Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

IITU

Lecture 9

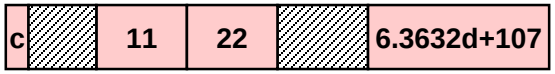
Derived Datatypes

MPI Datatypes

- In the previous chapters:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle

MPI Datatypes

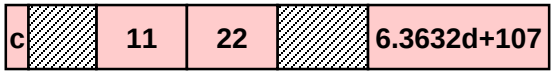
- In the previous chapters:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle
- New goals in this course chapter:
  - Transfer of any data in memory in one message



    - **Strided data (portions of data with holes between the portions)**
    - **Various basic datatypes within one message**
  - No multiple messages → **no multiple latencies**
  - No copying of data into contiguous scratch arrays → **no waste of memory bandwidth**

MPI Datatypes

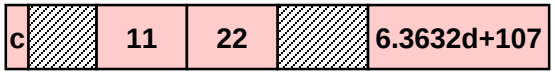
- In the previous chapters:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle
- New goals in this course chapter:
  - Transfer of any data in memory in one message



    - **Strided data (portions of data with holes between the portions)**
    - **Various basic datatypes within one message**
  - No multiple messages → **no multiple latencies**
  - No copying of data into contiguous scratch arrays → **no waste of memory bandwidth**
- Method: **Datatype handles**
  - Memory layout of send / receive buffer
  - Basic types / **derived types**:
    - **vectors**
    - **subarrays**
    - **structs**
    - **others**

MPI Datatypes

- In the previous chapters:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle
- New goals in this course chapter:
  - Transfer of any data in memory in one message



    - **Strided data (portions of data with holes between the portions)**
    - **Various basic datatypes within one message**
  - No multiple messages → **no multiple latencies**
  - No copying of data into contiguous scratch arrays → **no waste of memory bandwidth**
- Method: **Datatype handles**
  - Memory layout of send / receive buffer
  - Basic types / **derived types**:
    - **vectors**
    - **subarrays**
    - **structs**
    - **others**

Message passing:
- **Goal and reality may differ !!!**

Parallel file I/O:
- Derived datatypes are **important** to express I/O patterns

Data Layout and the Describing Datatype Handle

```
struct buff_layout
  { int    i_val[3];
    double d_val[5];
  } buffer;
```

Compiler

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;
MPI_Type_create_struct(2, array_of_blocklengths,
    array_of_displacements, array_of_types,
                          &buff_datatype);
MPI_Type_commit(&buff_datatype);
```

MPI_Send(**&buffer**, 1, **buff_datatype**, …)

&buffer = the start address of the data

the datatype handle describes the data layout

Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

- Matching datatypes:
  - List of basic datatypes must be identical
  - *(Displacements irrelevant)*

Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

- Matching datatypes:
  - List of basic datatypes must be identical
  - *(Displacements irrelevant)*

basic datatype 0	disp 0
basic datatype 1	disp 1
...	...
basic datatype n-1	disp n-1

Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

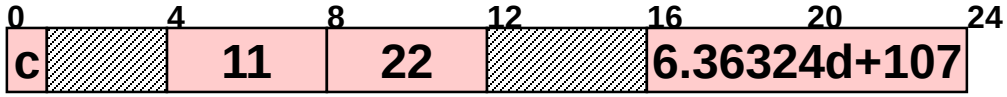
- Matching datatypes:
  - List of basic datatypes must be identical
  - *(Displacements irrelevant)*

basic datatype 0	disp 0
basic datatype 1	disp 1
...	...
basic datatype n-1	disp n-1

  - More precisely: sendcount x list of basic datatypes of the sendtype (n elements)
    must be identical with (pt-to-pt: the first n elements / collective: all elements of)
    recvcount x list of basic datatypes of the recvtype

Derived Datatypes — Type Maps

Example:



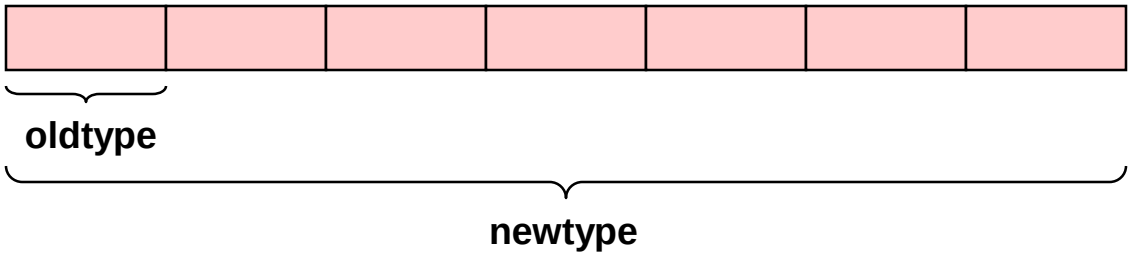
derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g.,
structures,
common blocks, module data subarrays,
some variables in the memory

Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



C

- C/C++: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype **newtype*)

Fortran

- Fortran: MPI_TYPE_CONTIGUOUS(count, oldtype, *newtype, ierror*)

```
mpi_f08:        INTEGER                 :: count
                TYPE(MPI_Datatype)      :: oldtype, newtype
                INTEGER, OPTIONAL       :: ierror

