Winter term 2025/2026

Practical parallel algorithms with MPI

Dr. Jan Eitzinger
Erlangen National High Performance Computing Center (NHR@FAU)

Lecture 6: Advanced MPI: derived data types and virtual topologies

Motivation heterogenous data

Example: Root reads configuration and broadcasts it to all others

```
// root: read configuration from
// file into struct config
MPI_Bcast(&cfg.nx, 1, MPI_INT, …);
MPI_Bcast(&cfg.ny, 1, MPI_INT, …);
MPI_Bcast(&cfg.du, 1, MPI_DOUBLE,…);
MPI_Bcast(&cfg.it, 1, MPI_INT, …);
```

Want to do something like:

```
MPI_Bcast(
   &cfg, 1, <type cfg>,…);
```

```
MPI_Bcast(&cfg, sizeof(cfg),
    MPI_BYTE, ..)
```

is **not** a solution. It's not portable as no data conversion can take place

Motivation non-contiguous data

- Example: Send column of matrix (noncontiguous in C):
  - Send each element alone?
  - Manually copy elements out into a contiguous buffer and send it?

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

MPI general datatypes

- Datatypes allow to (de)serialize arbitrary data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O
- Powerful, generic specification to describe arbitrary data layouts
- Declarative specification of data-layout “what” and not “how” , leaves optimization to implementation

- Derived datatypes constructed from basic datatypes using constructors
- Recursive concept: Derived data types can be constructed from other derived datatypes

Specification

- A general MPI datatype object consists of a pair of
  - Sequence of basic datatypes
  - Sequence of integer (byte) displacements

Need not be distinct, positive or in increasing order!

- Nomenclature:
  - Type map: Sequence of pairs (datatype, displacement)

```
{(int, 64),(double, 8),(double,8),(double,48)};
```

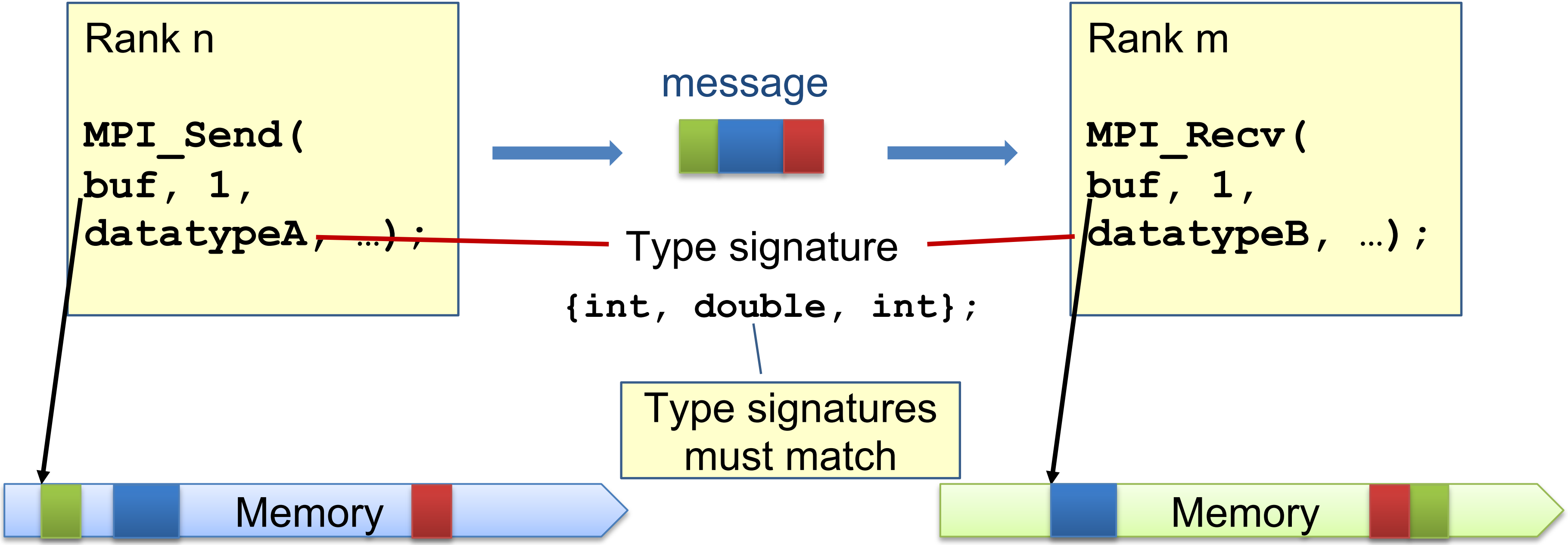
  - Type signature: Sequence of datatypes (displacements ignored)

```
{int, double, double, double};
```

Example

Typemap datatypeA
`{(int, 0),(double,8),(int,64)};`

Typemap datatypeB
`{(int,68),(double,0),(int,64)};`



MPI datatype extent

- First byte to last byte occupied, rounded to alignment requirements

$$\texttt{Typemap} = \{(\texttt{type}_0, \texttt{disp}_0), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1})\},$$
$$\texttt{lb(Typemap)} = \min(\texttt{disp}_j)$$
$$\texttt{ub(Typemap)} = \max(\texttt{disp}_j + \texttt{sizeof}(\texttt{type}_j)) + \varepsilon$$

$$\texttt{extent(Typemap)} = \texttt{ub(Typemap)} - \texttt{lb(Typemap)}$$

Example

```
{(double, 0),(char, 8)};
```

Assumption: doubles are 8-byte aligned!

Extent of datatype is 16.

Creating an MPI data type

Three steps:

1. Construct with

   ```
   MPI_Type_*(...);
   ```

2. Commit new data type with

   ```
   MPI_Type_commit(MPI_Datatype * nt);
   ```

3. After use, deallocate the data type with

   ```
   MPI_Type_free(MPI_Datatype * nt);
   ```

All local, non-collective calls

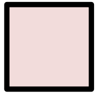
The simplest type: `MPI_Type_contiguous`

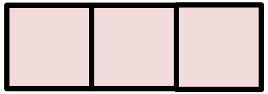
```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype * newtype);
```

`count` `3 (no. of items)`

Can also be general derived MPI type!

`oldtype` `MPI_INT`

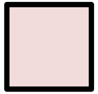


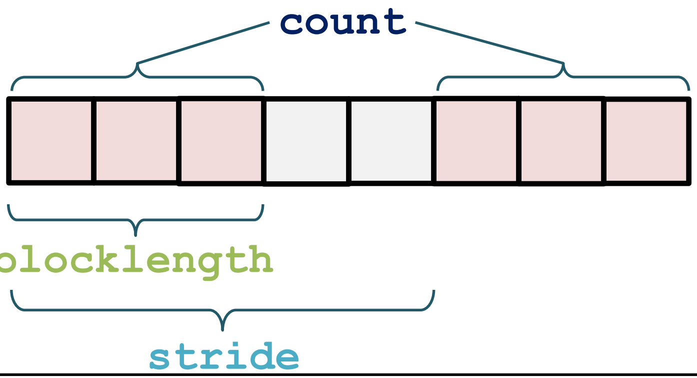


```
MPI_Datatype nt;
MPI_Type_contiguous(
3, MPI_INT, &nt);

MPI_Type_commit(&nt);
// use nt…
MPI_Type_free(&nt);
```

A flexible, vector-like type: `MPI_Type_vector`

```
MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatype oldtype,
                MPI_Datatype * newtype);
```

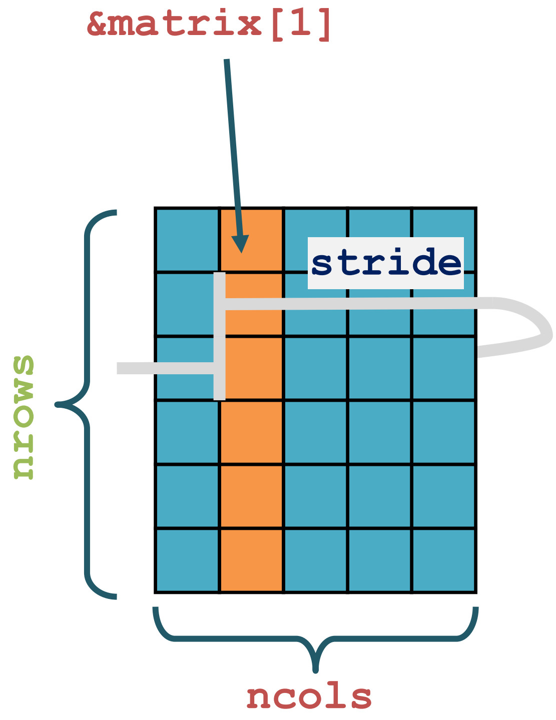
count	2 (no. of blocks)
blocklength	3 (no. of elements in each block)
stride	5 (no. of elements b/w start of each block)
oldtype	MPI_INT 




```
MPI_Datatype nt;
MPI_Type_vector(
2, 3, 5, MPI_INT, &nt);

MPI_Type_commit(&nt);
// use nt…
MPI_Type_free(&nt);
```

Sending a column of a matrix in C

Row-major data layout in C → cannot use plain array




```
double matrix[30];
MPI_Datatype nt;

// count = nrows, blocklength = 1,
// stride = ncols
MPI_Type_vector(nrows, 1, ncols,
                MPI_DOUBLE, &nt);
MPI_Type_commit(&nt);

// send column
MPI_Send(&matrix[1], 1, nt, …);

MPI_Type_free(&nt);
```

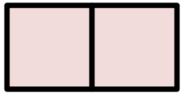
An even more flexible type: MPI_Type_indexed

```
MPI_Type_indexed(int count,
int blocklengths[], int displacements[],
 MPI_Datatype oldtype, MPI_Datatype * newtype);
```

`int count`	number of blocks, also number of entries in array of displacements and array of blocklengths
`int blocklengths[]`	number of elements per block (array of non-negative integers)
`int displacements[]`	displacement for each block, in multiples of oldtype (array of integers)

Simple example `MPI_Type_indexed`

`Old typemap : {(double, 0),(double,8)};`



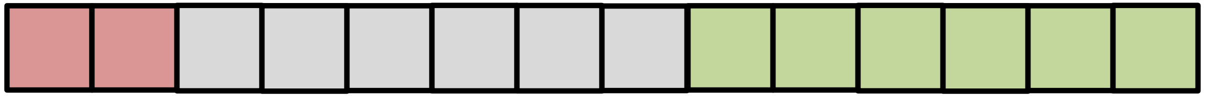
`extent : 16`

`count : 2`

`blocklengths : (3, 1)`

`displacements : (4, 0)`

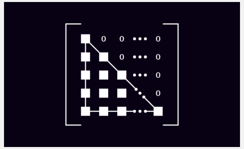
```
New typemap : { (double, 64), (double,72),
               (double,80), (double,88), (double,96), (double,104),
               (double,0), (double,8) };
```



A more realistic example

```c
double a[100], b[100];
int displs[10];
int lens[10];
MPI_Datatype tril;
MPI_Status status;

for (int i = 0; i < 10; i++) {
   displs[i] = i * 10;
   lens[i]   = i + 1;
}

MPI_Type_indexed(10, lens, displs, MPI_DOUBLE, &tril);
MPI_Type_commit(&tril);

MPI_Sendrecv(a, 1, tril, rank, 0, b, 1, tril, rank, 0,
     MPI_COMM_WORLD, &status);
```

Copy the lower triangular part of a matrix between two arrays on same process



Constructor variants using addresses

- Type constructors provide variant with displacement in bytes instead of element extent:
  - `MPI_Type_create_hvector`
  - `MPI_Type_create_hindexed`
- Special MPI type for addresses and displacements: `MPI_Aint`
- Always use MPI provided routines for address computations!

- Usually, addresses are relative to buffer address provided to communication routines
- It is allowed to use absolute addresses, in this case set buffer address = `MPI_BOTTOM`

How to obtain and handle addresses?

```
MPI_Get_address(const void *location, MPI_Aint *address);
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2);
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp);
```

- Example:

```
double a[100];
MPI_Aint a1, a2, disp;
MPI_Get_address(&a[0],  &a1);
MPI_Get_address(&a[50], &a2);
disp = MPI_Aint_diff(a2,a1);
```

Result would usually be `disp = 400` (50 x 8)

Derived type size and extent

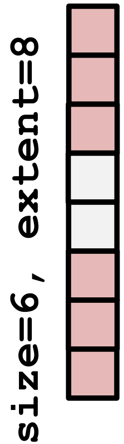
- Get the total **size** (in bytes) of datatype in a message (Type signature)

```
int MPI_type_size(MPI_Datatype newtype, int *size);
```

- Get the lower bound and the **extent** (span from the first byte to the last byte) of datatype

```
int MPI_type_get_extent(MPI_Datatype newtype,
                        MPI_Aint *lb,
                        MPI_Aint *extent);
```

MPI type for memory addresses or offsets

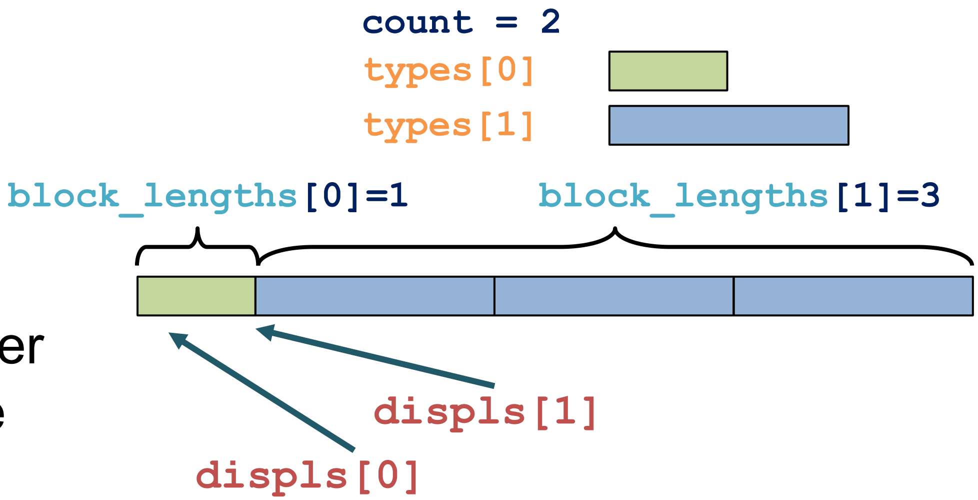


- MPI allows to change the extent of a datatype
  - using `lb_marker` and `ub_marker`
  - does not affect the size or count of a datatype, and the message content
  - does affect the outcome of a replication of this datatype

Most flexible type: `MPI_Type_create_struct`

Describe blocks with arbitrary data types and arbitrary displacements

```
MPI_Type_create_struct(int count, int block_lengths[],
   MPI_Aint displs[], MPI_Datatype types[],
   MPI_Datatype * newtype);
```



The contents of **displs** are either the displacements in bytes of the block bases or MPI addresses

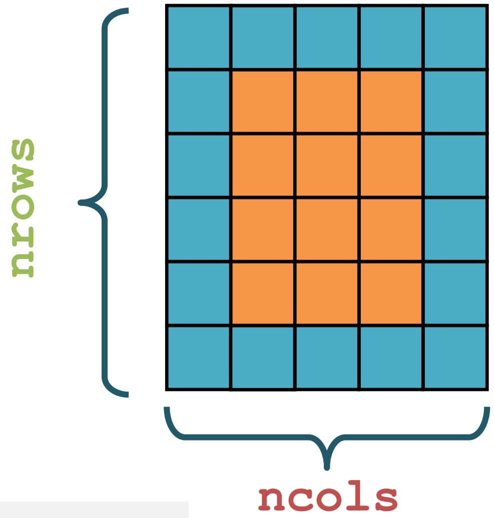
A sub-array type: `MPI_Type_create_subarray`

```
MPI_Type_create_subarray(int dims,
   int ar_sizes[], int ar_subsizes[], int ar_starts[],
   int order, MPI_Datatype oldtype, MPI_Datatype * newtype);
```

- **`dims`**: dimension of the array
- **`ar_sizes`**: array with sizes of array (dims entries)
- **`ar_subsizes`**: array with sizes of subarray (dims entries)
- **`ar_starts`**: start indices of the subarray inside array (dims entries), start at 0 (also in Fortran)
- **`order`**
  - row-major: `MPI_ORDER_C`
  - column-major: `MPI_ORDER_FORTRAN`

Example for a sub-array type: “bulk” of a matrix

`dims`	`2`
`ar_sizes`	`{ncols, nrows}`
`ar_subsizes`	`{ncols-2, nrows-2}`
`ar_starts`	`{1, 1}`
`order`	`MPI_ORDER_C`
`oldtype`	`MPI_INT`




```
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
                         ar_starts, order, oldtype, &nt);
MPI_Type_commit(&nt);
// use nt...
MPI_Send(&buf[0], 1, nt, …); // etc.
MPI_Type_free(&nt);
```

Performance considerations

- Derived datatypes enable zero copy: MPI forces “remote” copy

- MPI implementations copy internally
  - E.g., networking stack (TCP), packing derived data types
  - Zero-copy is possible (RDMA, I/O Vectors, SHMEM)
- MPI applications copy too often
  - E.g., manual pack, unpack or data rearrangement
  - Derived datatypes can do both!

