Advanced MPI

User-defined datatypes

Leon Kos

University of Ljubljana, FME, LECAD lab

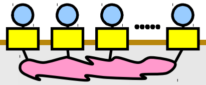
Acknowledgments

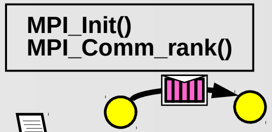
- *Derived types* is Chapter 12 from *Introduction to the Message Passing Interface (MPI)* course by Rolf Rabenseifner from University of Stuttgart and High-Performance Computing-Center Stuttgart (HLRS)
- The MPI-1.1 part of this course is partially based on the MPI course developed by the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
- Thanks to the EPCC, especially to Neil MacDonald, Elspeth Minty,
  Tim Harding, and Simon Brown.
- Course Notes and exercises of the EPCC course can be used together with this slides.
- The MPI-2.0 part is partially based on the MPI-2 tutorial at the MPIDC 2000 by Anthony Skjellum, Purushotham Bangalore, Shane Hebert (High Performance Computing Lab, Mississippi State University, and Rolf Rabenseifner (HLRS)
- Some MPI-3.0 detailed slides are provided by the MPI-3.0 ticket authors, chapter authors, or chapter working groups, Richard Graham (chair of MPI-3.0), and Torsten Hoefler (additional example about new one-sided interfaces)
- Thanks to Dr. Claudia Blaas-Schenner from TU Wien (Vienna) and many other trainers and participants for all their helpful hints for optimizing this course over so many years.

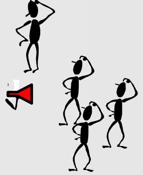
Derived Datatypes

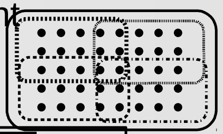
Derived Datatypes

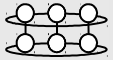
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. **Derived datatypes**
    - **(1) transfer of any combination of typed data**
    - **(2) advanced features, alignment, resizing**
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

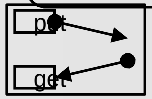


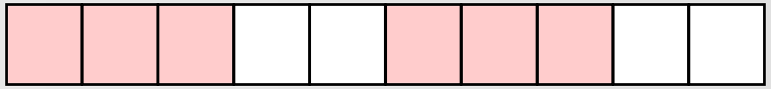


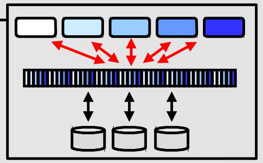










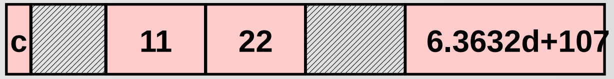


MPI Datatypes

- 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**

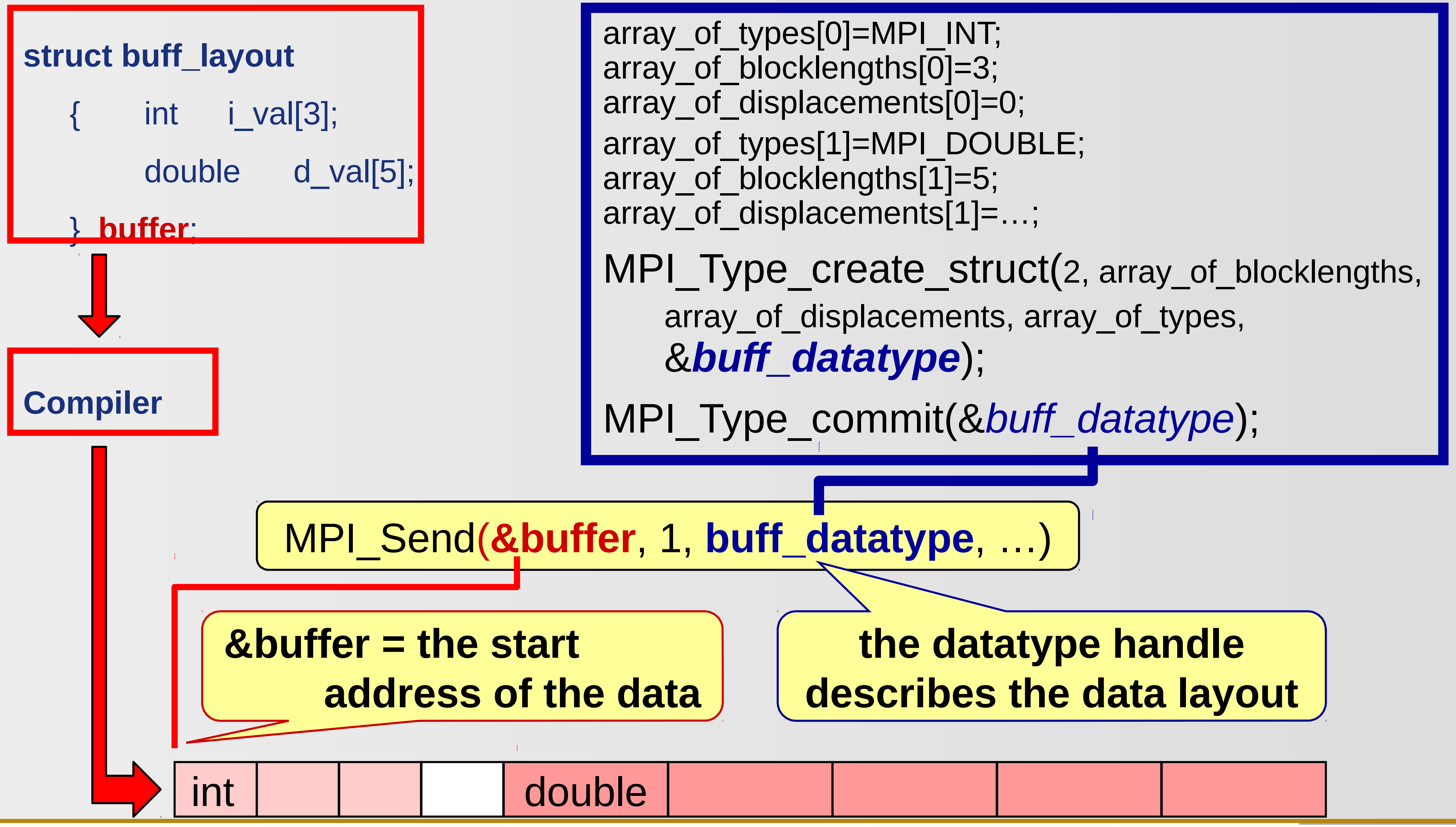
Message passing:

