

# HPC.NRW

# MPI in Small Bites

## PPCES 2024

HPC.NRW Competence Network



**EDIH**  
Rheinland



THE COMPETENCE NETWORK FOR HIGH PERFORMANCE COMPUTING IN NRW.

# Derived Datatypes

HPC.NRW Competence Network

## MPI in Small Bites

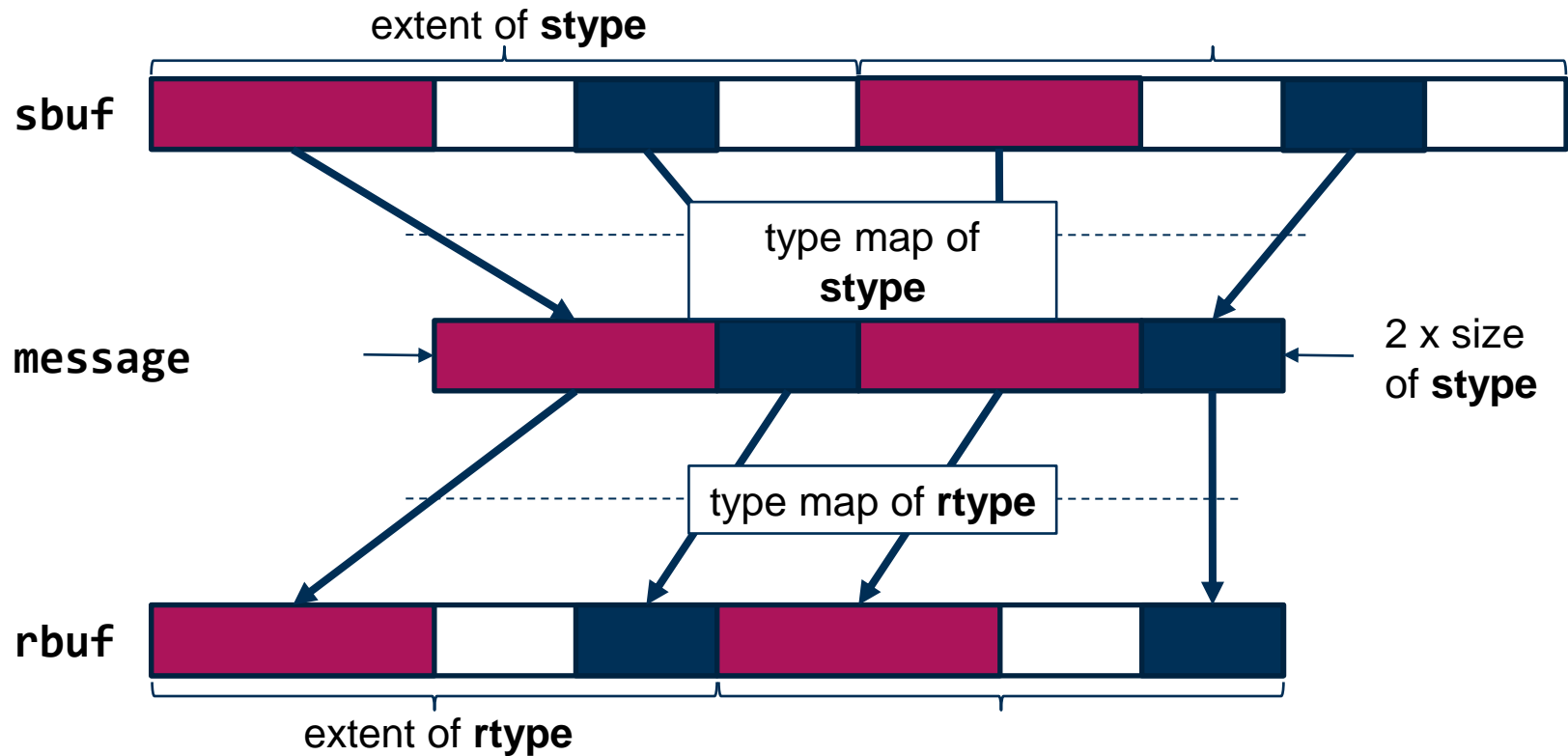
- MPI datatypes are opaque handles
  - Instructions for accessing the binary content of a memory buffer
- MPI predefines basic datatypes for each language binding
  - Distinct handles for C/C++ and Fortran (e.g., MPI\_INT vs. MPI\_INTEGER)
  - Describe a single data element
- More complex MPI datatypes can be constructed (derived)
  - Matrix rows & columns, diagonal matrices, structures

