

Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

Lecture 12:

- Distributed-Memory Programming with MPI
 - Derived Datatypes

MPI: Datatype

MPI Datatypes are opaque objects called **handles**.

- Used to interpret and access the contents of a memory buffer

MPI defines **primitive datatypes** that describe **a single data element**.

MPI data type	C native data type
MPI data type	C native data type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_INT	unsigned int
...	...
MPI_BYTE	

More complex datatypes can be **derived** from these primitive datatypes.

MPI: Datatype

- We often want to exchange messages that
 - contain values with different datatypes such as a **C struct**
 - are non-contiguous such as a block of matrix elements
- One naïve solution to sending non-contiguous data is by packing the non-contiguous data into a contiguous memory buffer at the sender and unpacking at the receiver.
 - This might require additional memory-to-memory copy operations at both sides.
- Rather, MPI allows specifying **more general, mixed-typed and non-continuous** memory buffers.
 - It is up to the given MPI implementation to decide whether data should be first packed into a contiguous buffer before being transmitted or whether it can be collected directly from where the data is kept.

MPI: Type Signature and Type Map

A **general datatype** is a general object that describe two things as follows:

- A sequence of basic datatypes
- A sequence of byte displacements

The **type signature** of a datatype is a sequence of datatypes described by a given type and count:

- e.g., {MPI_INT, MPI_INT, MPI_DOUBLE}, which describes the datatype as a sequence of two integers followed by one double.

The **type map of a datatype** is a sequence of basic datatypes and their displacements

- e.g., {(MPI_INT,0),(MPI_INT,4),(MPI_DOUBLE,8)}

Datatypes are local objects:

- They may differ across MPI processes
- Each process performs the marshalling and unmarshaling of messages in a transparent manner.

MPI: Terminology

Lower and Upper Bound

- $lb(datatype) = \min(displ_i)$
- $ub(datatype) = \max(displ_i + sizeof(type_i)) + padding$

Extent

- $extent(datatype) = ub(datatype) - lb(datatype)$
- The extent of a datatype is the step size in bytes when accessing consecutive elements of that datatype

Size

- $size(datatype) = \sum_i sizeof(type_i)$
- The size of a datatype is the amount in bytes taken by the datatype, **excluding any gaps in it**.