- **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

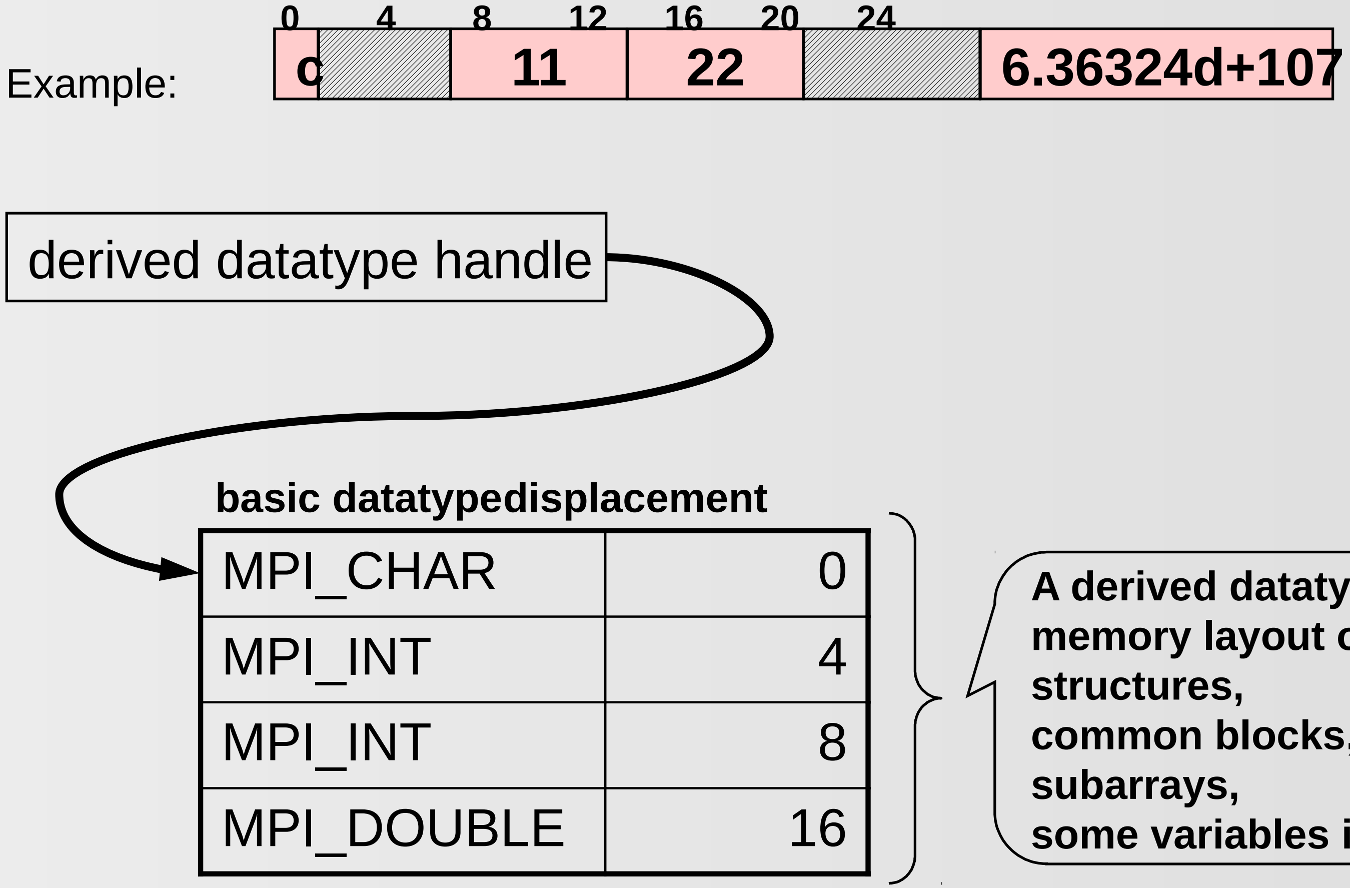
- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