- Type signature
  - A sequence of basic datatypes described by a given type and count
  - Example: {MPI\_INT, MPI\_INT, MPI\_DOUBLE}
- Type map
  - A sequence of basic datatypes and their displacements
  - Example: {(MPI\_INT, 0), (MPI\_INT, 4), (MPI\_DOUBLE, 8)}
- Datatypes are local objects
  - May differ across processes
  - Enable transparent type marshalling (encoding & decoding of data)

- Lower and upper bound:
  - $lb(\text{datatype}) = \min disp_j$
  - $ub(\text{datatype}) = \max (disp_j + \text{sizeof}(type_j)) + \text{padding}$
- Extent
  - $extent(\text{datatype}) = ub(\text{datatype}) - lb(\text{datatype})$
  - The size of the step when accessing consecutive elements of that type
- Size
  - $size(\text{datatype}) =$
  - The total amount of bytes taken by the datatype, not counting any gaps in it

- Example: MPI\_INT
  - *type map* = { (MPI\_INT, 0) }
  - *lb* = 0
  - *ub* = 4
  - *extent* = 4 bytes
  - *size* = 4 bytes
- All predefined basic MPI datatypes have lower bound 0, i.e. data is flush with the buffer start
- Platform-specific alignment rules are taken into account
  - The upper bound is therefore adjusted if necessary

```
MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);
```



```
MPI_Recv(rbuf, 2, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- Create a sequence of elements of an existing datatype

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

- The new datatype represents a contiguous sequence of **count** elements of **oldtype**
- The elements are separated from each other by the extent of **oldtype**
- A send/receive of one element of **newtype** is congruent with a receive/send of **count** elements of **oldtype**
- Useful for sending entire matrix rows (C/C++) or columns (Fortran)



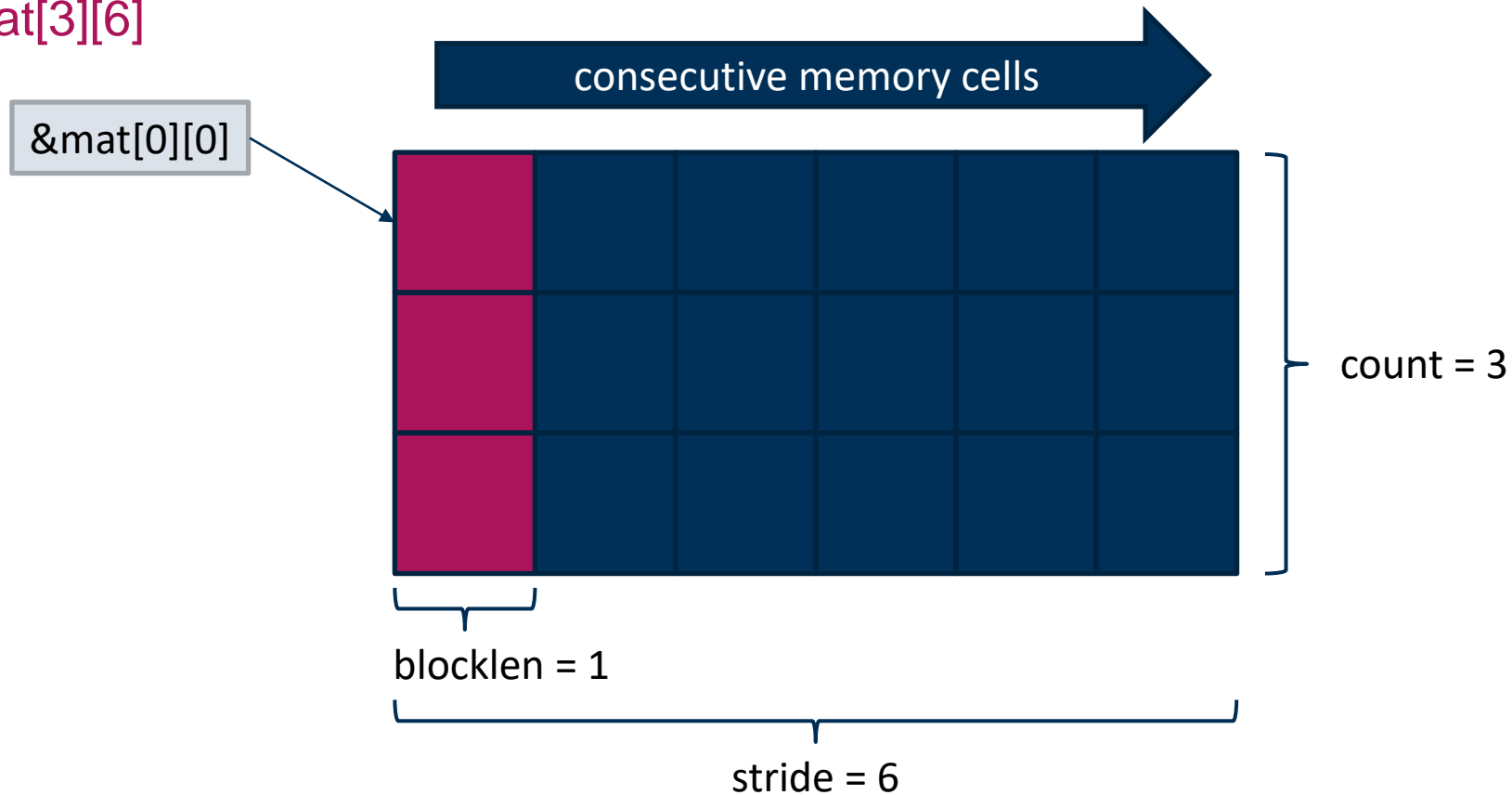
