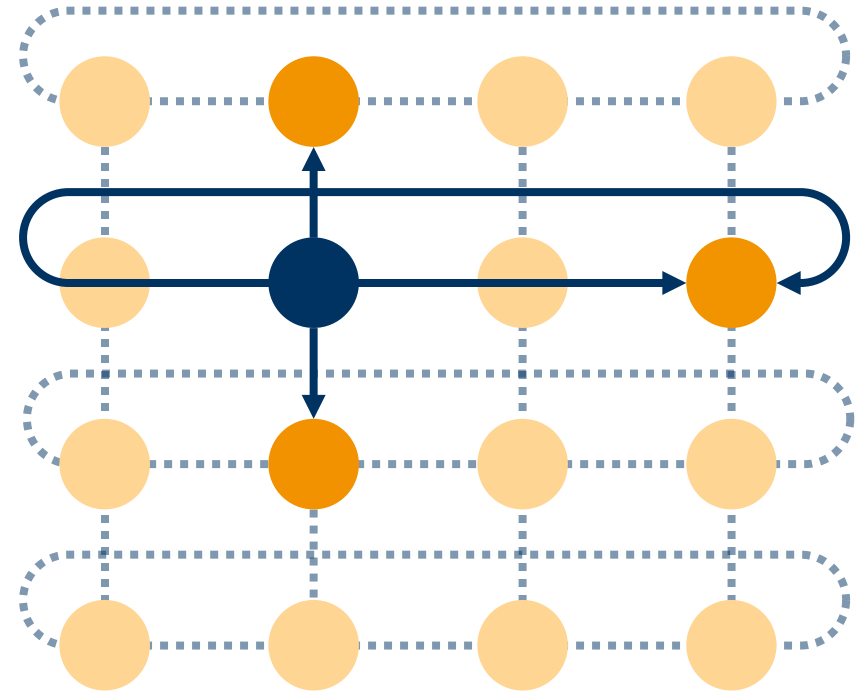
- ▶ **create topology, resulting in new communicator**
 - ▶ need to decide on dimensions, sizes, periodicity, etc...
 - ▶ per-dimension sizes can be computed using convenience function `MPI_Dims_create()`
 - ▶ new communicator implies ranks might have changed!
 - ▶ (remember MPI basics lecture: “[...] MPI semantics are relative to a “*communicator*” or “*group*”)
- ▶ compute rank numbers or coordinates as required
- ▶ communicate as you please
 - ▶ remember to specify correct communicator when using collective operations

Creating a Cartesian Topology

- ▶ `int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm* comm_cart)`
 - ▶ `comm_old`: current communicator
 - ▶ `ndims`: number of dimensions
 - ▶ `dims`: size, per dimension
 - ▶ `periods`: periodicity (0 = open, 1 = periodic), per dimension
 - ▶ `reorder`: reorder rank numbers (0 = false, 1 = true)
 - ▶ `comm_cart`: new communicator with cartesian topology

Shifting

- ▶ computes rank numbers of neighbors
 - ▶ requires direction and displacement (=distance)
- ▶ example on the right
 - ▶ partially periodic 2D topology of 4x4
 - ▶ up/down shift with displacement 1
 - ▶ left/right shift with displacement 2

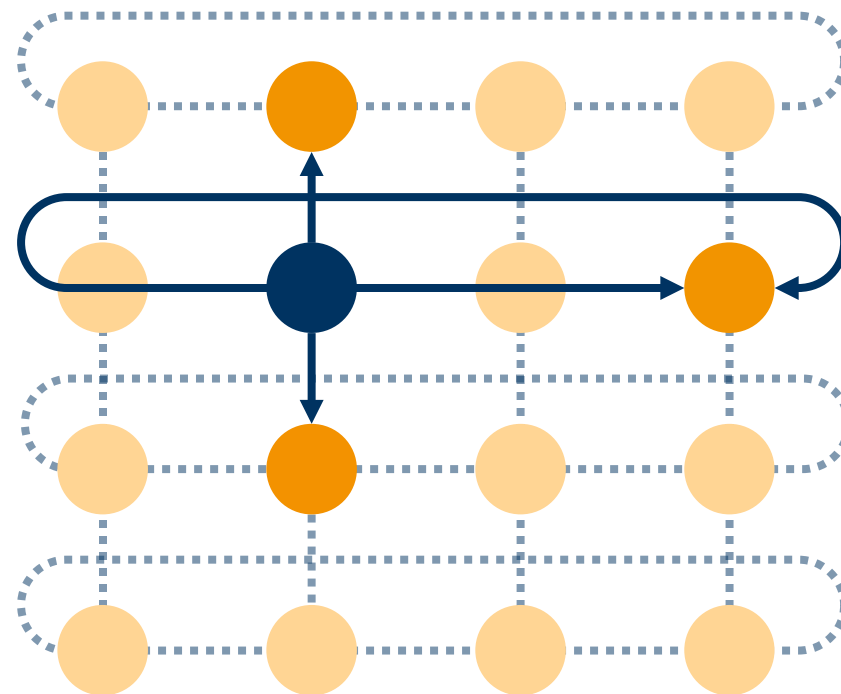


Shifting cont'd

- ▶ `int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int* rank_source, int* rank_dest)`
 - ▶ `comm`: communicator (must have cartesian topology!)
 - ▶ `direction`: dimension along which to select neighbors
 - ▶ `disp`: distance to neighbors
 - ▶ `rank_source`: neighbor for which the calling rank is the destination
 - ▶ `rank_dest`: requested neighbor for the calling rank