mpi & mpif.h:   INTEGER count, oldtype, newtype, ierror
```

Python

- Python: *newtype* = oldtype.Create_contiguous(int count)

Committing and Freeing a Datatype

- Before a dataytype handle is used in message passing communication, **it needs to be committed with MPI_TYPE_COMMIT**.

- This need be done only once (by each MPI process).
  (Using more than once ☹ corresponds to additional no-operations.)

C

Fortran

Python

- C/C++: int MPI_Type_commit(MPI_Datatype **datatype*);
- Fortran: MPI_TYPE_COMMIT(*datatype*, *IERROR*)

  mpi_f08: TYPE(MPI_Datatype) :: datatype
  INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER datatype, ierror
- Python: datatype.Commit()

IN-OUT argument
(although handle is not modified)

- If usage is over, one may call MPI_TYPE_FREE() to free a datatype and its internal resources.

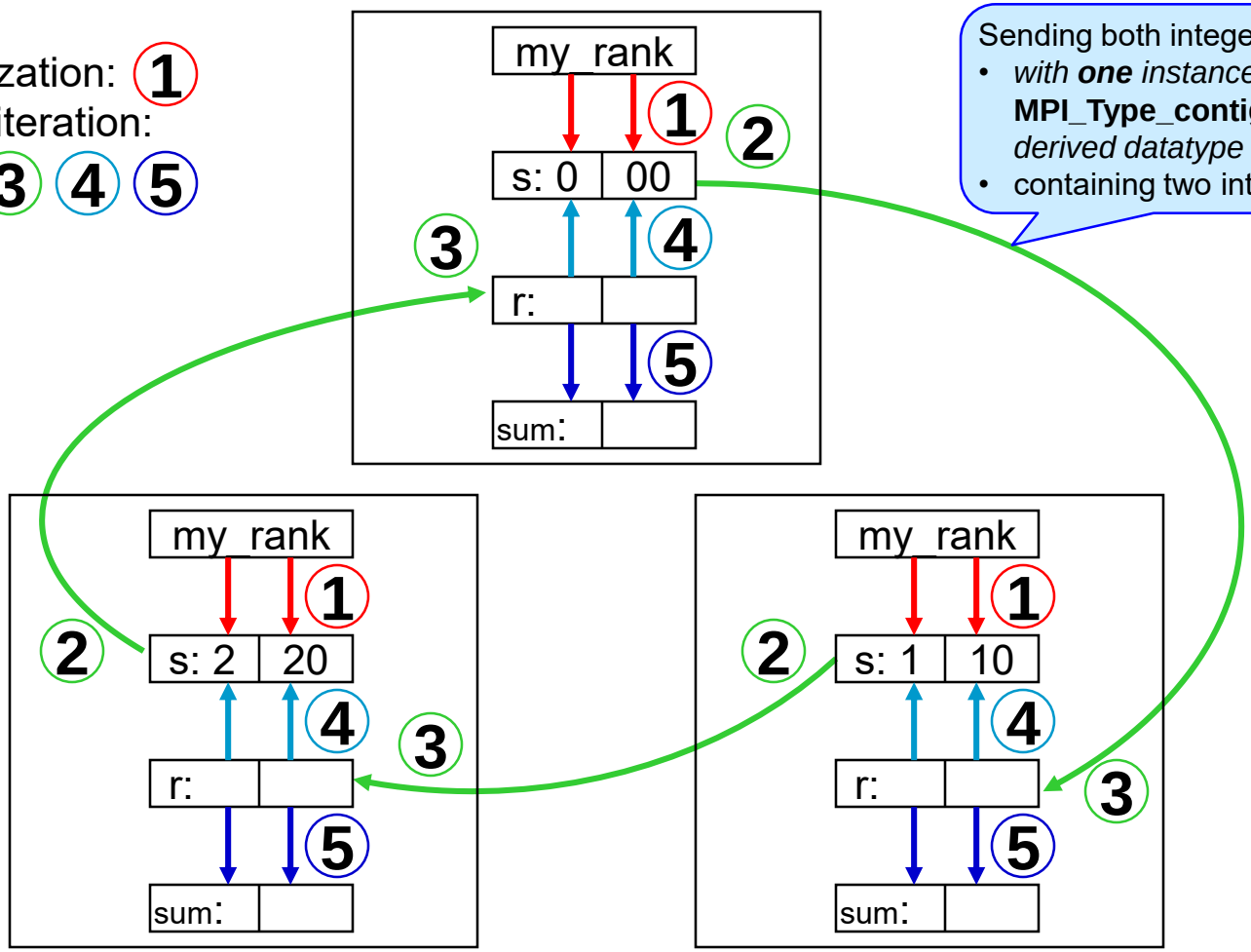
Exercise 1 — Derived Datatypes

In MPI/tasks/…

- Use C C/Ch12/derived-contiguous-skel.c
  or Fortran F_30/Ch12/derived-contiguous-skel_30.f90
  or Python PY/Ch12/derived-contiguous-skel.py
- We us a modified pass-around-the-ring exercise:
  It sends a struct with two integers
- They are initialized with **my_rank** and **10*my_rank**
- Therefore we calculate two separate sums.
- Currently, the data is send with the description
  - "snd_buf, 2, MPI_INTEGER"
- Please substitute this by using a
  - derived datatype
  - with a type map of "two integers"
  - Of course produced with the two routines on the previous slides

Exercise 1 — Derived Datatypes

Initialization: 1
Each iteration:
2 3 4 5

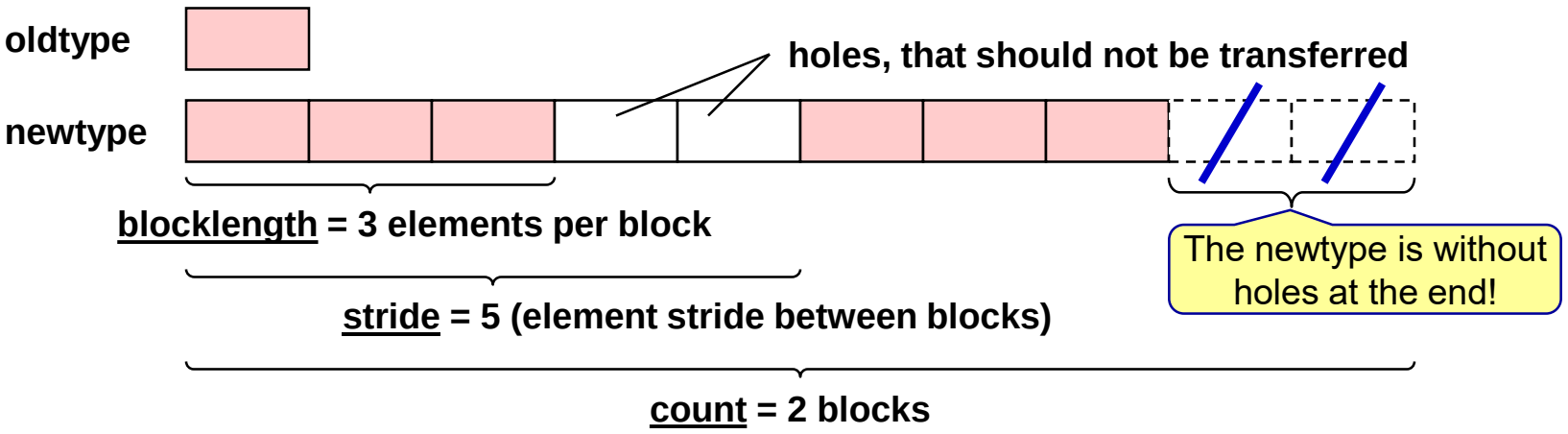


Sending both integers
- *with **one** instance of an* **MPI_Type_contiguous** *derived datatype*
- containing two integers

Exercises 1b (advanced) — MPI_Sendrecv

3. Substitute your Issend–Recv–Wait method by **MPI_Sendrecv** in your ring-with-datatype program:
   - MPI_Sendrecv is a *deadlock-free* combination of MPI_Send and MPI_Recv: **2** **3**
   - MPI_Sendrecv is described in the MPI standard.
     (You can find MPI_Sendrecv by looking at the function index on the last pages of the standard document.)

Vector Datatype



C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype **newtype*)

Fortran

- Fortran: MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, *newtype*, *ierror*)

mpi_f08:
```
INTEGER                 :: count, blocklength, stride
TYPE(MPI_Datatype)      :: oldtype, newtype
INTEGER, OPTIONAL       :: ierror
```

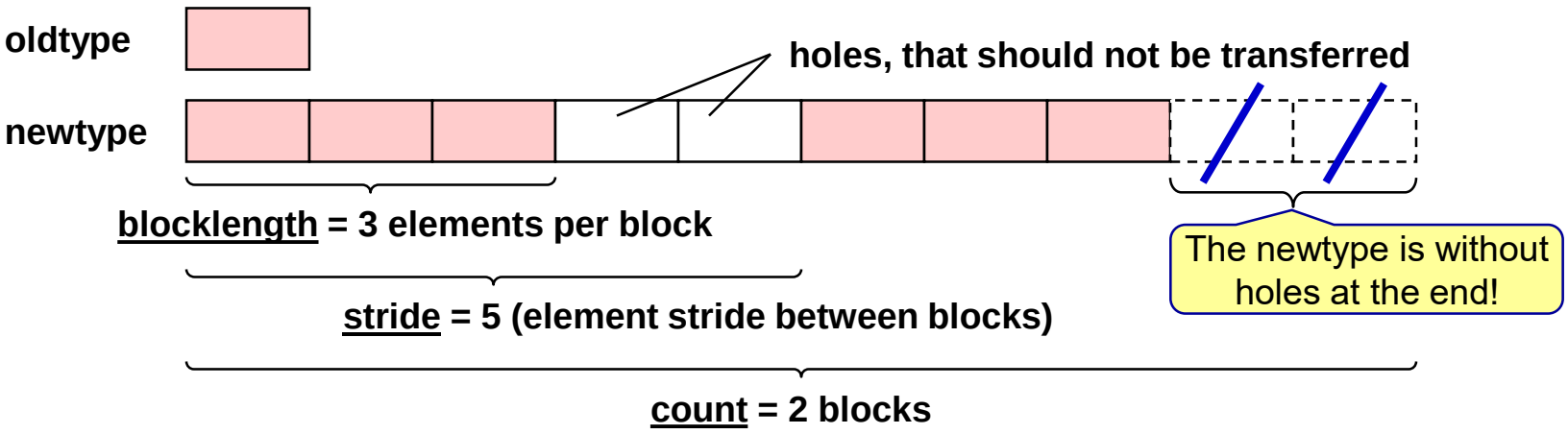
mpi & mpif.h:
```
INTEGER count, blocklength, stride, oldtype, newtype, ierror
```

Python

- Python: *newtype* = oldtype.Create_vector(int count, int blocklength, int stride)

Vector Datatype

MPI_Type_create_subarray is more flexible and usable for any dimensions, see course chapter 12-(2) and example in 13-(2)



C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype **newtype*)

Fortran

- Fortran: MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, *newtype*, *ierror*)

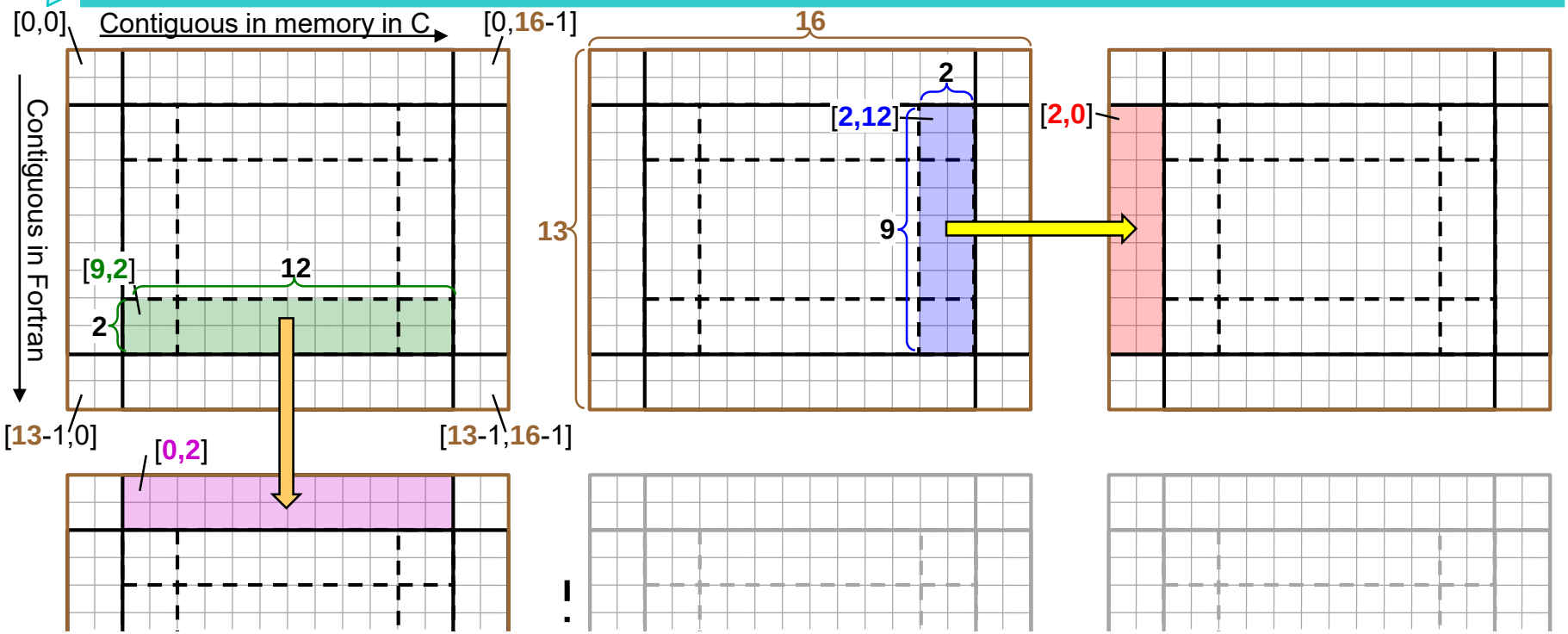
```
mpi_f08:          INTEGER                    :: count, blocklength, stride
                  TYPE(MPI_Datatype)         :: oldtype, newtype
                  INTEGER, OPTIONAL          :: ierror