MPI: Basic Datatype

Example: MPI_DOUBLE

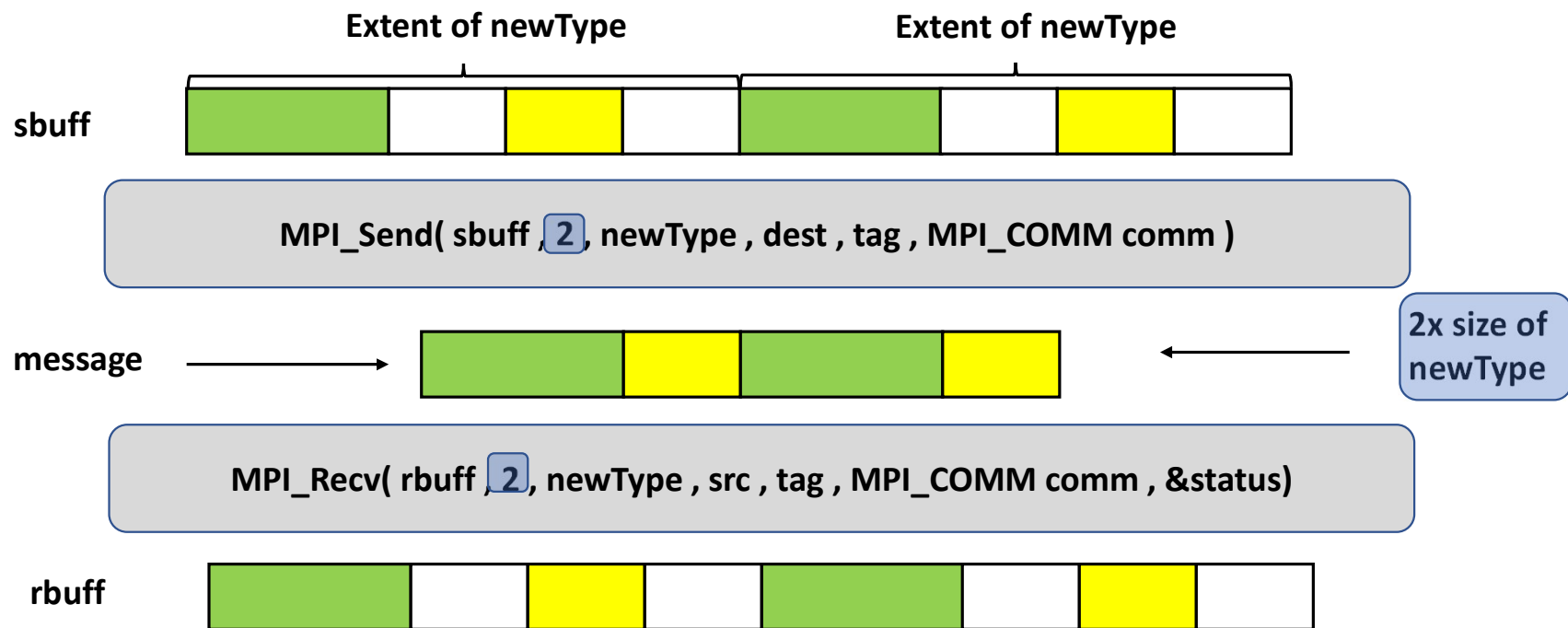
Type map = {(MPI_DOUBLE,0)}

- $lb(MPI_DOUBLE) = 0$
- $ub(MPI_DOUBLE) = sizeof(double) = 8$
- $extent(MPI_DOUBLE) = ub - lb = 8$
- $size(MPI_DOUBLE) = 8$

All primitive MPI datatypes have a lower bound of 0

The upper bound may be adjusted appropriately, depending on alignment constraints on the programming language and the machine.

MPI: What it looks like in action



MPI: Contiguous Datatypes

Create a sequence of elements of an existing datatype

C

```
MPI_Type_contiguous( int count , MPI_Datatype oldType , MPI_Datatype * newType)
```

- The new datatype *newType* represents a contiguous sequence of *count* elements of *oldType*.
- The elements (of *newType*) are separated from each other by the **extent** of *oldType*.
- Useful for sending entire rows (C/C++) or columns (Fortran) of matrices

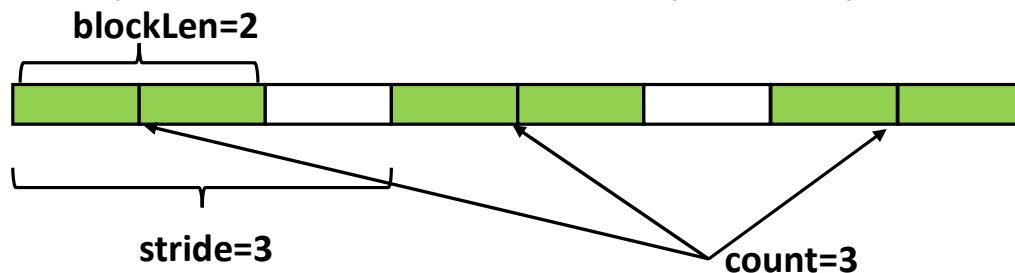
MPI: Vector Datatypes

Create a sequence of equally spaced blocks of elements of an existing datatype

C

```
MPI_Type_vector( int count , int blockLen , int stride , MPI_DataType oldType , MPI_Datatype * newType)
```