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)*

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

Derived Datatypes — Type Maps



Example:

derived datatype handle

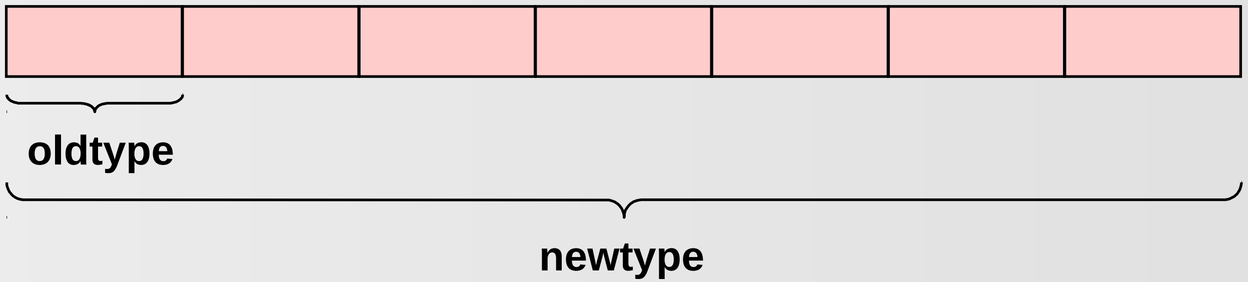
basic datatypedisplacement

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,
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

Handout only contains old style interface

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

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). (More than once use equivalent to additional no-operations.)