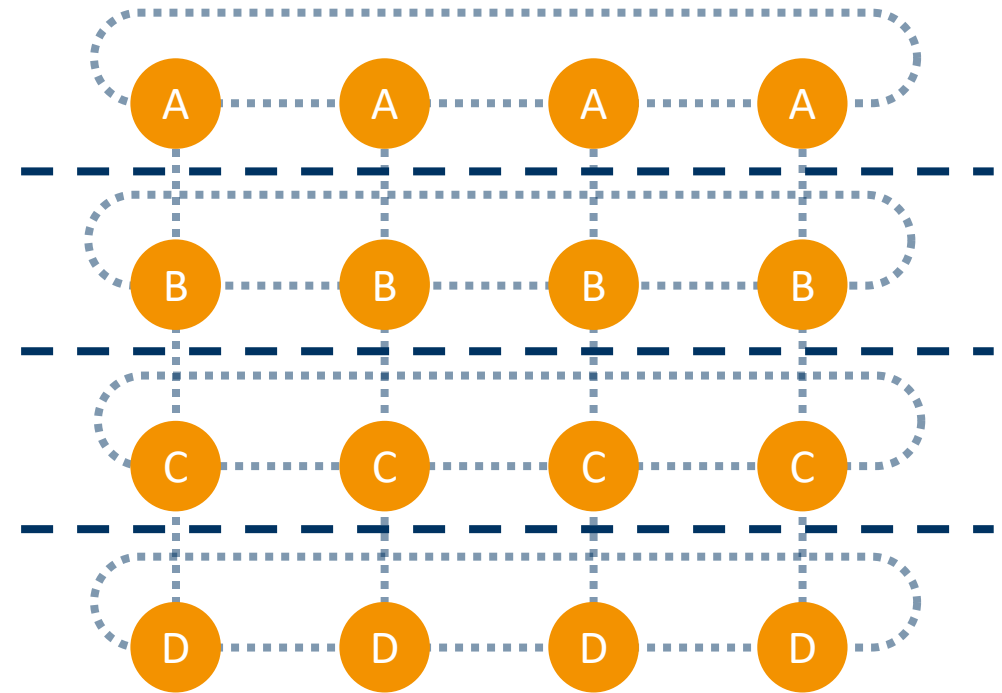
Shifting cont'd

```
MPI_Cart_shift(comm, 0, 1,  
               &source, &dest);  
// rank at 1,1:  
// source is 1,0  
// dest is 1,2
```



Slicing cont'd

- ▶ cuts a grid into slices
 - ▶ a new communicator is generated for each slice
 - ▶ enables slice-restricted collective communication
- ▶ example on the right
 - ▶ slicing a 2D topology horizontally
 - ▶ 4 new communicators A, B, C, and D with 4 ranks each
 - ▶ `MPI_Bcast(..., A)` only affects ranks of A



Slicing cont'd

- ▶ `int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm* newcomm)`
 - ▶ `comm`: current communicator (must have cartesian topology!)
 - ▶ `remain_dims`: which dimensions to keep in sub-grid (0 = drop, 1 = include)
 - ▶ `newcomm`: new communicator holding only ranks of this slice

Additional Convenience Functions

- ▶ `MPI_Cart_coords(...)`
 - ▶ compute coordinates from a given rank (17 → [4, 1])
- ▶ `MPI_Cart_rank(...)`
 - ▶ compute rank from given coordinates ([4,1] → 17)
- ▶ `MPI_Cart_sub(...)`
 - ▶ partition grid into lower-dimension sub-grids (e.g. 2D square from 3D cube)
- ▶ `MPI_Cartdim_get(...)/MPI_Cart_get(...)`
 - ▶ get topology information for a given communicator

Tales from the Proseminar: Verification and Validation

- ▶ absolutely not the same thing, though often used synonymously
- ▶ verification means checking your implementation
 - ▶ ensure that implementation meets the specification
 - ▶ check that software output is correct
- ▶ validation means checking your specification
 - ▶ ensure that the specification meets requirements
 - ▶ check that software output serves the use case purpose

Summary

- ▶ derived data types can be very handy
 - ▶ no need to copy data to basic, contiguous buffers
 - ▶ allows to easily transpose data
 - ▶ arbitrary nesting possible
- ▶ virtual topologies add semantic position information to ranks
 - ▶ makes rank positions easily identifiable
 - ▶ allows direct neighbor communication
 - ▶ enables limited-scope collectives
- ▶ verification vs. validation