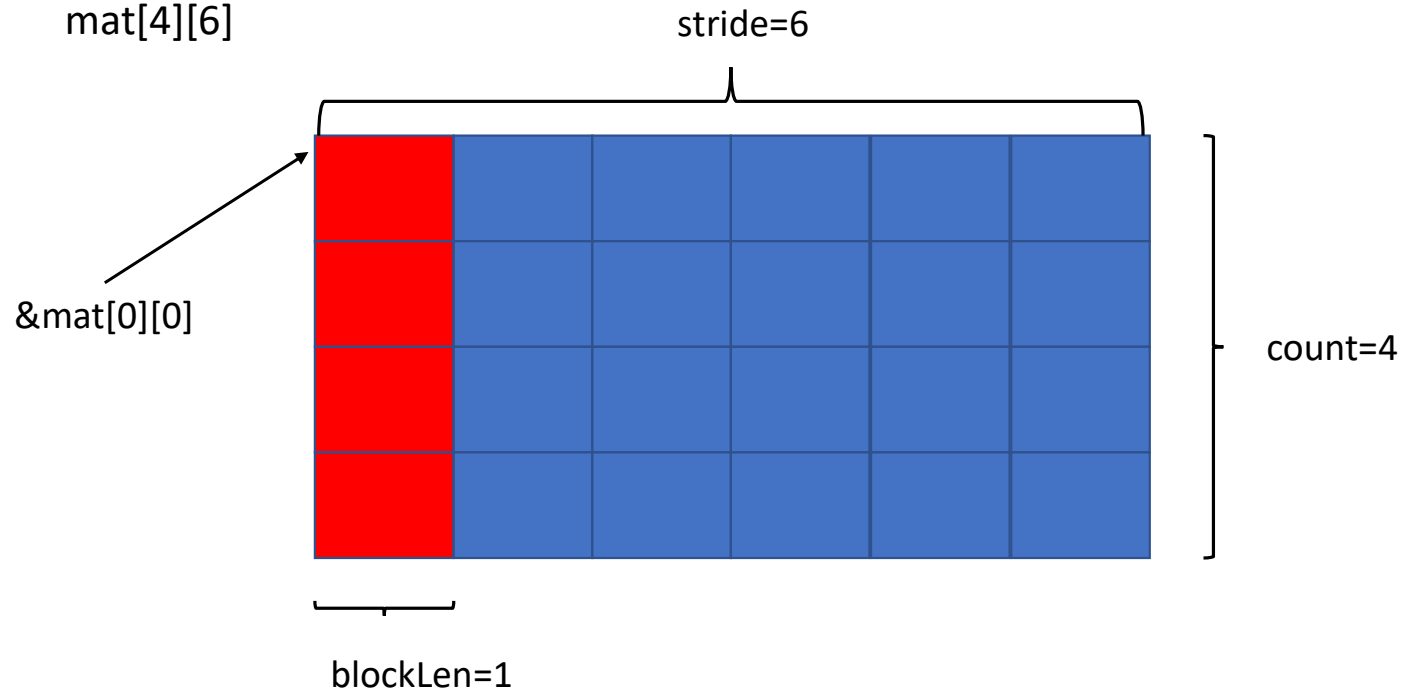
- The new datatype *newType* represents a sequence of *count* blocks, each of which contains *blockLen* elements of *oldType*.
- Every two consecutive blocks are separated by *stride* elements of *oldType*.



