NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

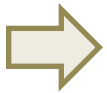
Erlangen National High Performance Computing Center

MPI Derived Datatypes

MPI data types: why?

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>,…);
```

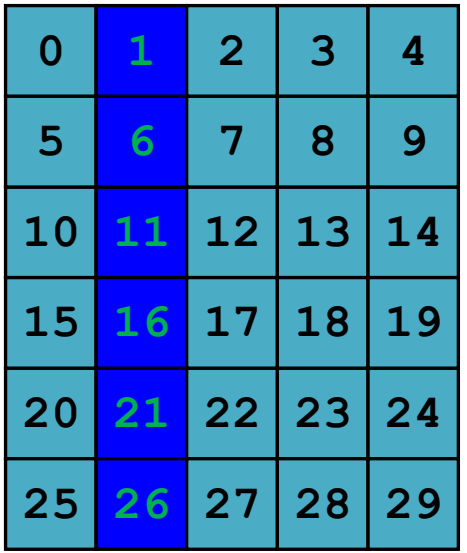
```
MPI_Bcast(&cfg, sizeof(cfg),
          MPI_BYTE, ..)
```

is **not** a solution. Its not portable as no data conversion can take place

MPI is supposed to support parallel computations across heterogeneous environments and communication in such environments may require data conversions.

MPI data types: why?

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



Making 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

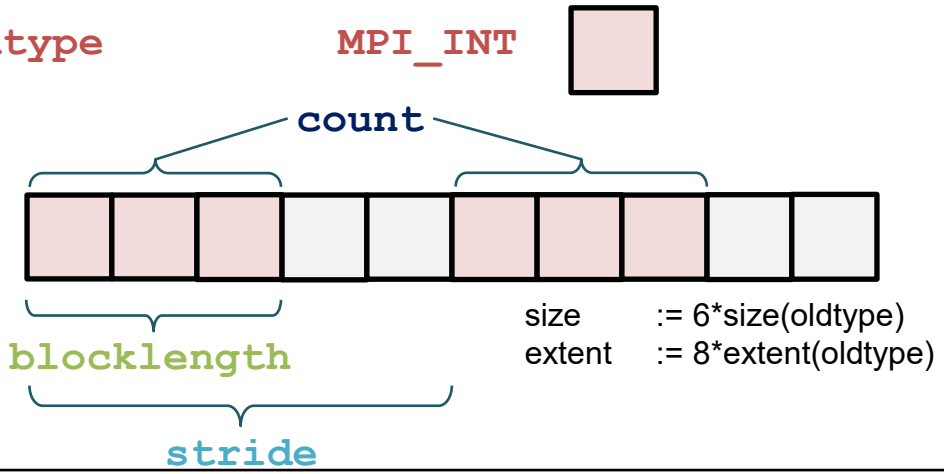
```
MPI_Type_free(MPI_Datatype * nt);
```

All local, non-collective calls

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



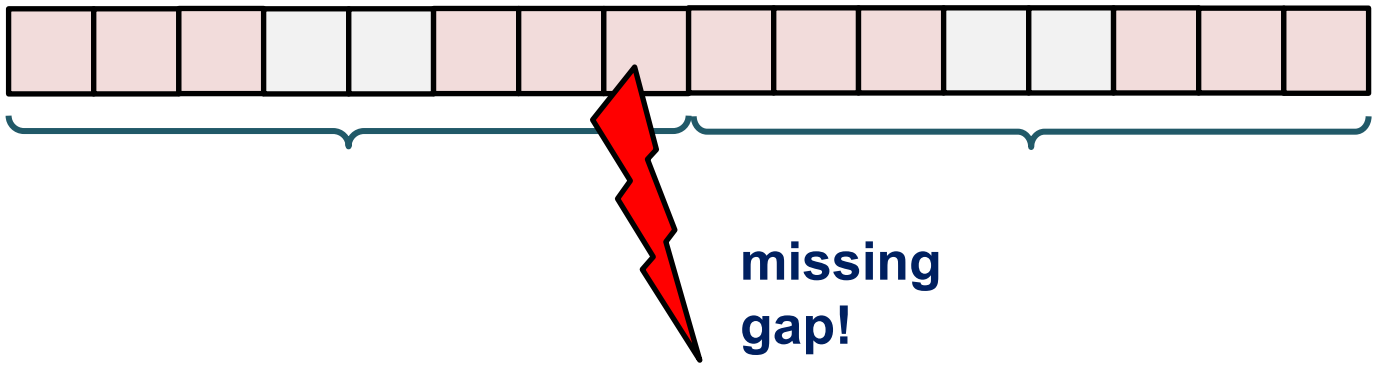

```
MPI_Datatype nt;
MPI_Type_vector(
2, 3, 5, MPI_INT, &nt);

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

Caveat when using a type

- Caution: Concatenating such types in a send operation can lead to unexpected results!
- **`count`** argument to **`send`** and others must be handled with care:

`MPI_Send(buf, 2, nt,...)` with **`nt`** (newtype from prev. slide)