C

Fortran

- 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
```

IN-OUT argument

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

Exercise 1 — Derived Datatypes

Exercise 1

- Use C C/Ch12/derived-contiguous-skel.c

  or Fortran F_30/Ch12/derived-contiguous-skel_30.f90
- We us a modified pass-around-the-ring exercise:

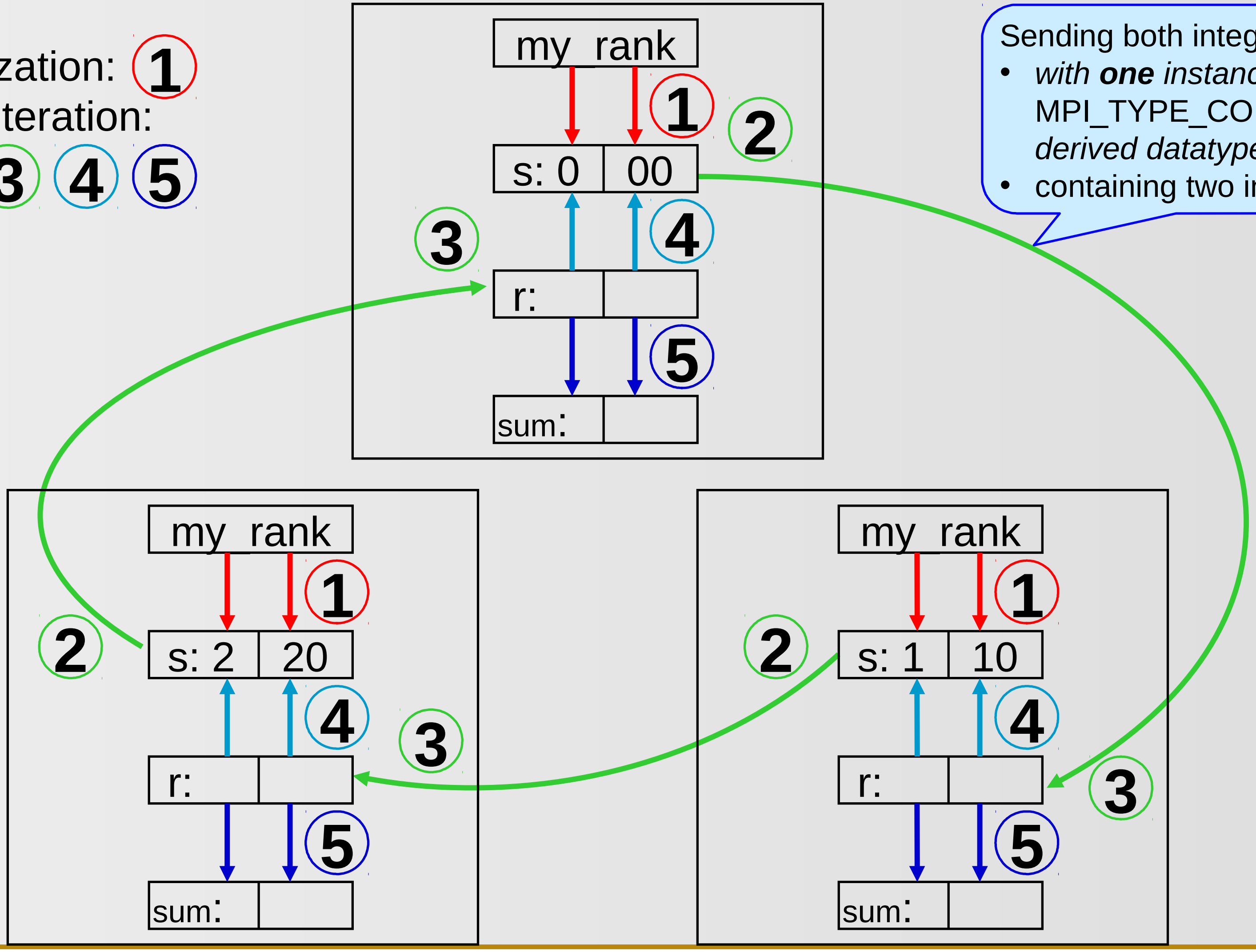
  It sends a struct with two integers
- They are initialized with **my_rank** and **10*my_rank**
- Therefore we calulate two separate sums.
- Currently, the data is send with the description
  - “snd_buf, 2, MPI_INTEGER”
- Please substitute this by using a
  - derived datatype
  - with a type map of “two integers”
  - Of course produced with the two routines on the previous slides

Exercise 1 — Derived Datatypes

Initialization: 1

Each iteration:

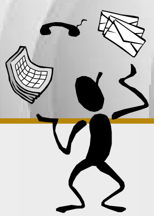
2 3 4 5



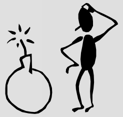
Sending both integers

- *with **one** instance of an* MPI_TYPE_CONTIGUOUS *derived datatype*
- containing two integers

During the Exercise



Please **stay here in the main room** while you do this exercise



And have fun with this **middle long** exercise

Please do not look at the solution before you finished this exercise,

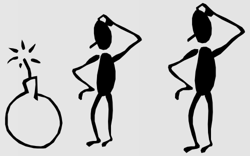
otherwise,

90% of your learning outcome may be lost

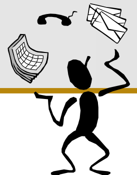
As soon as you finished the exercise,

please **go to your breakout room**

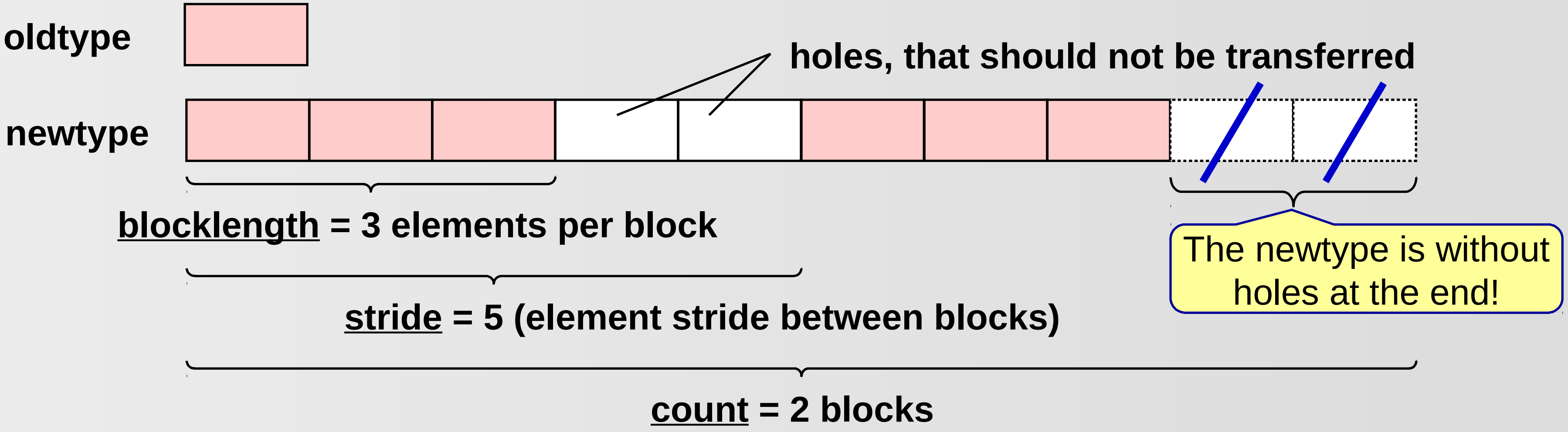
and continue your discussions with your fellow learners:



It looks easy, isn't it?



Vector Datatype



oldtype

newtype

holes, that should not be transferred

blocklength = 3 elements per block

stride = 5 (element stride between blocks)