- Create a sequence of equally spaced blocks of elements

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

- The new datatype represents a sequence of **count** blocks, each containing **blocklen** elements of the old datatype
- Every two consecutive blocks are separated by **stride** elements each
- Useful for sending matrix columns (C/C++) or rows (Fortran)
  - **stride** = row (C/C++) | column (Fortran) length (in number of elements)
  - **blocklen** = 1 (or the number of consecutive rows/columns)
  - **count** = number of rows (C/C++) | columns (Fortran)

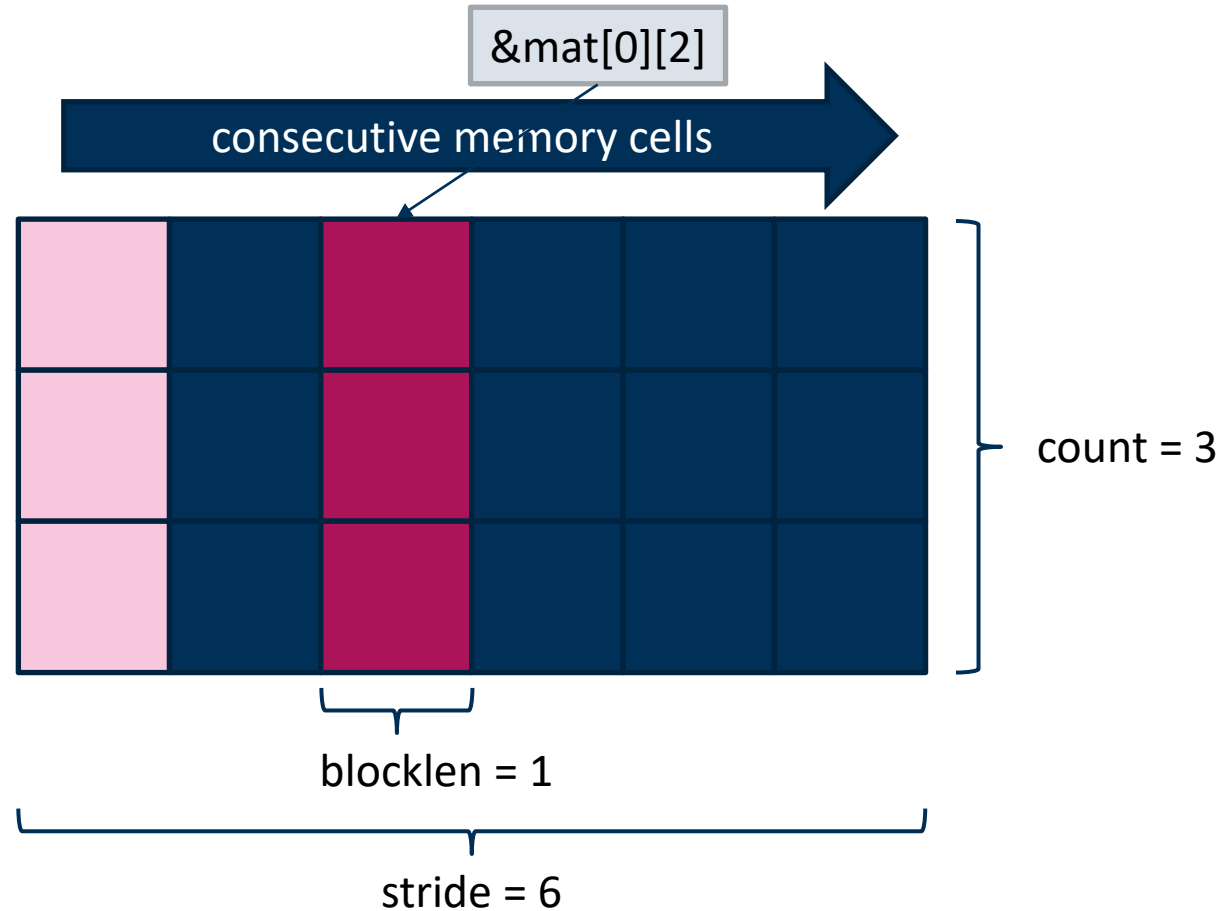
– Example: single column of a C/C++ matrix

– `mat[3][6]`



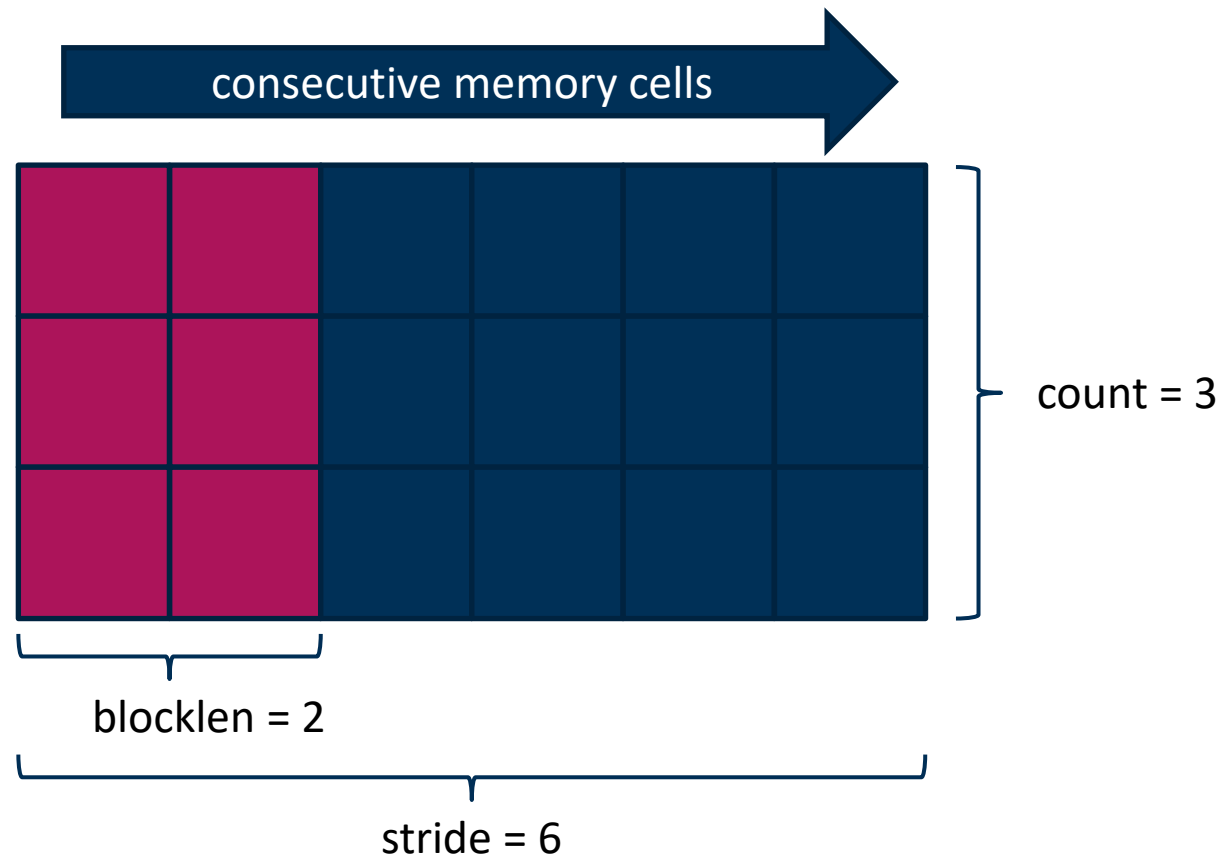
– Example: single column of a C/C++ matrix

– `mat[3][6]`



– Example: two consecutive columns of a C/C++ matrix

– `mat[3][6]`



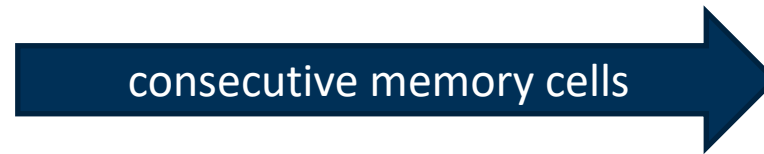
- Indexed list of arbitrary-sized blocks within a matrix