MPI: Vector Datatypes

Example: one column of a C/C++ matrix with 4 rows and 6 columns

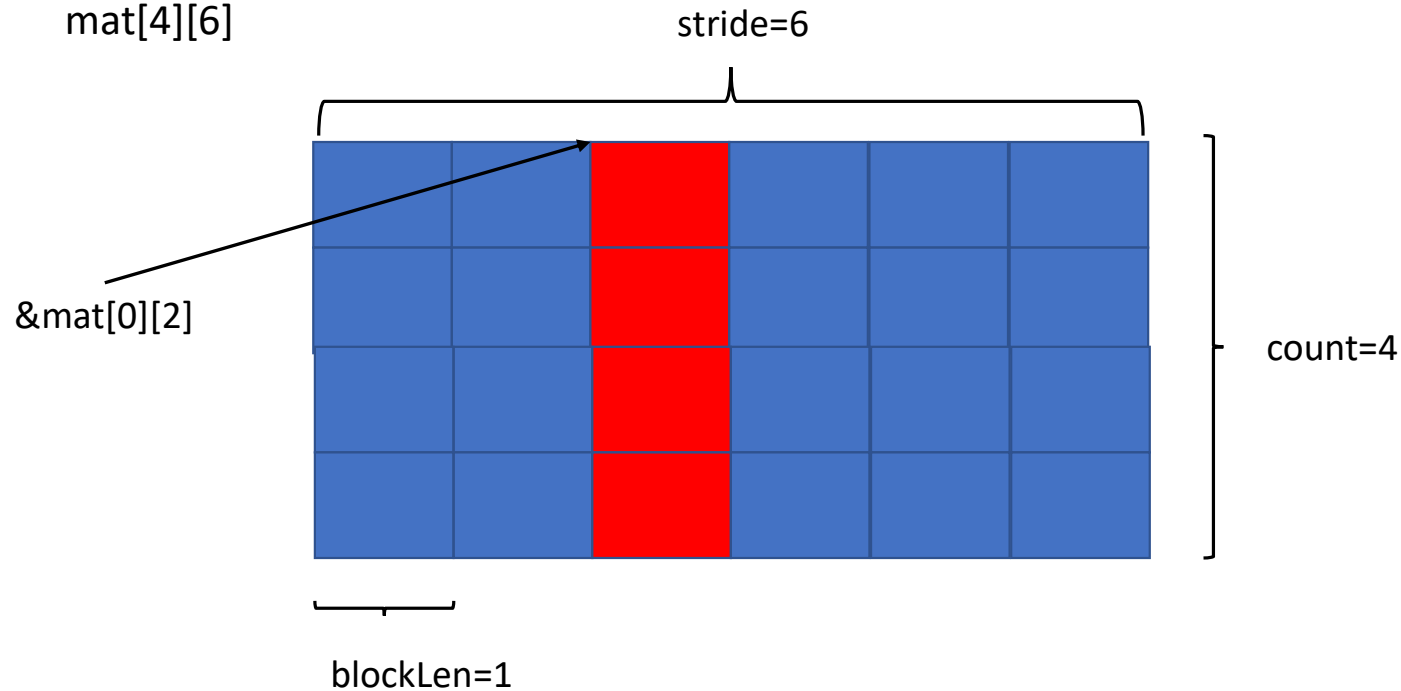
`mat[4][6]`



MPI: Vector Datatypes

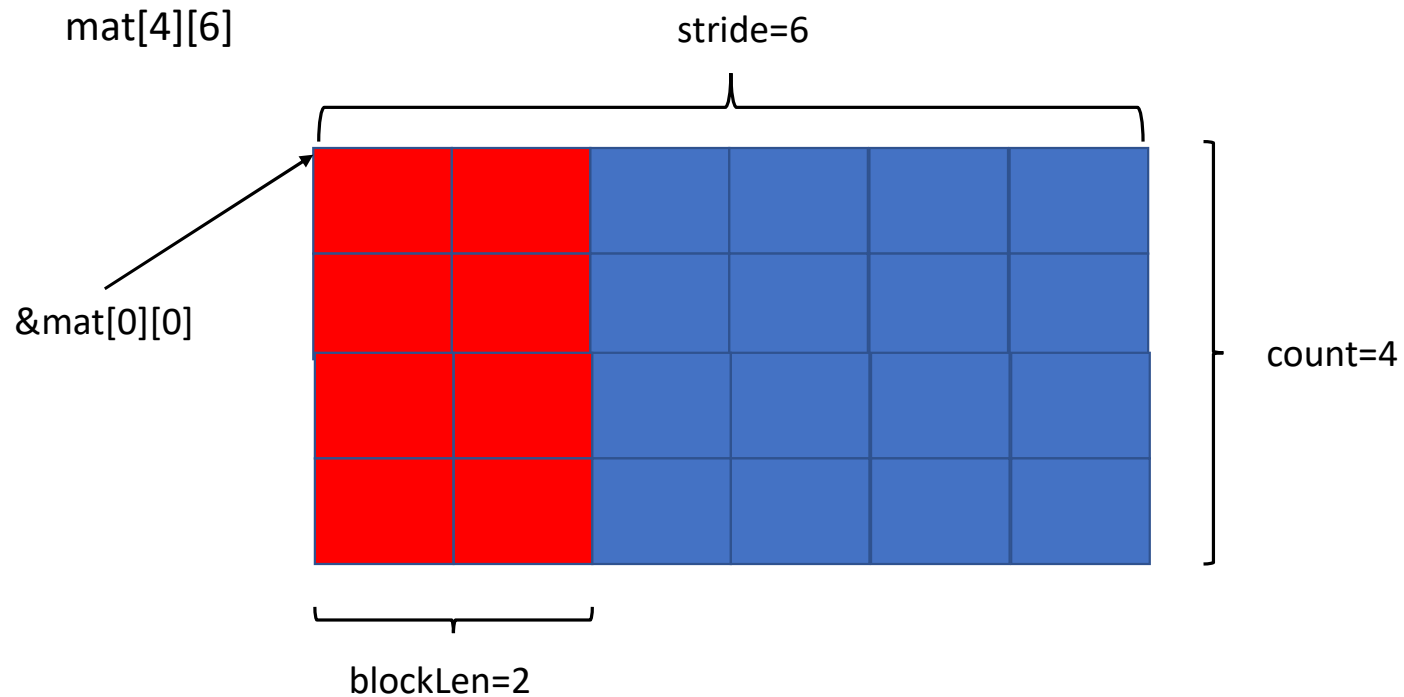
Example: one column of a C/C++ matrix with 4 rows and 6 columns

`mat[4][6]`



MPI: Vector Datatypes

Example: two columns of a C/C++ matrix with 4 rows and 6 columns



MPI: Indexed Datatypes

Create an 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)
```

C

The new datatype **newType** represents a sequence of **count** blocks, where the i^{th} block

- contains an **array_of_blocklengths[i]** elements of the old datatype **oldType** and
- Every two consecutive blocks are separated by **stride** elements each.

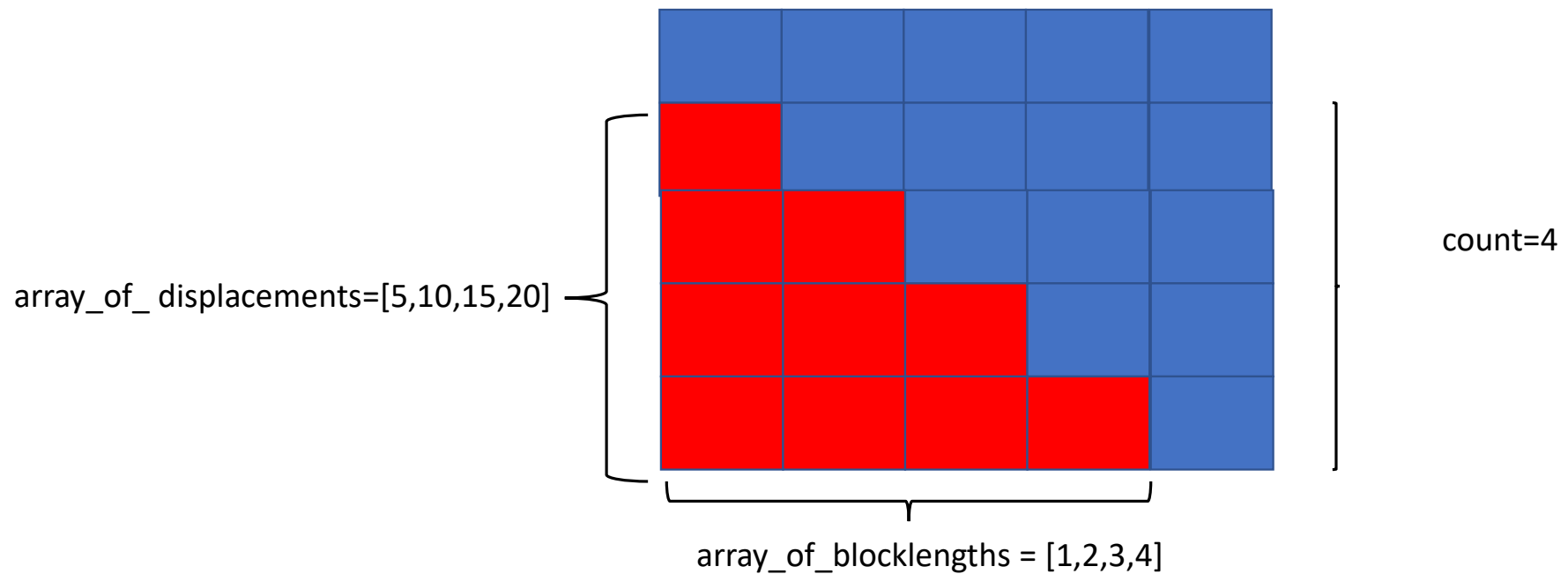
Useful for custom-shaped data of matrices with a fixed dimension

- Lower or upper triangular matrices

MPI: Indexed Datatypes

Example: lower triangle of a matrix of dimension 5 x 5

`mat[4][6]`