count = 2 blocks

The newtype is without holes at the end!

C

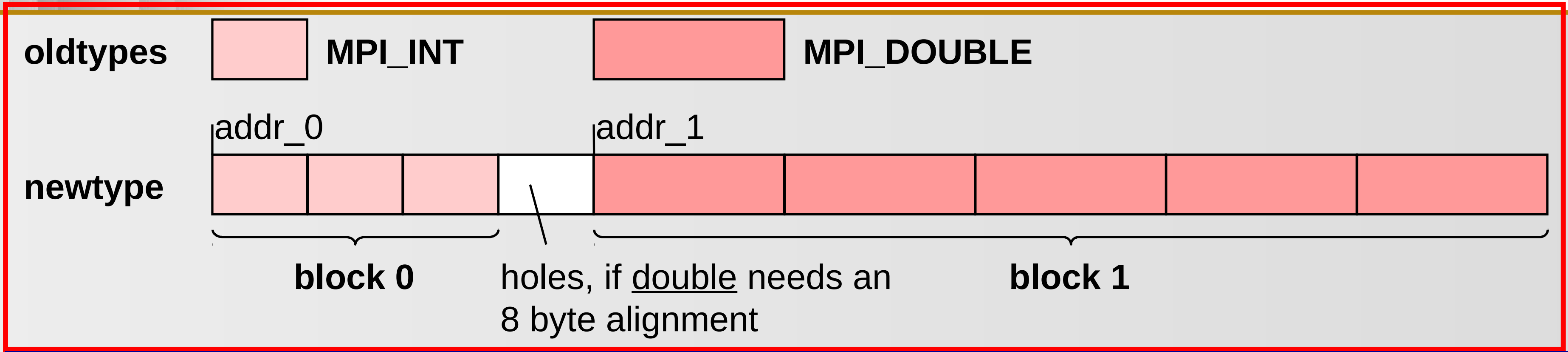
Fortran

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

Struct Datatype



oldtypes MPI_INT MPI_DOUBLE

newtype

addr_0

addr_1

block 0

holes, if double needs an 8 byte alignment

block 1

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*)

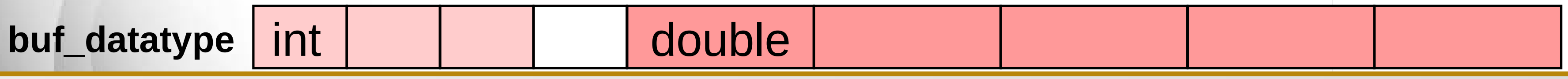
Fortran

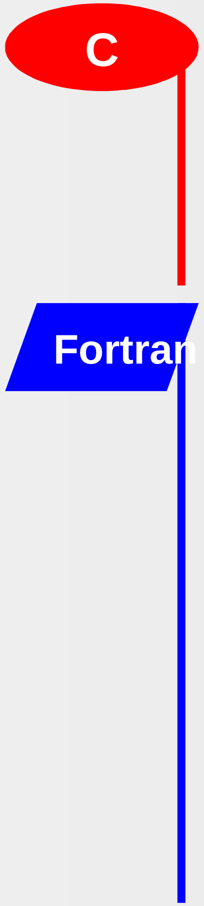
- Fortran: MPI_TYPE_CREATE_STRUCT(count,
  array_of_blocklengths, array_of_displacements [1] ,
  array_of_types, *newtype*, *ierror*)

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

[1] INTEGER**(KIND=MPI_ADDRESS_KIND)** array_of_displacements

Memory Layout of Struct Datatypes





Fixed memory layout:

- C

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

- Fortran, common block

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

- Fortran, derived types

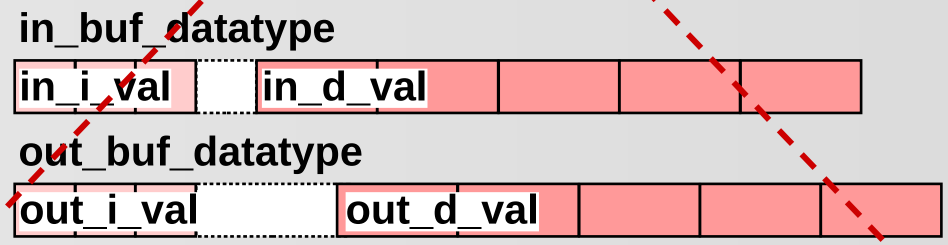
```
    TYPE buff_type
      SEQUENCE
      INTEGER, DIMENS !!! ):: i_val
      DOUBLE PRECISION, &
                 DIMENSION(5):: d_val
    END TYPE buff_type
    TYPE (buff_type) :: buff_variable
```

Alternative, in MPI-3.0:

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

~~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 optimi.:** MPI_Send & _Recv of **…d_val** is invisible for the compiler add MPI_Address~~



Not portable, because address differences are allowed only inside of structures or arrays MPI-3.1, 4.1.12

How to compute the displacement (1)

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

Retrieve an absolute address:

C

Fortran

- 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
```