mpi & mpif.h:     INTEGER count, blocklength, stride, oldtype, newtype, ierror
```

Python

- Python: *newtype* = oldtype.Create_vector(int count, int blocklength, int stride)

Example with MPI_Type_vector *)

Same indexes in C and Fortran



C

```
MPI_Type_vector(2, 12, 16, etype, &newt);
MPI_Send(&a[9,2], 1, newt, …);
```

```
MPI_Type_vector(9, 2, 16, etype, &newt);
MPI_Send(&a[2,12], 1, newt, …);
```

```
MPI_Type_vector(9, 2, 16, etype, &newt);
MPI_Recv(&a[2,0], 1, newt, …);
```

Fortran

```
CALL MPI_Type_vector(12, 2, 13, …,newt)
CALL MPI_Send(a(9,2), 1, newt, ...)
```

```
CALL MPI_Type_vector(2, 9, 13, …)
CALL MPI_Send(a(2,12), 1, newt, ...)
```

```
CALL MPI_Type_vector(2, 9, 13, …, newt)
CALL MPI_Recv(a(2,0), 1, newt, …)
```

```
MPI_Type_vector(2, 12, 16, etype, &newt);
MPI_Recv(&a[0,2], 1, newt, …);
```

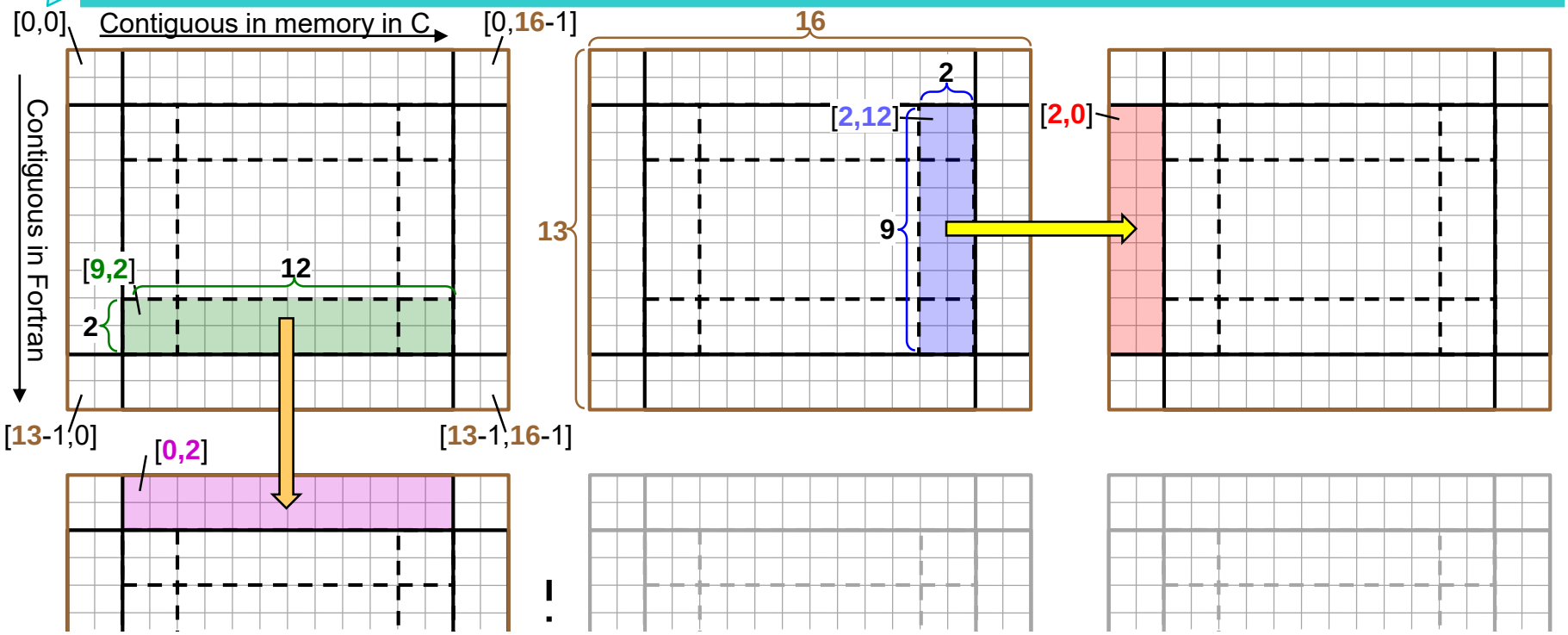
```
CALL MPI_Type_vector(12, 2, 13, …, &newt)
CALL MPI_Recv(a(0,2), 1, newt, ...)
```

Python Same numbering as with **C**

*) Figure: as in a math matrix, first index is vertical (i.e., not horizontal as in a x,y-diagram)

Same example with MPI_Type_create_subarray

Same indexes in C and Fortran



Fortran

```
CALL MPI_Type_create_subarray(
2, [13,16], [2,12], [9,2],
MPI_ORDER_FORTRAN, etype, newt)
MPI_Send(a, 1, newt, …);
```

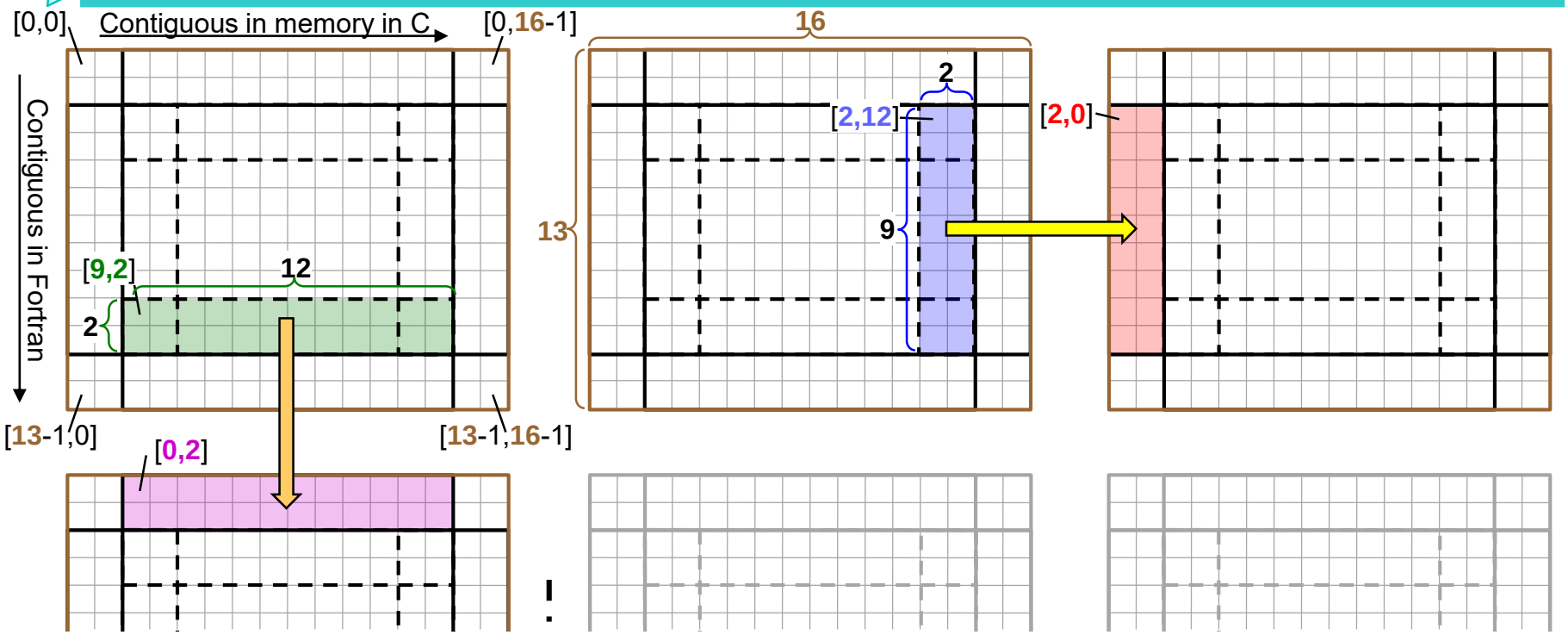
```
CALL MPI_Type_create_subarray(
2, [13,16], [2,12], [0,2],
MPI_ORDER_FORTRAN, etype, newt)
MPI_Recv(a, 1, newt, …);
```