MPI: Structure Datatypes

The most generic datatype constructor

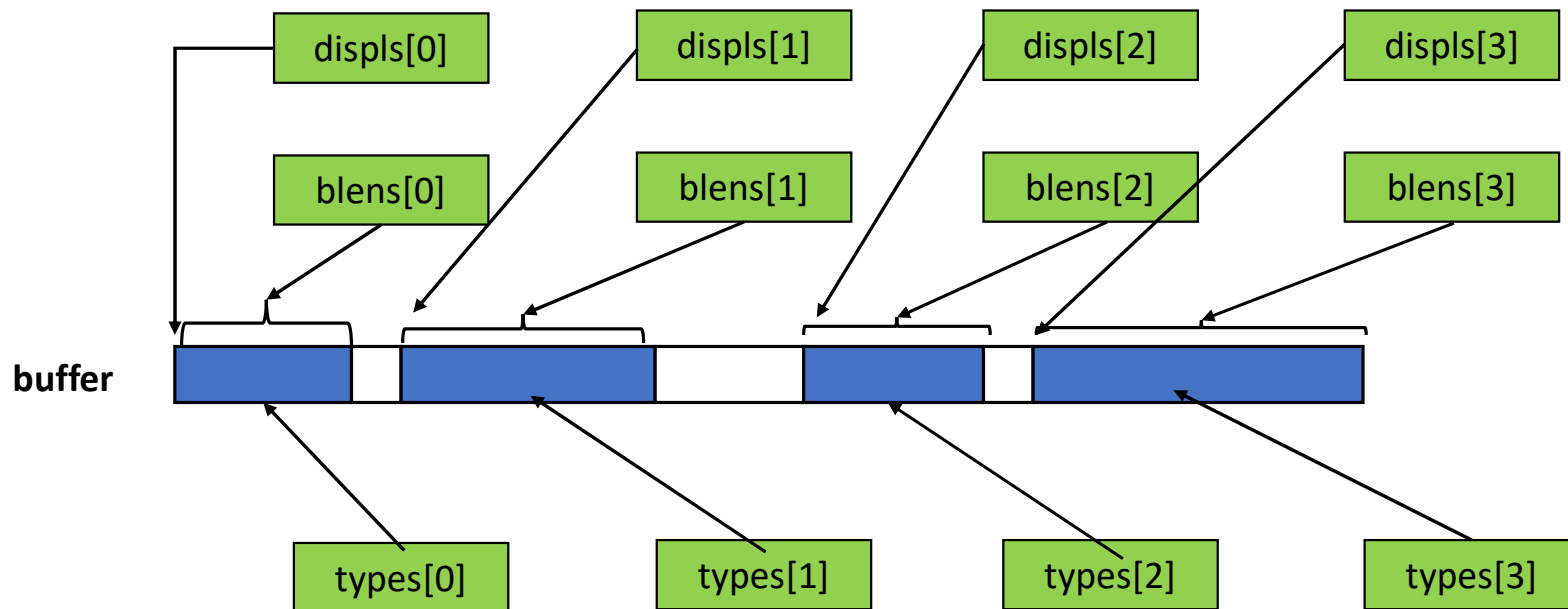
- Useful for C/C++ structures

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

C

- **count**: the number of blocks in the datatype
- **array_of_blocklengths**: the number of elements in each block
- **array_of_displacements**: the displacement in bytes from the start of each block
- **array_of_types**: the datatype of the elements in each block
- **newType**: handle of the new datatype

MPI: Structure Datatypes



MPI: Structure Datatypes

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

MPI: Registering Derived Datatypes

Register a derived datatype for using with communication operations