How to compute the displacement (2)

New in MPI-3.1

C

Fortran

Relative displacement:= absolute address 1 – absolute address 2

- 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

New in MPI-3.1

C

Fortran

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

- 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

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

C

New in MPI-3.1

```
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
```

Fortran

New in MPI-3.1

```
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
```

See also MPI-3.1, Example 4.8, page 102 and Example 4.17, pp 125-127

Performance options

Which is the fastest neighbor communication with strided data?

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

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

Exercise 2-4

- 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
```

- 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

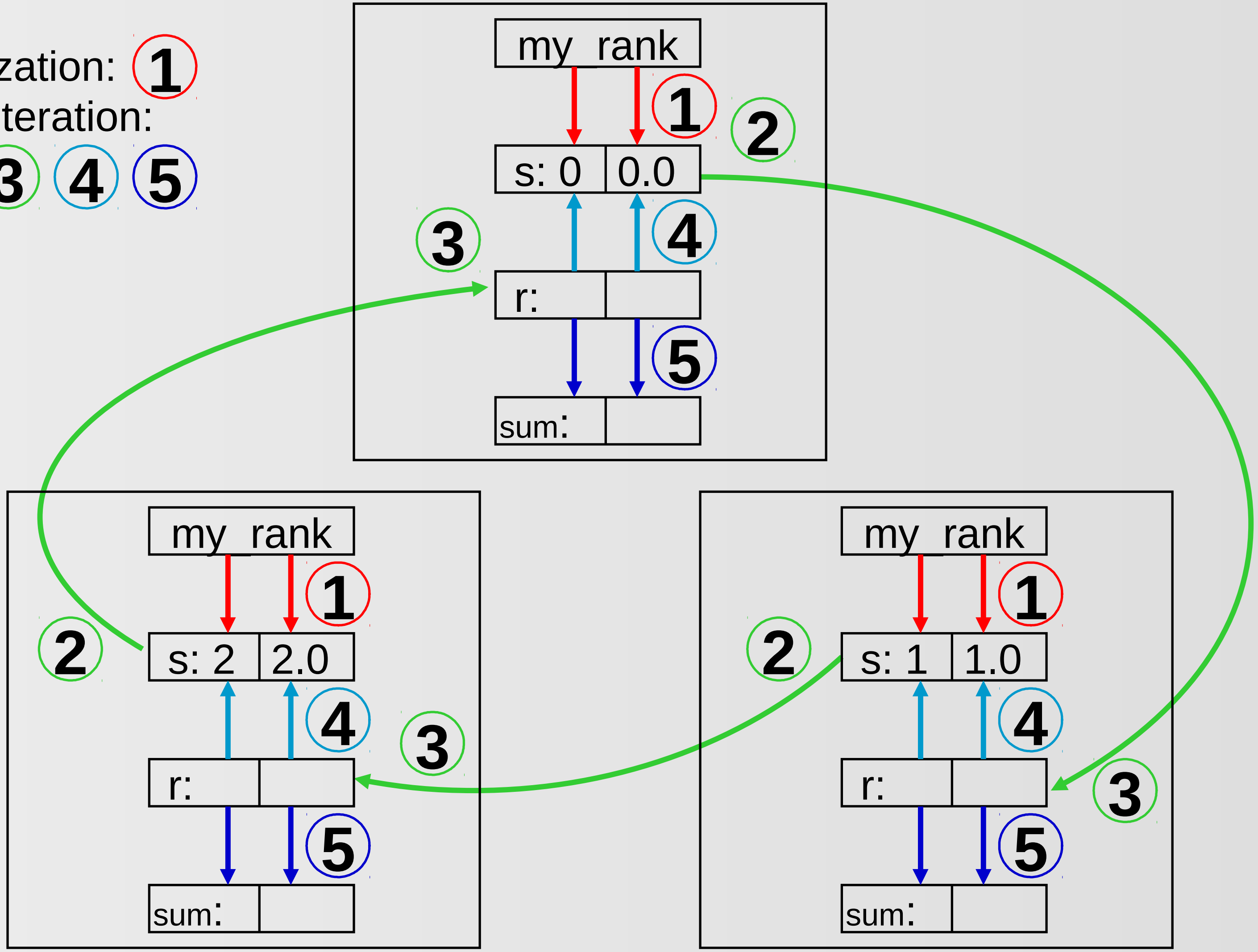
  and modify the second part, i.e., steps 1-5 of the ring example

Exercise 2 — Derived Datatypes

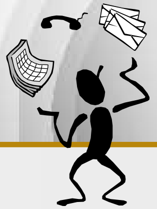
Initialization: 1

Each iteration:

2 3 4 5



During the Exercise



Please **stay here in the main room** while you do this exercise

And have fun with this **middle long** exercise

Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

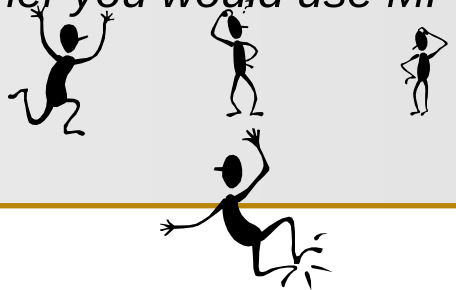
As soon as you finished the exercise,

please **go to your breakout room**

and continue your discussions with your fellow learners:

If you want, you can share your thoughts about

whether you would use MPI 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.)
   - Solution: MPI/tasks/C/Ch12/solutions/derived-struct-advanced-sendrecv.c

     and MPI/tasks/F_30/Ch12/solutions/derived-struct-advanced-sendrecv_30.f90

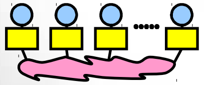
4. Substitute MPI_Sendrecv by **MPI_Sendrecv_replace**:
   - Three steps are now combined:
   - The receive buffer (rcv_buf) must be removed.
   - The iteration is now reduced to three statements: ② ③ ④
     - MPI_Sendrecv_replace to pass the ranks around the ring,
     - computing the integer sum,
     - computing the floating point sum.
   - Solution: MPI/tasks/C/Ch12/solutions/derived-struct-advanced-sendrecv-replace.c

     and MPI/tasks/F_30/Ch12/solutions/derived-struct-advanced-sendrecv-replace_30.f90

Derived Datatypes (2nd part)

Derived Datatypes (2nd part)

1. MPI Overview



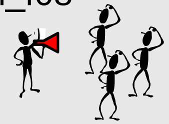
2. Process model and language bindings

MPI_Init()
MPI_Comm_rank()

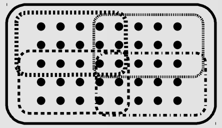
3. Messages and point-to-point communication

4. Nonblocking communication

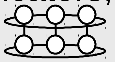
5. The New Fortran Module mpi_f08
6. Collective communication



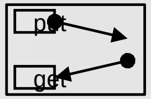
7. Error Handling



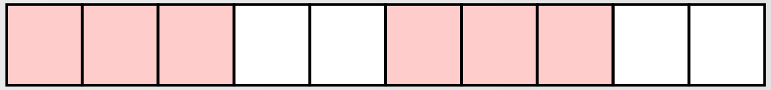
8. Groups & communicators, environment management
9. Virtual topologies



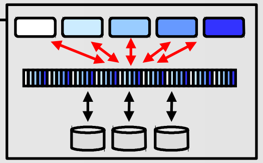
10. One-sided communication



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

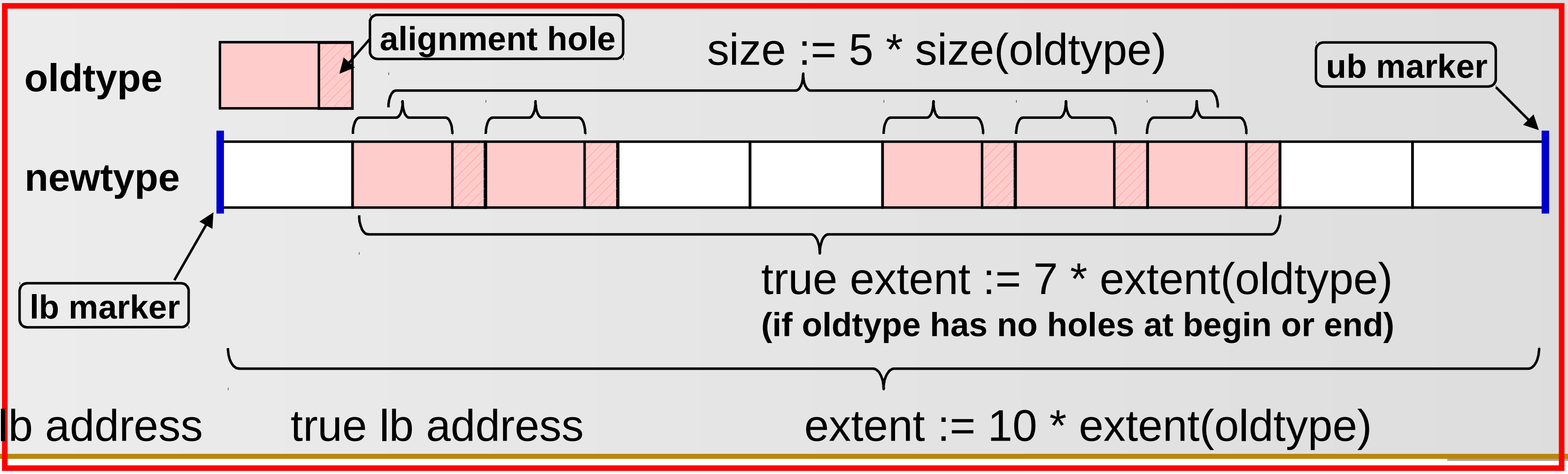


14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

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 by lb and ub markers: MPI_TYPE_CREATE_RESIZED
- Basic datatypes: Size = Extent = number of bytes used by the compiler.

Example:s



oldtype
alignment hole
size := 5 * size(oldtype)
ub marker
newtype
lb marker
true extent := 7 * extent(oldtype)
(if oldtype has no holes at begin or end)
lb address
true lb address
extent := 10 * extent(oldtype)

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 and MPI-3.1, 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 different sizes)!
  - See MPI-3.0 / MPI-3.1, Sect. 4.1.6, Advice to users on page 106
- Alignment hole at the end is important when using an array of structures!
- Implication **(for C and Fortran!)**:
  - 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
  - applies additionally MPI_TYPE_CREATE_RESIZED to this datatype handle.
  - See Example in MPI-3.0 / MPI-3.1, Sect. 17.1.15 on pages 629-630 / 637-638.
- 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-3.0 / MPI-3.1, Sect. 4.1.6, Advice to users on page 106 / 106

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: Call MPI_Type_create_resized with lb=0 and new_extent=sizeof(one structure), or use the following method:
  - & 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)
```