```
CALL MPI_Type_create_subarray(
ndims=2, array_of_sizes=[13,16],
array_of_subsizes=[9,2],
array_of_starts=[2,12],
order=MPI_ORDER_FORTRAN,
oldtype=etype, newtype=newt)
CALL MPI_Send(a, 1, newt, …)
```

```
CALL MPI_Type_create_subarray(
ndims=2, array_of_sizes=[13,16],
array_of_subsizes=[9,2],
array_of_starts=[2,0],
order=MPI_ORDER_FORTRAN,
oldtype=etype, newtype=newt)
MPI_Recv(a, 1, newt, …)
```

Same example with MPI_Type_create_subarray

Same indexes in C and Fortran



Fortran

```
CALL MPI_Type_create_subarray(
2, [13,16], [2,12], [9,2],
MPI_ORDER_FORTRAN, etype, newt)
MPI_Send(a, 1, newt, …);
```

↓

```
CALL MPI_Type_create_subarray(
2, [13,16], [2,12], [0,2],
MPI_ORDER_FORTRAN, etype, newt)
MPI_Recv(a, 1, newt, …);
```

```
CALL MPI_Type_create_subarray(
ndims=2, array_of_sizes=[13,16],
array_of_subsizes=[9,2],
array_of_starts=[2,12],
order=MPI_ORDER_FORTRAN,
oldtype=etype, newtype=newt)
CALL MPI_Send(a, 1, newt, …)
```

→

```
MPI_Recv(a, 1, newt, …)
```

```
CALL MPI_Type_create_subarray(
ndims=2, array_of_sizes=[13,16],
array_of_subsizes=[9,2],
array_of_starts=[2,0],
order=MPI_ORDER_FORTRAN,
oldtype=etype, newtype=newt)
```

C **Python**

Same numbers in C and Fortran, only the **order** is different: MPI_ORDER_C in C and Python

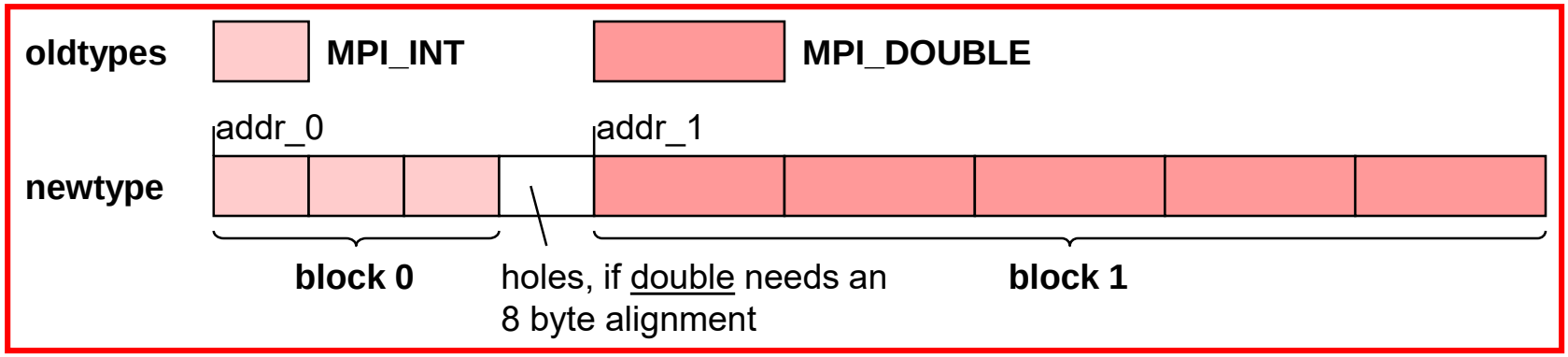
Struct Datatype

C

Fortran

Python

Struct Datatype

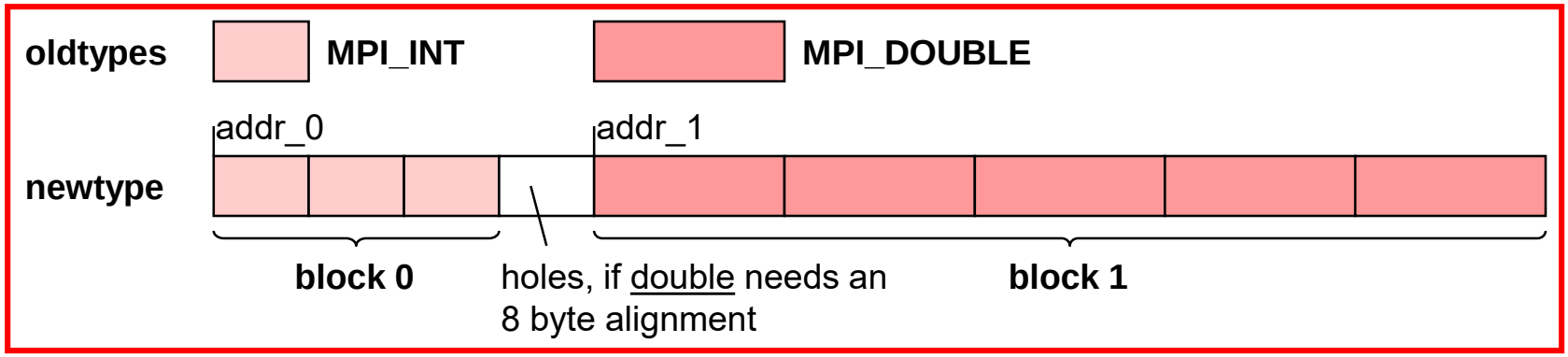


C

Fortran

Python

Struct Datatype



C

- C/C++: int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *_newtype_)

Fortran

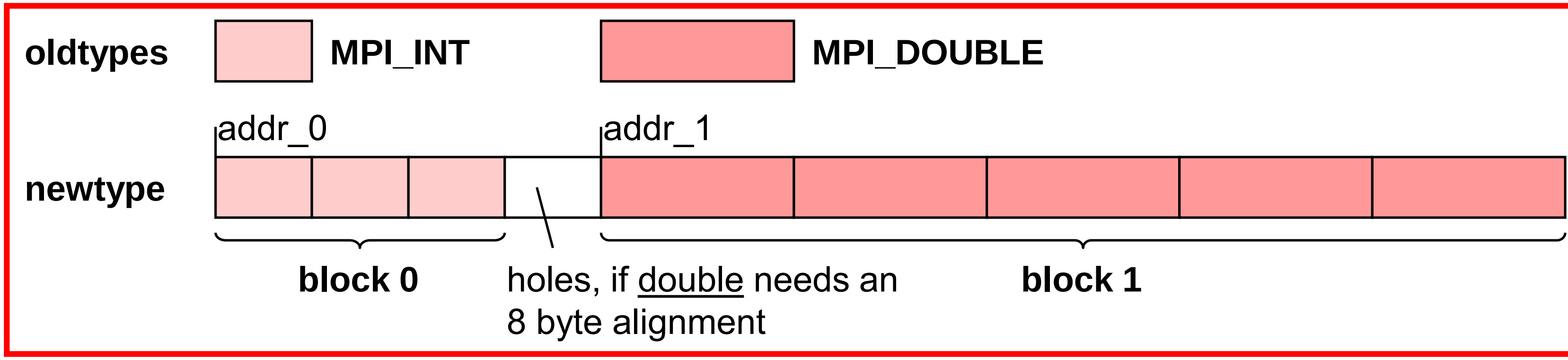
- Fortran: MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements[1)], array_of_types, _newtype_, _ierror_)

Python

- Python: _newtype_ = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)

[1)] INTEGER(KIND=MPI_ADDRESS_KIND) array_of_displacements

Struct Datatype



C

- C/C++: int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

Fortran

- Fortran: MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements[1], array_of_types, *newtype*, *ierror*)

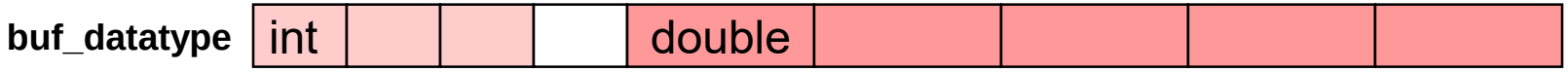
Python

- Python: *newtype* = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)

count	**= 2**		
array_of_blocklengths	**= (3,**	**5**	**)**
array_of_displacements	**= (0,**	**addr_1 – addr_0**	**)** [2]
array_of_types	**= (MPI_INT,**	**MPI_DOUBLE**	**)**

[1] INTEGER**(KIND=MPI_ADDRESS_KIND)** array_of_displacements
[2] Via MPI_Get_address and MPI_Aint_diff, see following slides

Memory Layout of Struct Datatypes



Fixed memory layout:

C

- C