```
MPI_Type_indexed (int count, const int array_of_blocklengths[],  
                  const int array_of_displacements[], MPI_Datatype oldtype,  
                  MPI_Datatype *newtype)
```

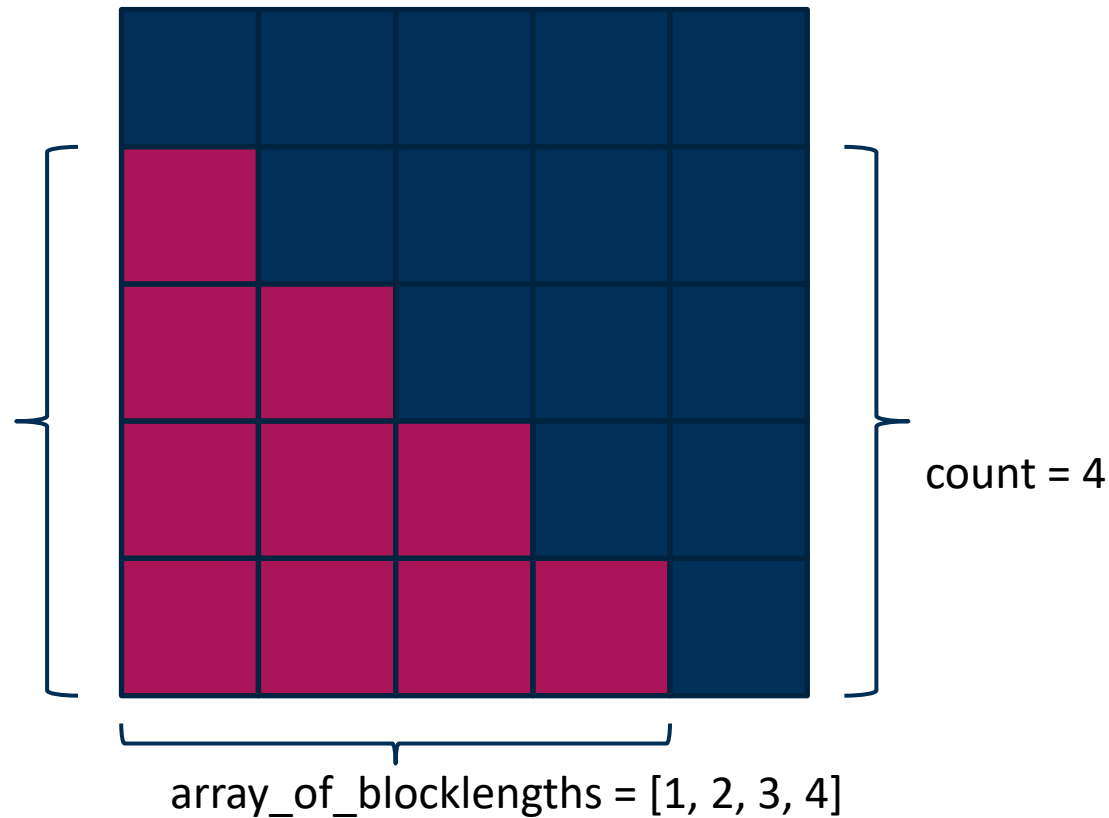
- The new datatype represents a sequence of **count** blocks, where the *i*-th block
  - contains **array\_of\_blocklengths[i]** elements of the old datatype and
  - Every two consecutive blocks are separated by **stride** elements each
- Useful for custom shapes of data in fixed-sized matrices
  - **Lower/upper triangle**
    - Blocksizes increase with every block

- Example: lower triangle of fixed size matrices

- `mat[5][5]`



array\_of\_displacements  
= [5, 10, 15, 20]

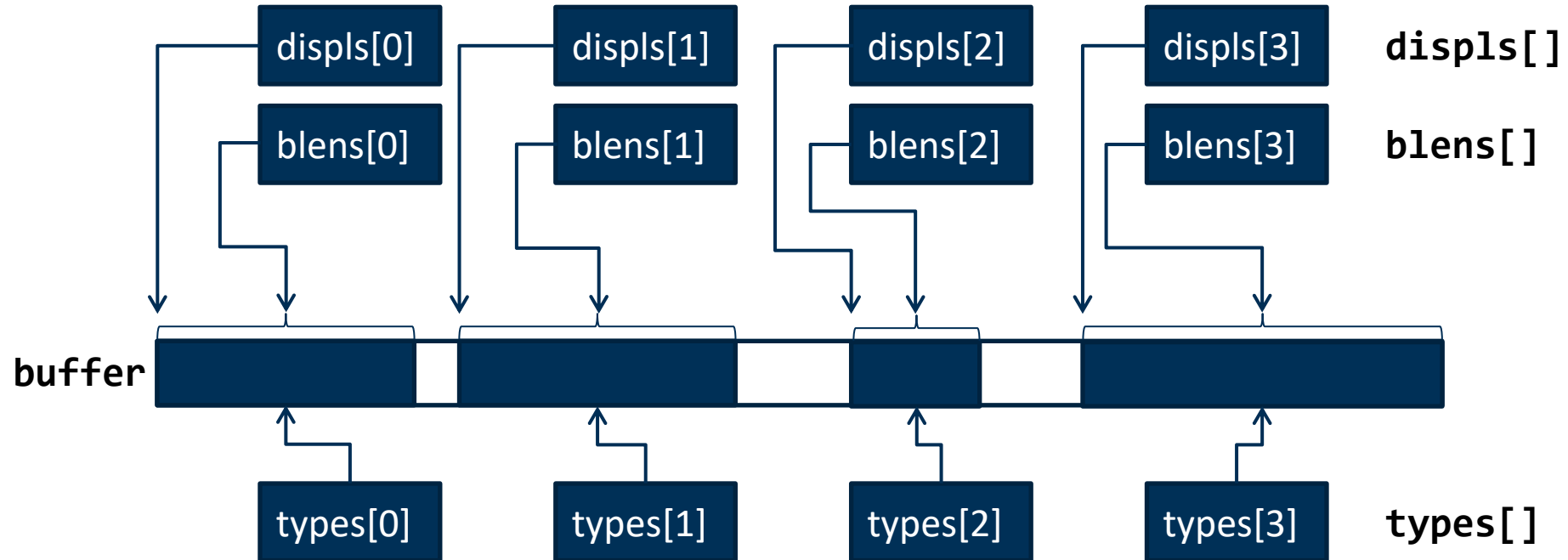


- The most generic datatype
  - Useful for C/C++ structures and Fortran derived data type / COMMON blocks