```
MPI_Type_commit( MPI_Datatype * newType)
```

C

- Datatypes need to be committed before they can be used in communication operations.
- All primitive datatypes, e.g., MPI_INT, MPI_FLOAT etc., are already committed.
- Intermediate datatypes defined by the user that are used to implement more complex datatypes but are not used in communication operations **can be left uncommitted**.

MPI: Registering Derived Datatypes

Deregister and free up a derived datatype

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

C

- Derived datatypes that were previously derived from the freed datatype are unaffected.
- Upon a successful return, *datatype* is set to **MPI_TYPE_NULL**.

MPI: Structure Datatypes

```
typedef struct{
    float mass;
    double pos[3];
    char sym;
}Particle;

int blens = { 1 , 3 , 1 };

MPI_Aint displs [3] = { offsetof(Particle,mass) , offsetof(Particle,pos) , offsetof(Particle,sym)};

MPI_Datatype types[3] = { MPI_FLOAT , MPI_DOUBLE , MPI_CHAR };
MPI_Datatype particleType;
MPI_Type_create_struct( 3 , blens , displs , types , &particleType );
MPI_Type_commit(&particleType); //Here we commit the new datatype
```

MPI: Structure Datatypes

The handle *particleType* can now be used to send and receive ONE element of *Particle*.



```
typedef struct{
    float mass;
    double pos[3];
    char sym;
}Particle;

int blens = { 1, 3, 1 };

MPI_Aint displs [3] = { offsetof(Particle,mass), offsetof(Particle,pos), offsetof(Particle,sym)};

MPI_Datatype types[3] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Datatype particleType;

MPI_Type_create_struct(3, blens, displs, types, &particleType);

MPI_Type_commit(&particleType); //Here we commit the new datatype
```

MPI: Structure Datatypes

We need to resize to the actual size of the structure.

```
typedef struct{
    float mass;
    double pos[3];
    char sym;
}Particle;

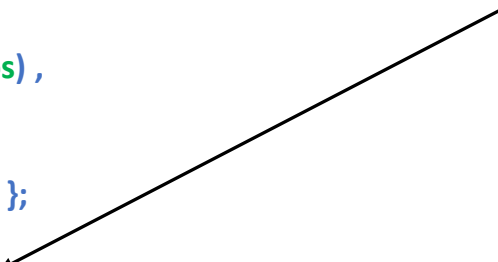
int blens = { 1 , 3 , 1 };

MPI_Aint displs [3] = { offsetof(Particle,mass) , offsetof(Particle,pos) ,
offsetof(Particle,sym)};

MPI_Datatype types[3] = { MPI_FLOAT , MPI_DOUBLE , MPI_CHAR };
MPI_Datatype intermediateType;
MPI_Type_create_struct( 3 , blens , displs , types , &intermediateType );

MPI_Aint true_extent = sizeof(Particle);
MPI_Type_create_resized(intermediateType,0,true_extent,&particleType);
MPI_Type_commit(&particleType);
```

Note that there is no need to commit the intermediate type unless it will be used in communication operations.



References

- [1] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.
- [2] Marc-Andre Hermanns. 2021. *MPI in Small Bites*. PPCES 2021.