```
struct buff
    {  int       i_val[3];
       double  d_val[5];  }
```

Fortran

- Fortran, derived types

```
TYPE buff_type
  SEQUENCE
  INTEGER, DIMENSION(3):: i_val
  DOUBLE PRECISION, &
                DIMENSION(5):: d_val
END TYPE buff_type
TYPE (buff_type) :: buff_variable
```

!!!

Alternative, in MPI-3.0:

```
TYPE, BIND(C) :: buff_type
```

- Fortran, common block

```
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
```

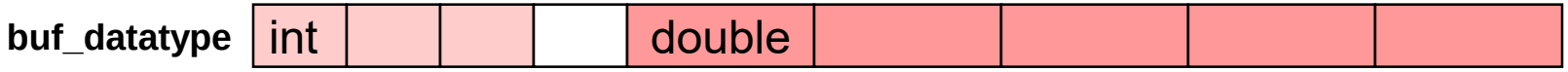
Python

- Python – mpi4py with numpy:

```
buff_type = np.dtype([('i', np.intc, 3), ('d', np.double, 5)], align=True)
buff = np.empty((),dtype=buff_type)
```

True: with hole, as in C.
Default = False: no holes, problematic.

Memory Layout of Struct Datatypes



C

Fortran

Python

Fixed memory layout:

- C

```
struct buff
    {  int     i_val[3];
       double  d_val[5];  }
```

- Fortran, derived types

```
TYPE buff_type
  SEQUENCE
  INTEGER, DIMENSION(3):: i_val
  DOUBLE PRECISION, &
                DIMENSION(5):: d_val
END TYPE buff_type
TYPE (buff_type) :: buff_variable
```

!!!

Alternative, in MPI-3.0:

```
TYPE, BIND(C) :: buff_type
```

- Fortran, common block

```
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
```

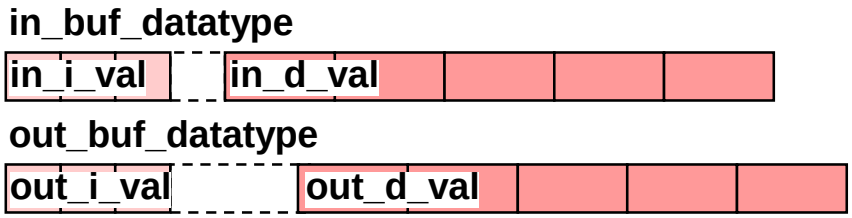
- Python – mpi4py with numpy:

```
buff_type = np.dtype([('i', np.intc, 3), ('d', np.double, 5)], align=True)
buff = np.empty((),dtype=buff_type)
```

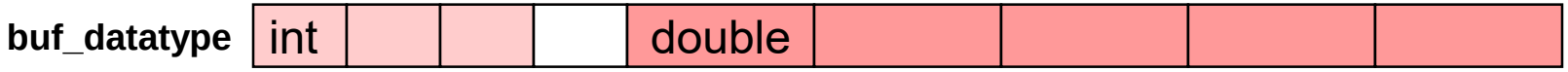
True: with hole, as in C.
Default = False: no holes, problematic.

Alternatively, arbitrary memory layout:

- Each array is allocated independently.
- Each buffer is a pair of a 3-int-array and a 5-double-array.
- The length of the hole may be any arbitrary positive or negative value!
- For each buffer, one needs a specific datatype handle
- **CAUTION – Fortran register optimization:** MPI_Send & _Recv of **…d_val** is invi-sible for the compiler → add MPI_Address



Memory Layout of Struct Datatypes



C

Fortran

Python

Fixed memory layout:

- C

```
struct buff
    {  int     i_val[3];
       double  d_val[5];  }
```

- Fortran, derived types

```
TYPE buff_type
  SEQUENCE
  INTEGER, DIMENSION(3):: i_val
  DOUBLE PRECISION, &
                DIMENSION(5):: d_val
END TYPE buff_type
TYPE (buff_type) :: buff_variable
```

!!!

Alternative, in MPI-3.0:

```
TYPE, BIND(C) :: buff_type
```

- Fortran, common block

```
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
```

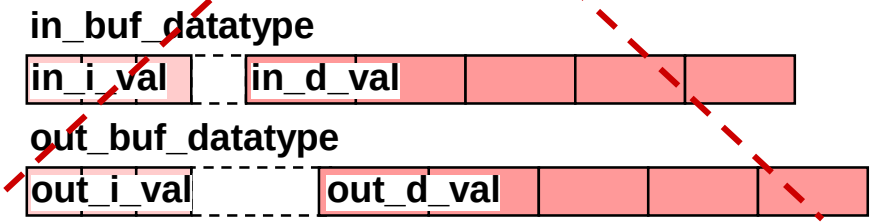
- Python – mpi4py with numpy:

```
buff_type = np.dtype([('i', np.intc, 3), ('d', np.double, 5)], align=True)
buff = np.empty((),dtype=buff_type)
```

True: with hole, as in C.
Default = False: no holes, problematic.

Alternatively, arbitrary memory layout:

- Each array is allocated independently.
- Each buffer is a pair of a 3-int-array and a 5-double-array.
- The length of the hole may be any arbitrary positive or negative value!
- For each buffer, one needs a specific datatype handle
- **CAUTION – Fortran register optimization:** MPI_Send & _Recv of **…d_val** is invi-sible for the compiler → add MPI_Address



Not portable, because address differences are allowed only inside of structures or arrays

→ MPI-3.1/-4.0, Sect. 4/5.1.12 “Correct Use of Addresses”

How to compute the displacement (1)

- array_of_displacements[i] := address(block_i) – address(block_0)

Retrieve an absolute address:

C

Fortran

Python

– C/C++: int MPI_**Get**_address(void* location, MPI_Aint **address*)

– Fortran: MPI_**GET**_ADDRESS(location, *address*, *ierror*)

mpi_f08:
```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
INTEGER(KIND=MPI_ADDRESS_KIND) :: address
INTEGER, OPTIONAL :: ierror
```

mpi & mpif.h:
```
<type>     location(*)
INTEGER(KIND=MPI_ADDRESS_KIND) address
INTEGER ierror
```

– Python: *address* = MPI.**Get**_address(location)

How to compute the displacement (2)

New in MPI-3.1

Relative displacement:= absolute address 1 – absolute address 2

C

Fortran

- C/C++: *MPI_Aint* MPI_Aint_diff(MPI_Aint addr1, MPI_Aint *addr2*)
- Fortran: MPI_AINT_DIFF(addr1, addr2)
  - mpi_f08: INTEGER(KIND=MPI_ADDRESS_KIND) :: addr1, addr2
  - mpi & mpif.h: INTEGER(KIND=MPI_ADDRESS_KIND) addr1, addr2
- Python: *int* MPI.Aint_diff(addr1, *addr2*)

Python

Python's int allows 64 bit

Python

How to compute the displacement (2)

New in MPI-3.1

Relative displacement:= absolute address 1 – absolute address 2

C

Fortran

Python

- C/C++: *MPI_Aint* MPI_Aint_diff(MPI_Aint addr1, MPI_Aint *addr2*)
- Fortran: MPI_AINT_DIFF(addr1, addr2)
  - mpi_f08: INTEGER(KIND=MPI_ADDRESS_KIND) :: addr1, addr2
  - mpi & mpif.h: INTEGER(KIND=MPI_ADDRESS_KIND) addr1, addr2
- Python: *int* MPI.Aint_diff(addr1, *addr2*)

Python's int allows 64 bit

New in MPI-3.1

New absolute address := existing absolute address + relative displacement:

C

Fortran

Python

- C/C++: *MPI_Aint* MPI_Aint_add(MPI_Aint base, MPI_Aint *disp*)
- Fortran: MPI_AINT_ADD(base, disp)
  - mpi_f08: INTEGER(KIND=MPI_ADDRESS_KIND) :: base, disp
  - mpi & mpif.h: INTEGER(KIND=MPI_ADDRESS_KIND) base, disp
- Python: *int* MPI.Aint_add(base, *disp*)

Example for array_of_displacements[i] := address(block_i) – address(block_0)

See also MPI-3.1/MPI-4.0, Example 4.8/5.8, page 102/142 and Example 4.17/5.17, pp 125-127/168-171

C