```
MPI_Type_create_struct (int count, const int array_of_blocklengths[],  
                        MPI_Aint array_of_displacements[],  
                        MPI_Datatype array_of_types[],  
                        MPI_Datatype *datatype)
```

- **count:** number of blocks in the datatype
- **array\_of\_blocklengths[]:** number of elements in each block
- **array\_of\_displacements[]:** displacement in bytes from the start of each block
- **array\_of\_types[]:** datatype of the elements in each block
- **datatype:** handle of the new datatype

```
MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],  
MPI_Datatype types[], MPI_Datatype *datatype)
```





- The most generic datatype
  - Corresponds to C/C++ struct

```
typedef struct {  
    float mass;  
    double pos[3];  
    char sym;  
} Particle;  
  
int blens[] = { 1, 3, 1 };  
MPI_Aint displs[] = { offsetof(Particle, mass),  
                     offsetof(Particle, pos),  
                     offsetof(Particle, sym) };  
MPI_Datatype types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };  
  
MPI_Datatype particle_type;  
MPI_Type_create_struct(3, blens, displs, types, &particle_type);
```

- Register a datatype for use with communication operations:

```
MPI_Type_commit (MPI_Datatype *datatype)
```

- A datatype must be committed before it can be used in communications
  - All predefined datatypes are already committed
  - Intermediate datatypes, i.e. ones used for building more complex datatypes but not used in communication, can be left uncommitted
- Deregister and free a datatype:

```
MPI_Type_free (MPI_Datatype *datatype)
```

- Derived datatypes, build from the freed datatype, are not affected
- **datatype** set to **MPI\_TYPE\_NULL** upon successful return

```
typedef struct {  
    float mass;  
    double pos[3];  
    char sym;  
} Particle;  
  
int blens[] = { 1, 3, 1 };  
MPI_Aint displs[] = { offsetof(Particle, mass),  
                     offsetof(Particle, pos),  
                     offsetof(Particle, sym) };  
MPI_Datatype types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };  
  
MPI_Datatype particle_struct;  
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);  
MPI_Type_commit(&particle_struct);
```

- `particle_struct` can now be used to send a single Particle

- Resize to the true size of the structure

```
int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                    offsetof(Particle, pos),
                    offsetof(Particle, sym) };
MPI_Datatype types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Datatype particle_struct;
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);
// No need to commit particle_struct - not used in communication

MPI_Aint true_extent = sizeof(Particle);
MPI_Type_create_resized(particle_struct, 0, true_extent, &particle_type);
MPI_Type_commit(&particle_type);
```

- MPI\_Type\_create\_resized takes an existing datatype and creates a new one with modified lower bound and extent

- Local handles describing how to access memory
- Can be mixed and matched on both sides of a communication operation as long as their type signatures match
- Lower and upper bound can be manipulated to account for padding at beginning and end
- Need to be committed before use in communication or I/O
- Language-specific handles need to be used in mixed-language applications