Derives type size and extent

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

```
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

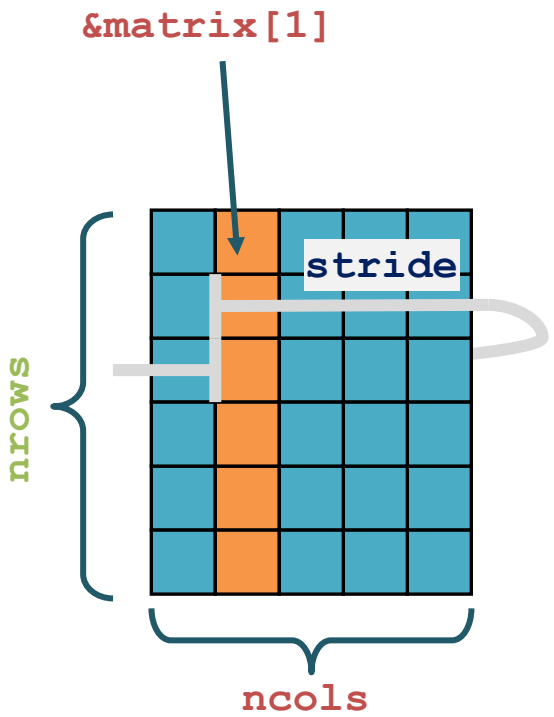
```
int MPI_type_get_extent(MPI_Datatype newtype,
                        MPI_Aint *lb,
                        MPI_Aint *extent);
```

MPI type for memory addresses or offsets

- MPI allows to change the extent of a datatype using
  - **`MPI_Type_create_resized`**
    - **`Sizeof`**
    - **`MPI_Get_address/MPI_Aint_diff`**

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);
```

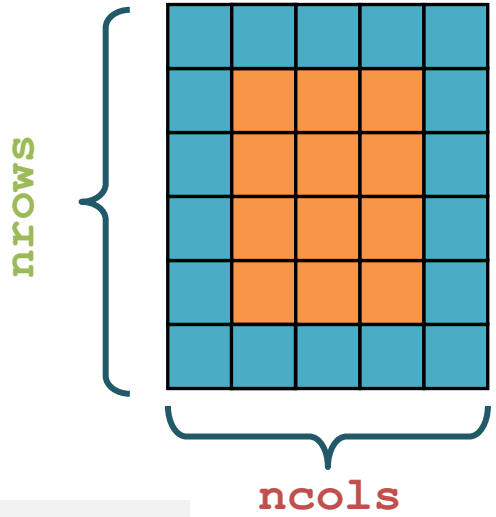
A sub-array type: MPI_Type_create_subarray

```
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

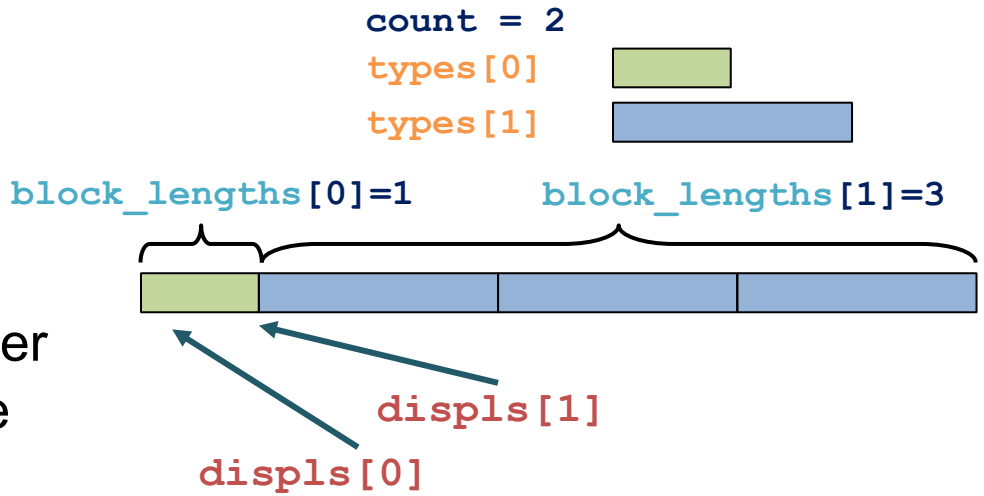



```
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
                         ar_starts, order, oldtype, &nt);
MPI_Type_commit(&nt);
// use nt...
MPI_Send(&buf[0], 1, nt, …); // etc.
MPI_Type_free(&nt);
```

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

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)

- When using absolute addresses, set buffer address = **`MPI_BOTTOM`**

Derived data types: summary

- A flexible tool to communicate complex data structures in MPI
- Most important calls:

```
MPI_Type_vector                (second simplest)
MPI_Type_create_subarray
MPI_Type_create_struct         (most advanced)
MPI_Type_commit/MPI_Type_free
MPI_Get_address,
  MPI_Aint_add, MPI_Aint_diff
MPI_Type_get_extent, MPI_Type_size
```

- Other useful features:
  `MPI_Type_contiguous`, **`MPI_Type_indexed`**,…
- Matching rule: send and receive match if specified basic datatypes match one by one, regardless of displacements
  - Correct displacements at receiver side are automatically matched to the corresponding data items

Quiz:

1) Which one is a use case for MPI derived datatypes?
   a) Noncontiguous elements of an array
   b) A set of members in a `struct` (in C) or `type` (in Fortran)

2) Which of the bindings `MPI_Type_*` is most flexible?

3) The use of MPI derived datatype is helpful but can be unsafe
   a) Correct
   b) Incorrect