Alignment rule, holes and resizing of structures (3)

- Correctness problem with array of structures (continued):
  - 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**

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):
  - 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**

Fortran struct with SEQUENCE attribute

- 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 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

New in MPI-3.0

- 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**

— skipped —

All Derived Datatype Creation Routines (1)

- **MPI_Type_contiguous()**
   already discussed
- **MPI_Type_vector()**　**MPI_Type_create_hvector()**
   already discussed　 stride as byte size
- **MPI_Type_indexed()**　**MPI_Type_create_hindexed()**
   similar to .._struct(),　 with byte displacements
  same oldtype for all sub-blocks,
  displacements based on 0-based index in “array of oldtype”
- **MPI_Type_create_indexed_block()MPI_Type_create_hindexed_block()**
   same as MPI_Type_indexed() with byte displacements
  but same block length
  for each sub-block
- **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*

<u>Removed MPI-1 interfaces</u>	<u>substituted by</u>
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_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-3.0 / MPI-3.1, Section 2.5.4, page 15 line 42/45 – page 16 line 3/6

    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

    □ MPI_F_SYNC_REG 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?

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

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

Exercises 5+6

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**

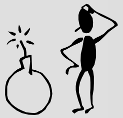
5. **Compile and test** with different compilers and accompanying MPI libraries
   - Pipe the stdout to: `| sort +0 -1  -n  +1 -2`
   - Example:

```
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.

During the Exercise

Please **stay here in the main room** while you do this exercise

And have fun with this **middle long** exercise 

Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

As soon as you finished the exercise,

please **go to your breakout room**

and continue your discussions with your fellow learners:

I recommend that you ***directly go to your breakout room***

to *exchange your* *questions, remarks* *and results* *with your colleagues.*

APPENDIX: Solution to exercises

Chapter 12-(1), Exercise 1: MPI_TYPE_CONTIGUOUS

MPI/tasks/F_30/Ch12/solutions/derived_contiguous_30.f90

Fortran

```
TYPE t
   SEQUENCE
   INTEGER :: i
   INTEGER :: j
END TYPE t
```

Provided in the skeleton

```
MPI_Datatype send_recv_type;
```

```
MPI_Type_contiguous(2, MPI_INT, &send_recv_type);
CALL MPI_Type_commit(send_recv_type)
```