```
struct buff
    {       int     i[3];
            double d[5];
    } snd_buf;
MPI_Aint iaddr0, iaddr1, disp;
MPI_Get_address( &snd_buf.i[0], &iaddr0);  // the address value &snd_buf.i[0] is stored into variable iaddr0
MPI_Get_address(&snd_buf.d[0], &iaddr1);
disp = MPI_Aint_diff(iaddr1, iaddr0);          // MPI-3.0 & former: disp = iaddr1−iaddr0
```

New in MPI-3.1

Fortran

```
TYPE buff_type
  SEQUENCE
  INTEGER,                  DIMENSION(3) :: i
  DOUBLE PRECISION,         DIMENSION(5) :: d
END TYPE buff_type
TYPE (buff_type) :: snd_buf
INTEGER(KIND=MPI_ADDRESS_KIND) iaddr0, iaddr1, disp;  INTEGER ierror
CALL MPI_GET_ADDRESS( snd_buf%i(1), iaddr0, ierror)  ! The address of snd_buf%i(1) is stored in iaddr0
CALL MPI_GET_ADDRESS(snd_buf%d(1), iaddr1, ierror)
disp = MPI_AINT_DIFF(iaddr1, iaddr0)                          ! MPI-3.0 & former: disp = iaddr2−iaddr1
```

New in MPI-3.1

Python

```
np_dtype = np.dtype([('i', np.intc, 3), (‘d', np.double, 4)])
snd_buf = np.empty((),dtype=np_dtype)
addr0 = MPI.Get_address(snd_buf['i'])
addr1 = MPI.Get_address(snd_buf[‘d'])
disp = MPI.Aint_diff(addr1, addr0)
```

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer,
  communicating this send-buffer into a contiguous recv-buffer,
  and copying the recv-buffer back into the strided application array

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer,
  communicating this send-buffer into a contiguous recv-buffer,
  and copying the recv-buffer back into the strided application array
- Using derived datatype handles

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods, or mpi4py object serialization

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is not portable!

Exercise 2 — Derived Datatypes

- Modify the pass-around-the-ring exercise.
- Use the following skeletons to reduce software-coding time:

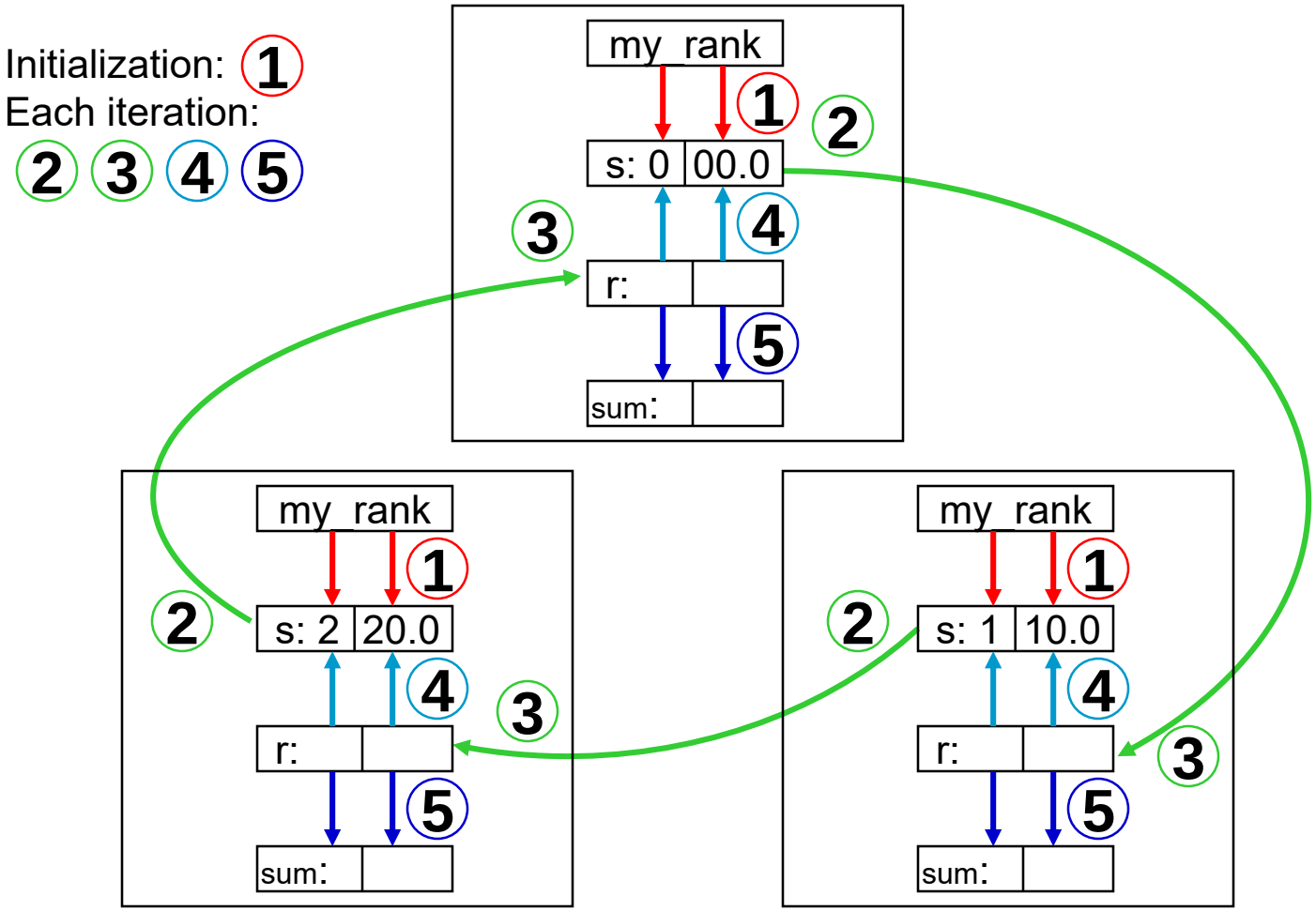
C
cd ~/MPI/tasks/**C**/Ch12/ ; cp -p derived-struct-skel.c derived-struct.c

Fortran
cd ~/MPI/tasks/**F_30**/Ch12/ ; cp -p derived-struct-skel_30.f90 derived-struct_30.f90

Python
cd ~/MPI/tasks/**PY**/Ch12/ ; cp -p derived-struct-skel.py derived-struct.py

- Calculate two separate sums:
  - rank integer sum (as before)
  - rank floating point sum
- Use a *struct* datatype for this
- with same fixed memory layout for send and receive buffer.
- Substitute all ___ within the skeleton
  and modify the second part, i.e., steps 1-5 of the ring example

Exercise 2 — Derived Datatypes



Exercises 3+4 (advanced) — Sendrecv & Sendrecv_replace

3. Substitute your Issend–Recv–Wait method by **MPI_Sendrecv** in your ring-with-datatype program:
   - MPI_Sendrecv is a *deadlock-free* combination of MPI_Send and MPI_Recv: 2 3
   - MPI_Sendrecv is described in the MPI standard.
     - **You can find MPI_Sendrecv by looking at the function index on the last pages of the standard document.**

Same Exercise as Advanced Exercise 1b

If you solved already Advanced Exercise 1b then move to Exercise 4

4. Substitute MPI_Sendrecv by **MPI_Sendrecv_replace**:
   - Three steps are now combined: 2 3 4
   - The receive buffer (rcv_buf) must be removed.
   - The iteration is now reduced to three statements:
     - **MPI_Sendrecv_replace to pass the ranks around the ring,**
     - **computing the integer sum,**
     - **computing the floating point sum.**

Quiz on Chapter 12-(1) – Derived datatypes

A. Which types of data in your application's memory can you describe with a derived datatype handle?

_____

B. Logically, to which internal structure points a derived datatype handle?

_____

C. Two pairs ($count_1$, $datatype_1$) and ($count_2$, $datatype_2$) match if …?

_____

D. If you have an array of a structure in your memory, how would you describe this?

_____

E. Which additional MPI procedure call is required, before you can use a newly generated derived datatype handle in an MPI communication procedure?

_____

F. If you have a (noncontiguous) subarray of a multidimensional array, which procedure would you use to generate an appropriate derived datatype handle and which count value would you use in MPI_Send or MPI_Recv?

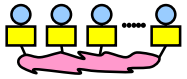
_____

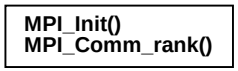
G. Which MPI procedures and functions should you use to calculate a byte displacement?

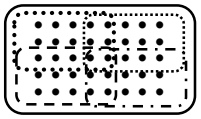
_____

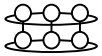
Chap.12 Derived Datatypes (2nd part)

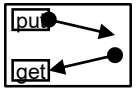
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environmental management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication



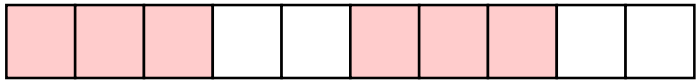






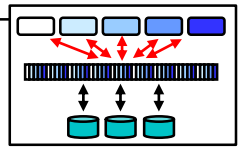


12. Derived datatypes



- (1) transfer of any combination of typed data
- **(2) alignment, resizing, large counts, other derived types, MPI_Pack, MPI_BOTTOM**

13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice
19. Heat example

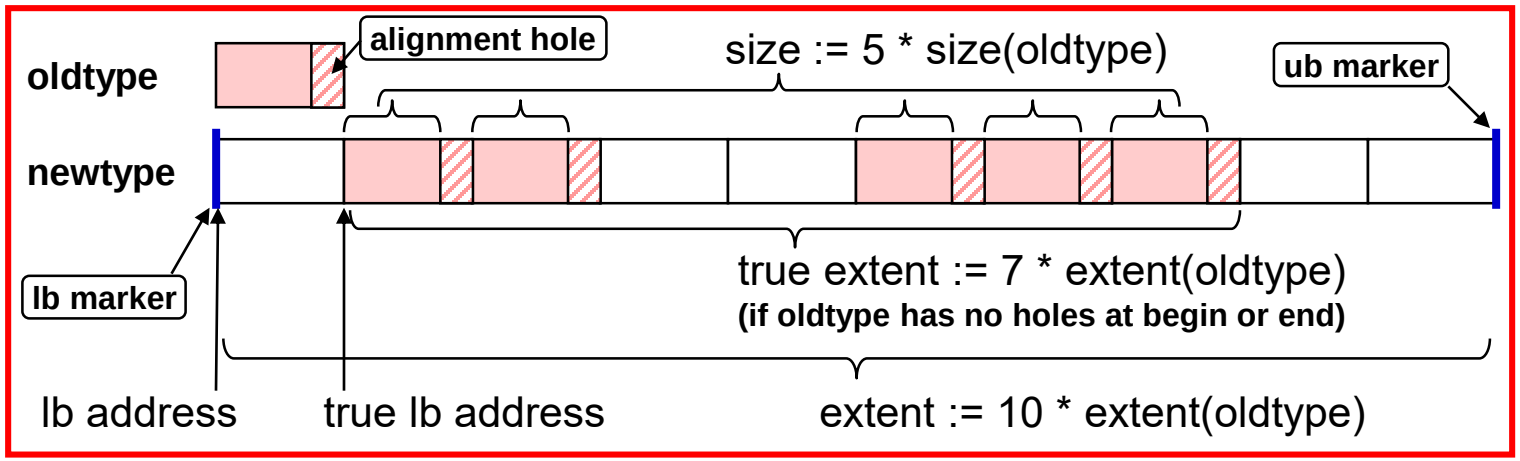


Does your MPI derived datatype really fits to the data in your memory?
→ More complicated than expected!

Size, Extent and True Extent of a Datatype, I.

- Size := number of bytes that have to be transferred.
- Extent := spans from first to last byte (including all holes).
- True extent := spans from first to last true byte (excluding holes at begin+end)
- Automatic holes at the end for necessary alignment purpose
- Additional holes at begin and end by *lower bound (**lb**)* and *upper bound (**ub**)* markers: MPI_TYPE_CREATE_RESIZED
- Basic datatypes: Size = Extent = number of bytes used by the compiler.

Example:



skipped

Size and Extent of a Datatype, II.

C

Fortran

Python

- C/C++: int MPI_Type_size(MPI_Datatype datatype, int *size*)
- Fortran: MPI_TYPE_SIZE(datatype, *size*, *ierror*)

```
mpi_f08:          TYPE(MPI_Datatype)    :: datatype
                  INTEGER               :: size
                  INTEGER, OPTIONAL     :: ierror
mpi & mpif.h:     INTEGER datatype, size, ierror
```

- Python: *size* = datatype.Get_size(MPI_Datatype)

C

Fortran

Python

- C/C++: int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint **lb*, MPI_Aint **extent*)
- Fortran: MPI_TYPE_GET_EXTENT(datatype, *lb*, *extent*, *ierror*)

```
mpi_f08:          TYPE(MPI_Datatype)                 :: datatype
                  INTEGER(KIND=MPI_ADDRESS_KIND)     :: lb, extent
                  INTEGER, OPTIONAL                  :: ierror
mpi & mpif.h:     INTEGER datatype, ierror
                  INTEGER(KIND=MPI_ADDRESS_KIND)     lb, extent
```

- Python: (*lb*, *extent*) = datatype.Get_extent()

C

Fortran

Python

- C/C++: int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint **true_lb*, MPI_Aint **true_extent*)
- Fortran, Python: ditto

- Removed MPI-1 interface: MPI_TYPE_EXTENT()

Fortran derived types and MPI_Type_create_struct

- SEQUENCE **and BIND(C)** derived application types can be used as buffers in MPI operations.
- Alignment calculation of basic datatypes:
  - In MPI-2.2, it was undefined in which environment the alignments are taken.
  - There is no sentence in the standard.
  - **It may depend on compilation options!**
  - In MPI-3.0 to MPI-4.0, still undefined, but recommended to use a BIND(C) environment.

Alignment rule, holes and resizing of structures (1)

- The compiler may add additional alignment holes
  - within a structure (e.g., between a float and a double)
  - at the end of a structure (after elements with different sizes)!
  - See MPI-3.1/MPI-4.0, Sect. 4 / 5.1.6, Advice to users on page 106 / 146.

Alignment rule, holes and resizing of structures (1)

- The compiler may add additional alignment holes
  - within a structure (e.g., between a float and a double)
  - at the end of a structure (after elements with different sizes)!
  - See MPI-3.1/MPI-4.0, Sect. 4 / 5.1.6, Advice to users on page 106 / 146.
- Alignment hole at the end is important when using an array of structures!

Alignment rule, holes and resizing of structures (1)

- The compiler may add additional alignment holes
  - within a structure (e.g., between a float and a double)
  - at the end of a structure (after elements with different sizes)!
  - See MPI-3.1/MPI-4.0, Sect. 4 / 5.1.6, Advice to users on page 106 / 146.
- Alignment hole at the end is important when using an array of structures!
- Implication **(for C, Fortran(!) and Python)**:
  - If an array of structures (in C/C++) or derived types (in Fortran) should be communicated, it is recommended that
  - the user creates a portable datatype handle and
  - should apply additionally **MPI_Type_create_resized** to this datatype handle.
  - See Example in MPI-3.1/MPI-4.0, Sect. 17/19.1.15 on pages 637-638 / 823-825.

Alignment rule, holes and resizing of structures (1)

- The compiler may add additional alignment holes
  - within a structure (e.g., between a float and a double)
  - at the end of a structure (after elements with different sizes)!
  - See MPI-3.1/MPI-4.0, Sect. 4 / 5.1.6, Advice to users on page 106 / 146.
- Alignment hole at the end is important when using an array of structures!
- Implication **(for C, Fortran(!) and Python)**:
  - If an array of structures (in C/C++) or derived types (in Fortran) should be communicated, it is recommended that
  - the user creates a portable datatype handle and
  - should apply additionally **MPI_Type_create_resized** to this datatype handle.
  - See Example in MPI-3.1/MPI-4.0, Sect. 17/19.1.15 on pages 637-638 / 823-825.
- Holes (e.g., due to alignment gaps) may cause significant loss of bandwidth
  - By definition, MPI is not allowed to transfer the holes.
  - Therefore the user should fill holes with dummy elements.
  - See Example MPI-3.1/MPI-4.0, Sect. 4/5.1.6, Advice to users on page 106 / 146.

Alignment rule, holes and resizing of structures (2)

- **Correctness** problem with **array of structures**:
  - Possibility: MPI extent of a structure != real size of the structure
  - Reason: MPI adds at the end an alignment hole because the MPI library has wrong expectations about compiler rules
    - **For a basic datatype within the structure**
    - **For the allowed size of the whole structure (e.g. multiple of 16)**

Alignment rule, holes and resizing of structures (2)

- **Correctness** problem with **array of structures**:
  - Possibility: MPI extent of a structure != real size of the structure
  - Reason: MPI adds at the end an alignment hole because the MPI library has wrong expectations about compiler rules
    - **For a basic datatype within the structure**
    - **For the allowed size of the whole structure (e.g. multiple of 16)**
  - Solution in C: MPI_Type_create_resized(old_struct_type, (MPI_Aint) 0, (MPI_Aint) sizeof(my_struct_array[0]), &correct_struct_type);
    [or use the following method:]

    lb=0, i.e., no hole at the beginning

  - & in Fortran:

```
INTEGER(KIND=MPI_ADDRESS_KIND)  &
     :: address1,  address2,  lb,  new_extent
CALL  MPI_Get_address( my_struct(1), address1, ierror)
CALL  MPI_Get_address( my_struct(2), address2, ierror)
new_extent  =  MPI_Aint_diff( address2, address1);  lb = 0
CALL  MPI_Type_create_resized ( &
     old_struct_type, lb, new_extent, correct_struct_type, ierror)
```

  - & in Python:

```
send_recv_resized =
    send_recv_type.Create_resized(0, snd_buf.itemsize)
```

correct type of one struct

type of one struct with possibly wrong extent

array of struct

length of one struct

Alignment rule, holes and resizing of structures (3)

- Correctness problem with array of structures (continued):

C

– Example in C with [double+int]-structure:

  - MPI/tasks/C/Ch12/derived-struct-double+int.c
  - Compiled and run on Cray with Intel compiler
    - **module switch PrgEnv-cray PrgEnv-intel**
    - **cc –Zp4 –o a.out ~/MPI/tasks/C/Ch12/derived-struct-double+int.c**
    - **aprun –n 4 ./a.out | sort**

With default alignment, all works on the tested platform (in Nov. 2015)

  - Result:
    - **MPI_Type_get_extent: 16**
    - **sizeof: 12**
    - **real size is: 12**

Python

– Similar in Python with default align=False

  - MPI/tasks/C/Ch12/derived-struct-double+int.py
  - Change align=True on line 34 to align=False → wrong results as above

For portable & correct applications
with **arrays of structures**,
the datatypes should be always **resized**!

Alignment rule, holes and resizing of structures (4)

- Correctness problem with array of structures (continued):

Fortran

– Example in Fortran with [double precision + integer]-structure:

- MPI/tasks/F_30/Ch12/derived_struct_dp+integer_30.f90
- Compiled and run on Cray with Intel compiler
  - **module switch PrgEnv-cray PrgEnv-intel**
  - **ftn –o a.out ~/MPI/tasks/F_30/Ch12/derived-struct-dp+integer_30.f90**
  - **aprun –n 4 ./a.out | sort**
- Result:
  - **MPI_Type_get_extent: 16**
  - **real size is: 12**
- **Surprise (?):**
  - **~/MPI/tasks/F_30/Ch12/derived-struct-dp+integer-bindC_30.f90**
  - **MPI_Type_get_extent: 16**
  - **real size is: 16**
- **2nd Surprise:** With PrgEnv-cray, all sizes are 16 bytes

Fortran struct with SEQUENCE attribute

Fortran struct with BIND(C)

Alignment rule, holes and resizing of structures (5)

- **Performance** problem with **holes in structures**:
  - Correct solution for homogeneous and heterogeneous environments:
    - **Add dummy elements to fill the holes (in the structure and in the datatype)**
  - In a homogeneous environment:
    - **One may use MPI_BYTE**
    - **Transfer whole structure as an array of bytes**
    - **CAUTION: No data conversion of different data representations (e.g., big and little endian) in heterogeneous environments**

New in MPI-3.0

Large Counts with MPI_Count, …

- MPI uses different integer types
  - int and INTEGER
  - MPI_Aint = INTEGER(KIND=MPI_ADDRESS_KIND)
  - MPI_Offset = INTEGER(KIND=MPI_OFFSET_KIND)
  - MPI_Count = INTEGER(KIND=MPI_COUNT_KIND)

New in MPI-3.0

- sizeof(int) ≤ sizeof(MPI_Aint) / sizeof(MPI_Offset) ≤ sizeof(MPI_Count)
- All count arguments are int or INTEGER.
- Real message sizes may be larger due to datatype size.
- MPI_Type_get_extent, MPI_Type_get_true_extent, MPI_Type_size, MPI_Type_get_elements
  return **MPI_UNDEFINED** if value is too large

New in MPI-3.0

Python with mpi4py:
Although mpi4py uses **Python's int counts** (mostly inferred from numpy arrays), and Python's int is **not restricted to 32 bit**, the mpi4py interfaces may be mapped to the **32 bit C int count** arguments of the underlying MPI library.

New in MPI-3.0

Deprecated in MPI-4.1

- MPI_Type_get_extent_**x**, MPI_Type_get_true_extent_**x**, MPI_Type_size_**x**, MPI_Type_get_elements_**x**
  return values as **MPI_Count**

New in MPI-4.0

- **MPI_Xxxx_c(…)** in C: additional interfaces with large counts
  MPI_Xxxx(…) !(_c) in Fortran: overloaded interfaces with large counts

Two exceptions with explicit _c in Fortran:
MPI_Op_create_c & MPI_Register_datarep_c

All Derived Datatype Creation Routines (1)

skipped

- **MPI_Type_contiguous()**
  → already discussed
- **MPI_Type_vector()**
  → already discussed

  MPI_Type_create_hvector()
  → stride as byte size
- **MPI_Type_indexed()**
  → similar to .._struct(),
  same oldtype for all sub-blocks,
  displacements based on 0-based index in “array of oldtype”

  MPI_Type_create_hindexed()
  → with byte displacements
- **MPI_Type_create_indexed_block()**
  → same as MPI_Type_indexed()
  but same block length
  for each sub-block

  MPI_Type_create_hindexed_block()
  → with byte displacements
- **MPI_Type_create_struct()**
  → already discussed

All Derived Datatype Creation Routines (2)

- **MPI_Type_create_subarray()**
  - Extracts a subarray of an n-dimensional array
  - All the rest are holes
  - Ideal for halo exchange with n-dimensional Cartesian data-sets
  - Similar to MPI_Type_vector(), which works primarily for 2-dim arrays
  - Example, see course Chapter 13 *Parallel File I/O*
- **MPI_Type_create_darray()**
  - A generalization of **MPI_Type_create_subarray()**
  - Example, see course Chapter 13 *Parallel File I/O*

Subarray and darray: newtype may contain holes at begin and end !!! ☺ **Important for filetypes → *Parallel File I/O***

Removed MPI-1 interfaces	substituted by
MPI_Address	MPI_Get_address
MPI_Type_extent	MPI_Type_get_extent
MPI_Type_hvector	MPI_Type_create_hvector
MPI_Type_hindexed	MPI_Type_create_hindexed
MPI_Type_struct	MPI_Type_create_struct
MPI_Type_LB / _UB	MPI_Type_get_extent
Constant MPI_LB / _UB	MPI_Type_create_resized

New in MPI-2.0 to solve Fortran problem with small integer:
- Unchanged argument list in C.
- Modified length arguments in Fortran.

Better usable interface

Other MPI features: Pack/Unpack

- MPI_Pack & MPI_Unpack
  - Pack several data into a message buffer
  - Communicate the buffer with datatype = MPI_PACKED
- Canonical Pack & Unpack
  - Header-free packing in “external32” data representation
  - Only useful for cross-messaging **between different MPI libraries**!
  - Communicate the buffer with datatype = MPI_BYTE

Other MPI features: MPI_BOTTOM and absolute addresses

- MPI_BOTTOM in point-to-point and collective communication:
  - Buffer argument is MPI_BOTTOM
  - Then absolute addresses can be used in
    - **Communication routines with byte displacement arguments**
    - **Derived datatypes with byte displacements**
  - Displacements must be retrieved with **MPI_Get_address()**
  - MPI_BOTTOM is an address,
    i.e., **cannot be assigned to a Fortran variable**!
  - MPI-3.1/MPI-4.0, Section 2.5.4, p. 15 line 44 – p. 16 line 6 / p. 21 lines 12-22 shows all such address constants
    that cannot be used in expressions or assignments **in Fortran,** e.g.,
    - **MPI_STATUS_IGNORE (→ point-to-point comm.)**
    - **MPI_IN_PLACE (→ collective comm.)**
  - Fortran: Using MPI_BOTTOM & absolute displacement of variable X
    → the really used **`buffer`** must be declared as ASYNCHRONOUS
    & **`IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(buf)`**
    is needed:
    - **MPI_BOTTOM in a blocking MPI routine → MPI_F_SYNC_REG before and after this routine**
    - **in a nonblocking routine → MPI_F_SYNC_REG before this routine & after final WAIT/TEST**

Fortran

Already discussed in course Chapter 9 ***Virtual Topologies*** → Exercise with **MPI_Neighbor_alltoallw**

Performance options [already mentioned at the end of 12-(1)]

Which is the fastest neighbor communication with strided data?

- Copying the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is not portable!

Exercise 5+6 — Resizing a Derived Datatypes

Use the following examples for testing and as code-basis:

C
- **MPI/tasks/C/Ch12/derived-struct-double+int.c** or

Fortran
- **MPI/tasks/F_30/Ch12/derived-struct-dp+integer_30.f90** and
- **MPI/tasks/F_30/Ch12/derived-struct-dp+integer-bindC_30.f90**

Python
- **MPI/tasks/PY/Ch12/derived-struct-double+int.py**

5. **Compile and test** with different compilers and accompanying MPI libraries
   - Pipe the stdout to: `| sort +0 -1 -n +1 -2`
   - Example:

```
mpiexec -n 4 ./a.out | sort +0 -1 -n +1 -2
```

6. **Implement a new datatype handle by resizing the old one.**
   - Don’t forget to substitute the datatype handle in all communication calls.