- Simple and effective performance model:
  - More parameters == slower

Some implementations might still be slower than manual packing unpacking!

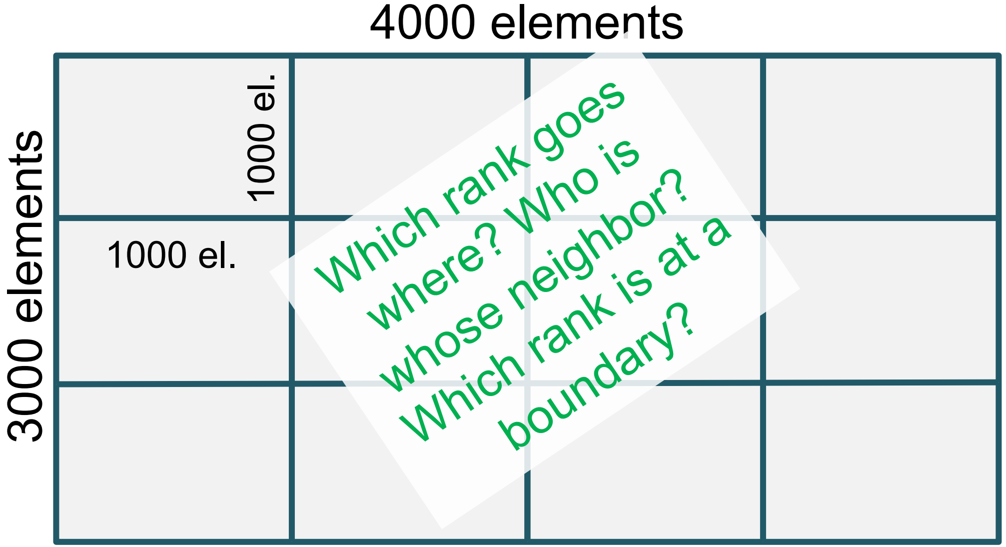
Virtual (Cartesian) topologies in MPI

A convenient process naming scheme for multi-dimensional problems

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

Example: distribute 2-D array of 4000 x 3000 elements equally on 12 ranks

- Let MPI map ranks to coordinates
- User: map array segments to ranks



Creating a new Cartesian communicator

- Create new communicator attached to Cartesian topology

```
MPI_Cart_create(MPI_Comm oldcomm,
     int ndims, int dims[], int periods[],
     int reorder, MPI_Comm *cart_comm);
```

`ndims`: number of dimensions

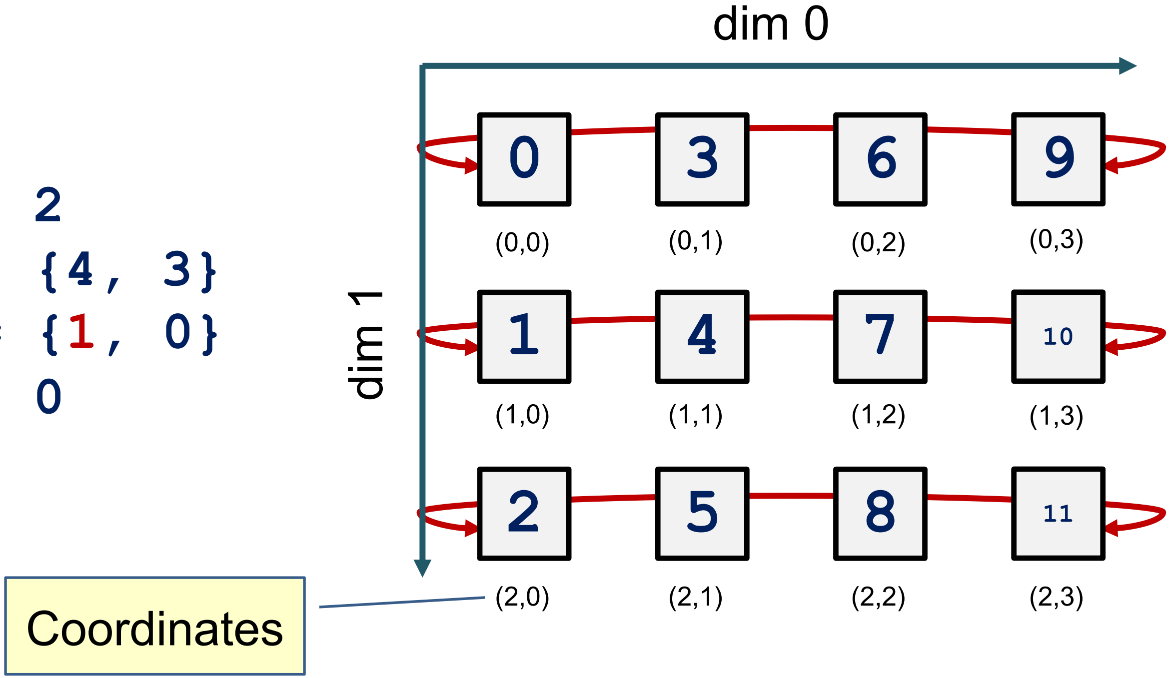
`dims`: array with `ndims` elements,
`dims[i]` specifies the number of ranks in dimension `i`

`periods`: array with `ndims` elements,
`periods[i]` specifies if dimension `i` is periodic

`reorder`: allow rank of `oldcomm` to have a different rank in `cart_comm`

Cartesian topology example

```
ndims    = 2
dims     = {4, 3}
periods  = {1, 0}
reorder  = 0
```



Convenience Function: `MPI_Dims_create`

- Select a balanced distribution of processes per coordinate direction

```
MPI_Dims_create(
    int nnodes, int ndims, int dims[]);
```

`nnodes`: number of nodes in grid, usually the number of processes
`ndims`: number of cartesian dimensions
`dims`: array with **`ndims`** elements, number of nodes in each dimension

Argument **`dims`** allows to control the layout:

- If **`dims[i]`** is set to a positive number, # nodes in dimension i is not touched
- Only entries where **`dims[i]`** is set to 0 are modified by call! There is an error if **`dims[i]`** is not initialized!

Cartesian topology service functions

- Retrieve rank in new Cartesian communicator ("who am I in the grid?")

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

- Map rank → coordinates ("where am I in the grid? ")

```
MPI_Cart_coords(cart_comm, rank, int maxdims, int coords[]);
```

`rank:` any rank which is part of Cartesian communicator `cart_comm`
`coords:` array of `maxdims` elements, receives the coordinates for `rank`

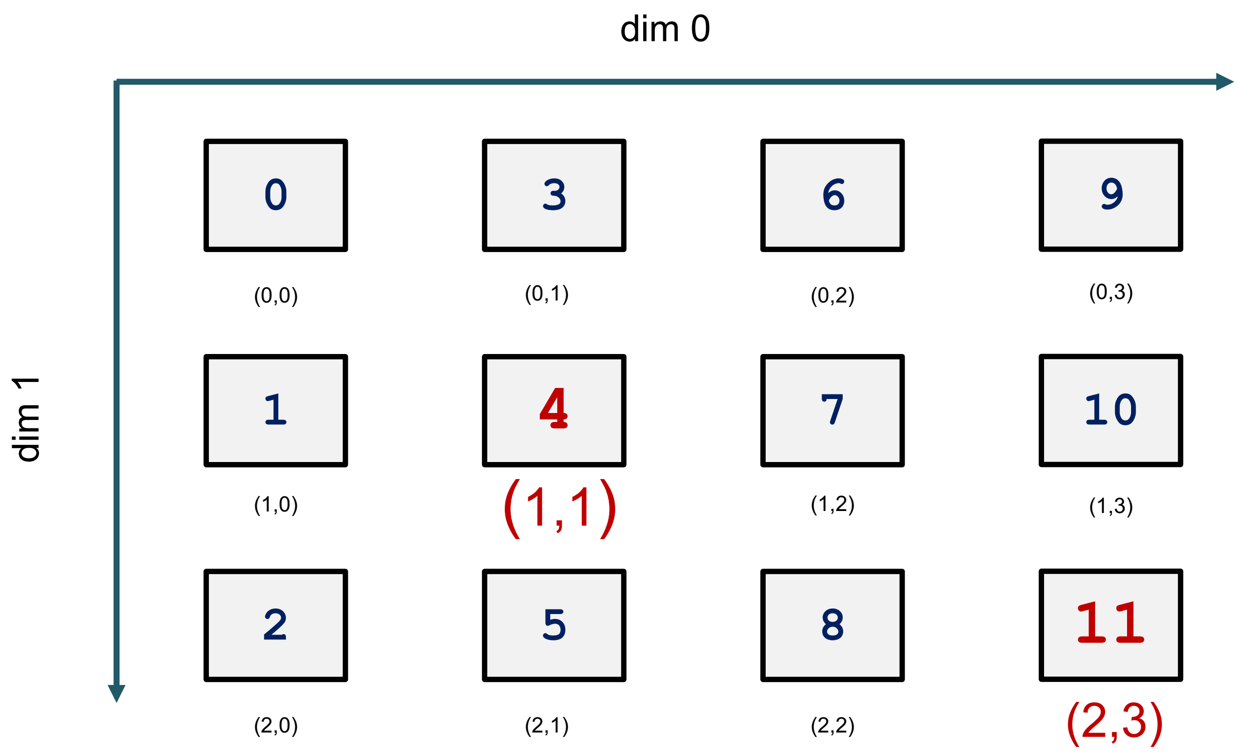
- Map coordinates → rank ("who is at that position? ")

```
MPI_Cart_rank(cart_comm, int coords[], int *rank);
```

`coords:` coordinates; if periodic in direction `i`, `coords[i]` are automatically mapped into the valid range, else they are erroneous

Example

- Example: 12 processes arranged on a 4 x 3 grid
- Process coordinates begin with 0



Next-neighbor communication

Sending/receiving from neighbors is a typical task in Cartesian topologies

```
MPI_Cart_shift(cart_comm, direction, disp,
               int *source_rank, int *dest_rank);
```

`direction`: dimension to shift

`disp`: offset to shift: > 0 shift in positive direction,
< 0 shift in negative direction

`source_rank/dest_rank`: returned ranks as input into `MPI_Sendrecv*` calls

Next-neighbor communication

Exampe: 4x3 process grid, periodic in 1st dimension, each process has an **`int`** value, which gets shifted

for non-periodic dimensions **`MPI_PROC_NULL`** is returned on boundaries

```
MPI_Cart_shift(cart_comm, 0, 1, &src, &dst);
MPI_Sendrecv_replace(&value, 1, MPI_INT,
                     dst, 0, src, 0, cart_comm, …)
```

9	0	3	6	9	0
10	1	4	7	10	1
11	2	5	8	11	2

Rank 0:
src: 9
dst: 3

Rank 4:
src: 1
dst: 7

shift in 1st dimension, which is periodic

```
MPI_Cart_shift(cart_comm, 1, 1, &src, &dst);
MPI_Sendrecv_replace(&value, 1, MPI_INT,
                     dst, 0, src, 0, cart_comm, …)
```

-2	-2	-2	-2
0	3	6	9
1	4	7	10
2	5	8	11
-2	-2	-2	-2

Rank 4:
src: 3
dst: 5

Rank 8:
src: 7
dst: MPI_PROC_NULL

shift in 2nd dimension, which is non-periodic

Putting it all together

Setup virtual topology for cartesian 2D grid:

```
int dims[2] = {0, 0};
int periods[2] = {0, 0};
MPI_Dims_create(size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Cart_shift(comm, 0, 1, &iNeighbours[0], &iNeighbours[1]);
MPI_Cart_shift(comm, 1, 1, &jNeighbours[0], &jNeighbours[1]);
MPI_Cart_get(comm, 2, dims, periods, coords);
```

Check if rank is at specific boundary:

```
if ( coords[1] == (dims[1]-1) ) { //set top bc
…
}
```

Derived MPI datatypes for ghost layers

- Compute size of local domain

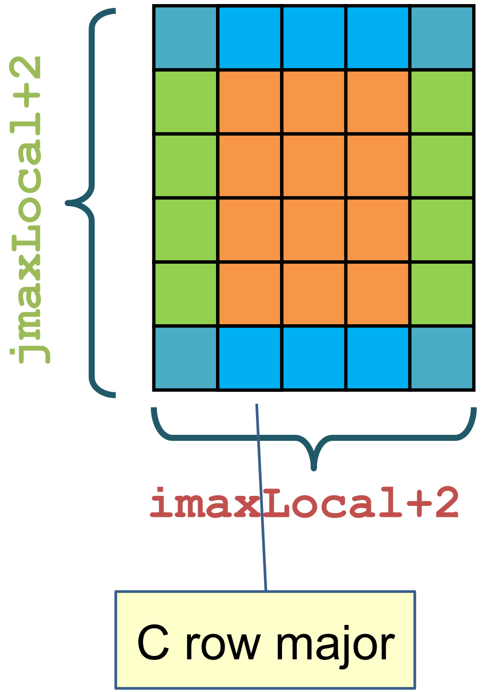
```
imaxLocal = sizeOfRank(rank, dims[0], imax);
jmaxLocal = sizeOfRank(rank, dims[1], jmax);
```

- Create type for contiguous dimension i

```
MPI_Type_contiguous(imaxLocal, MPI_DOUBLE,
                    &jBufferType);
MPI_Type_commit(&jBufferType);
```

- Create type for non-contiguous dimension j

```
MPI_Type_vector(jmaxLocal, 1, imaxLocal+2,
                MPI_DOUBLE, &iBufferType);
MPI_Type_commit(&iBufferType);
```



Example: Exchange boundary

```
double* buf[8];  MPI_Request requests[8];
for ( int i=0; i<8; i++ ) requests[i] = MPI_REQUEST_NULL;
buf[0] = grid + 1; //recv bottom
buf[1] = grid + (imaxLocal+2) + 1; //send bottom
// Setup send and receive buffers

for (int i=0; i<2; i++) {
     /* exchange ghost cells with bottom/top neighbor */
     MPI_Irecv(buf[i*2], 1, jBufferType, jNeighbours[i], 1,
            comm , &requests[i*2]);
     MPI_Isend(buf[(i*2)+1], 1, jBufferType, jNeighbours[i], 1,
            comm, &requests[i*2+1]);
     /* exchange ghost cells with left/right neighbor */
     MPI_Irecv(buf[i*2+4], 1, iBufferType, iNeighbours[i], 1,
            comm , &requests[i*2+4]);
     MPI_Isend(buf[i*2+5], 1, iBufferType, iNeighbours[i], 1,
            comm, &requests[(i*2)+5]);
}
MPI_Waitall(8, requests, MPI_STATUSES_IGNORE);
```

What else? Covered later in lecture!

Support for general graph topologies

- Constructors **`MPI_Graph_create`** and **`MPI_Dist_graph_create`**

- **`MPI_TOPO_TEST`** is used to query for the type of topology associated with a communicator

- **`MPI_GRAPHDIMS_GET`** and **`MPI_GRAPH_GET`** to retrieve graph topology information
- **`MPI_GRAPH_NEIGHBORS_COUNT`** and **`MPI_GRAPH_NEIGHBORS`** to obtain the neighbors of an arbitrary node in the graph
- **`MPI_DIST_GRAPH_NEIGHBORS_COUNT`** and **`MPI_DIST_GRAPH_NEIGHBORS`** to obtain the neighbors of the calling process

NHR FAU

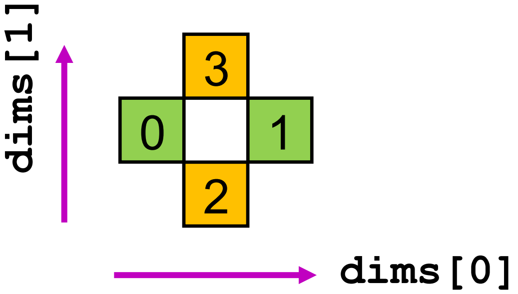
Friedrich-Alexander-Universität Erlangen-Nürnberg

Neighborhood Collective Communication on Process Topologies

MPI-3 feature: Neighborhood collective

- Nearest neighbor communication that can be expressed as MPI virtual topologies
- Opportunity for optimized mapping on network topology
- Collective operations, must be called by all processes in communicator

- With Cartesian topology the sequence of neighbors in buffers is defined by
  - order of the dimensions
  - first the neighbor in the negative direction
  - then in the positive direction



Simplest Neighborhood collective: allgather

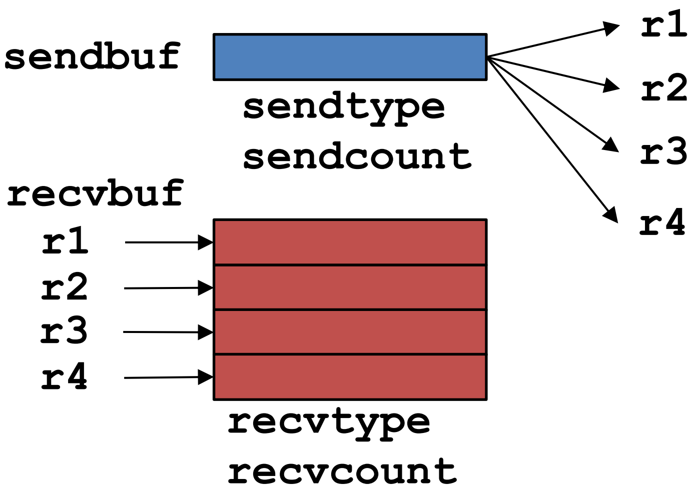
- For all direct neighbor ranks:
  Send the same data to all ranks
  Receive data from all ranks

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

`sendbuf, sendcount, sendtype`: Send data specification.
`recvbuf, recvcount, recvtype`: Receive data specification.
`recvcount`: number of elements received from each neighbor.
`comm`: communicator created with `MPI_Cart_create`

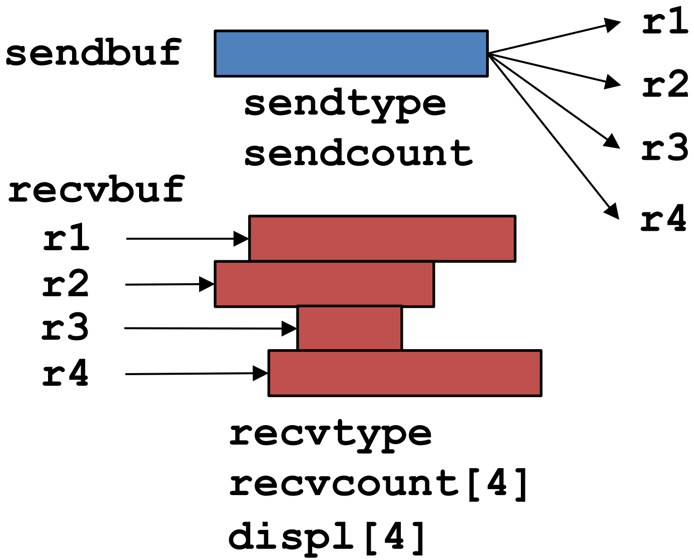
Neighborhood collective variations allgather

MPI_Neighbor_allgather



Contiguous chunks from each neighbor

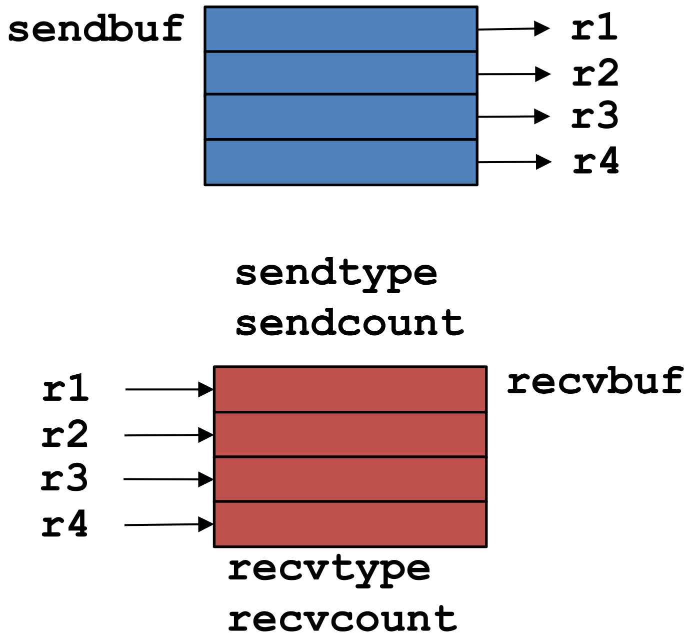
MPI_Neighbor_allgatherv



Non-contiguous variable-sized chunks from each neighbor

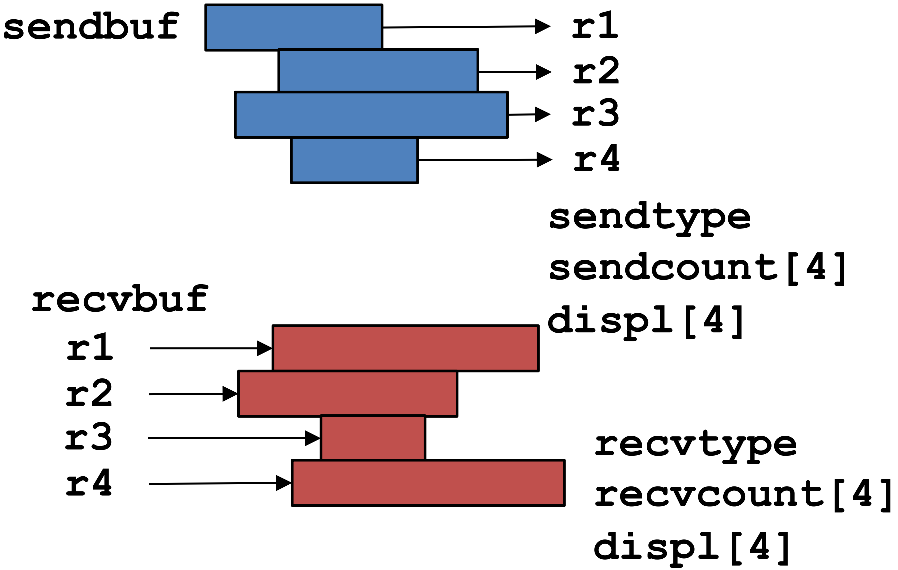
Neighborhood collective variations alltoall

MPI_Neighbor_alltoall



Contiguous chunks in send and recv buffers for each neighbor

MPI_Neighbor_alltoallv



Non-contiguous variable-sized chunks in send and recv buffers for each neighbor

Neighborhood collective `MPI_Neighbor_alltoallw`

```
MPI_Neighbor_alltoallw(
void* sendbuf, int sendcounts[], MPI_Aint sdispls[],
 MPI_Datatype sendtypes[],
void* recvbuf, int recvcounts[], MPI_Aint rdispls[],
 MPI_Datatype recvtypes[],
MPI_Comm comm);
```

`sendbuf, sendcounts, sdispls, sendtypes`: Send data specification.

`recvbuf, recvcounts, rdispls, recvtypes`: Receive data specification

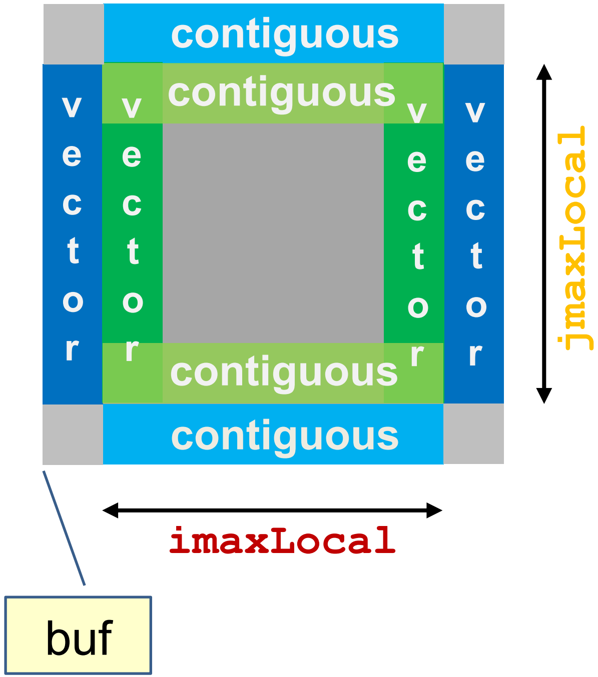
`comm`: communicator created with `MPI_Cart_create`

In bytes!

2D Halo exchange with `MPI_Neighbor_alltoallw`

- Communication setup

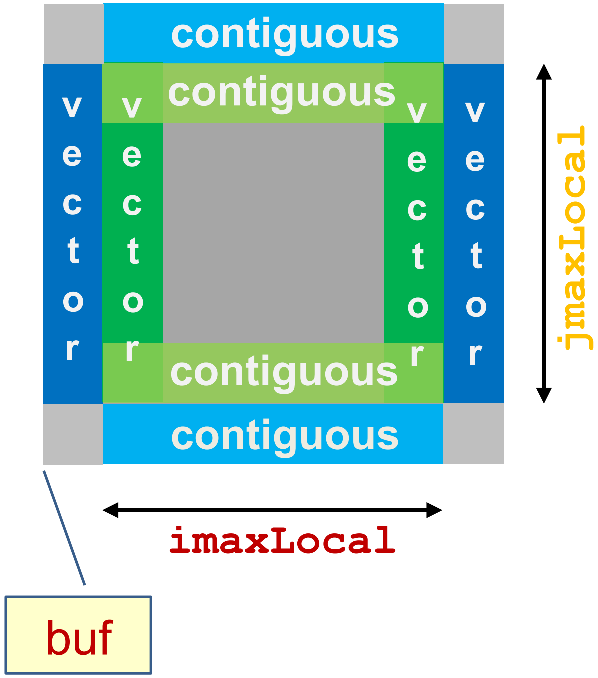
```
int counts[4] = {1, 1, 1, 1};

bufferTypes[0] = vectorType;     //left
bufferTypes[1] = vectorType;     //right
bufferTypes[2] = contiguousType; //bottom
bufferTypes[3] = contiguousType; //top

sdispls[0] = ((imaxLocal+2)+1)*dblsize;
sdispls[1] = ((imaxLocal+2)+imaxLocal)*dblsize;
sdispls[2] = ((imaxLocal+2)+1)*dblsize;
sdispls[3] = ((jmaxLocal)*(imaxLocal+2)+1)*dblsize;

rdispls[0] = (imaxLocal+2)*dblsize;
rdispls[1] = ((imaxLocal+2)+(imaxLocal+1)*dblsize;
rdispls[2] = 1*dblsize;
rdispls[3] = ((jmaxLocal+1)*(imaxLocal+2)+1))*dblsize;
```



2D Halo exchange with `MPI_Neighbor_alltoallw`

- Communication exchange

```
MPI_Neighbor_alltoallw(
buf, counts, sdispls, bufferTypes,
buf, counts, rdispls, bufferTypes, comm );
```



Simple debugging of communication

With small domain size:

1. Initialize array with rank id
2. Perform exchange once
3. Print array

```
20:   1.00000000    1.00000000    1.00000000    1.00000000    1.00000000    1.00000000
21:   1.00000000    1.00000000    1.00000000    1.00000000    1.00000000    1.00000000
### RANK 2 #######################################################
00:   2.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
01:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
02:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
03:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
04:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
05:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
06:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
07:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
08:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
09:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
                                        00    2.00000000    2.00000000    2.00000000
20:   0.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
21:   2.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
### RANK 3 #######################################################
00:   3.00000000    2.00000000    2.00000000    2.00000000    2.00000000    2.00000000
01:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
02:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
03:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
04:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
05:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
06:   1.00000000    3.00000000    3.00000000    3.00000000    3.00000000    3.00000000
```

```
for ( int i=0; i<(imaxLocal+2)*(jmaxLocal+2); i++ ) {
        buf[i] = rank;
}
exchange(buf);
print(buf);
```

Debug output of distributed array

```
for ( int i=0; i < size; i++) {
    if ( i == rank ) {
       printf("### RANK %d ####################\n", rank);
       for( int j=0; j < jmaxLocal+2; j++ ) {
           printf("%02d: ", j);
           for( int i=0; i < solver->imaxLocal+2; i++ ) {
               printf("%12.8f  ", grid[j*(imaxLocal+2) + i]);
           }
           printf("\n");
       }
       fflush( stdout );
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```