```
sum%i = 0           ; sum%r = 0 ;
snd_buf%i = my_rank ; snd_buf%j = my_rank
DO i = 1, size
  CALL MPI_Issend(snd_buf,1,send_recv_type,right,17,MPI_COMM_WORLD,request)
  CALL MPI_Recv ( rcv_buf,1,send_recv_type,left, 17,MPI_COMM_WORLD,status)
  CALL MPI_Wait(request, status)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
  snd_buf = rcv_buf
  sum%i = sum%i + rcv_buf%i ;  sum%j = sum%j + rcv_buf%j
END DO
WRITE(*,*) 'PE', my_rank, ': Sum%i =', sum%i, ' Sum%j =', sum%j
```

Chapter 12-(1), Exercise 2: Halo-copy with derived types

C

MPI/tasks/C/Ch12/solutions/derived-struct.c

Provided in the skeleton

```
struct buff{
     int    i;
     float f;
} snd_buf, rcv_buf, sum;
int      array_of_blocklengths[2];
MPI_Aint array_of_displacements[2], first_var_address, second_var_address;
MPI_Datatype array_of_types[2], send_recv_type;

array_of_types[0] = MPI_INT;  array_of_types[1] = MPI_FLOAT;
array_of_blocklengths[0] = 1;  array_of_blocklengths[1] = 1;
MPI_Get_address(&snd_buf.i, &first_var_address);
MPI_Get_address(&snd_buf.f, &second_var_address);
array_of_displacements[0] = (MPI_Aint) 0;
array_of_displacements[1]=MPI_Aint_diff(second_var_address-first_var_address);
MPI_Type_create_struct(2, array_of_blocklengths,  array_of_displacements,
                          array_of_types, &send_recv_type);
MPI_Type_commit(&send_recv_type);

sum.i = 0;                  sum.f = 0;
snd_buf.i = my_rank;  snd_buf.f = 10*my_rank;
for( i = 0; i < size; i++)
{ MPI_Issend(&snd_buf,1,send_recv_type,right,17,MPI_COMM_WORLD, &request);
  MPI_Recv ( &rcv_buf,1,send_recv_type,left, 17,MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;
  sum.i += rcv_buf.i;  sum.f += rcv_buf.f;
}
printf ("PE %i: Sum = %i and %f \n", my_rank, sum.i, sum.f);
```

Chapter 12-(1), Exercise 2: Halo-copy with derived types

Fortran

MPI/tasks/F_30/Ch12/solutions/derived_struct_30.f90

Provided in the skeleton

```
TYPE t
   SEQUENCE
   INTEGER :: i
   REAL    :: r
END TYPE t
TYPE(t), ASYNCHRONOUS :: snd_buf
TYPE(t) :: rcv_buf, sum
TYPE(MPI_Datatype) :: send_recv_type

INTEGER(KIND=MPI_ADDRESS_KIND) :: array_of_displacements(2)
INTEGER(KIND=MPI_ADDRESS_KIND) :: first_var_address, second_var_address

CALL MPI_Get_address(snd_buf%i, first_var_address)
CALL MPI_Get_address(snd_buf%r, second_var_address)
array_of_displacements(1) = 0
array_of_displacements(2)=MPI_Aint_diff(second_var_address-first_var_address)

CALL MPI_Type_create_struct(2, (/1,1/), &
 &      array_of_displacements, (/MPI_INTEGER,MPI_REAL/), send_recv_type)
CALL MPI_Type_commit(send_recv_type)

sum%i = 0           ; sum%r = 0 ;
snd_buf%i = my_rank ; snd_buf%r = REAL(10*my_rank)
DO i = 1, size
  CALL MPI_Issend(snd_buf,1,send_recv_type,right,17,MPI_COMM_WORLD,request)
  CALL MPI_Recv ( rcv_buf,1,send_recv_type,left, 17,MPI_COMM_WORLD,status)
  CALL MPI_Wait(request, status)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
  snd_buf = rcv_buf
  sum%i = sum%i + rcv_buf%i ;  sum%r = sum%r + rcv_buf%r
END DO
WRITE(*,*) 'PE', my_rank, ': Sum%i =', sum%i, ' Sum%r =', sum%r
```

Chapter 12-(2), Exercises 5+6:Resizing of derived types (major changes)

C

MPI/tasks/C/Ch12/solutions/derived-struct-double+int-resized.c

```
MPI_Datatype ... send_recv_type, send_recv_resized;
```

```
MPI_Type_create_struct(COUNT, ..., &send_recv_type);
MPI_Type_create_resized(send_recv_type,
         (MPI_Aint) 0, (MPI_Aint) sizeof(snd_buf[0]), &send_recv_resized);
MPI_Type_commit(&send_recv_resized);
```

```
  MPI_Issend(&snd_buf, arr_lng-1, send_recv_resized, ...
```

Fortran

MPI/tasks/F_30/Ch12/solutions/derived-struct-dp+integer-resized_30.f90
MPI/tasks/F_30/Ch12/solutions/derived-struct-dp+integer-bindC-resized_30.f90

```
TYPE(MPI_Datatype) :: send_recv_type, send_recv_resized
```

```
CALL MPI_Type_create_struct(2, ..., send_recv_type)
CALL MPI_Get_address(snd_buf(1), first_var_address)
CALL MPI_Get_address(snd_buf(2), second_var_address)
lb = 0;   extent = MPI_Aint_diff(second_var_address, first_var_address)
CALL MPI_Type_create_resized(send_recv_type, lb,extent, send_recv_resized)
CALL MPI_Type_commit(send_recv_resized)
```

```
   CALL MPI_Issend(snd_buf, arr_lng-1, send_recv_resized, ...
```

Thanks!

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro