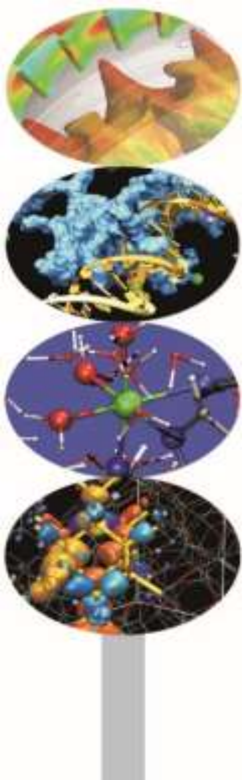
MPI Derived Datatypes



SuperComputing Applications and Innovation Department

Courses Edition 2017

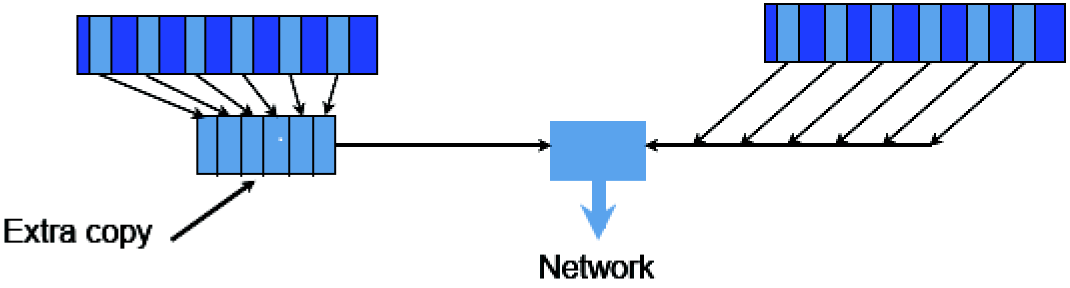
Derived datatypes

You may need to send messages that contain

1. non-contiguous data of a single type (e.g. a sub-block of a matrix)
2. contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers)
3. non-contiguous data of mixed types

Datatype solution:

1. The idea of MPI derived datatypes is to provide a simple, portable, elegant and efficient way of communicating non-contiguous or mixed types in a message.
   - During the communication, the datatype tells MPI system where to take the data when sending or where to put data when receiving.
2. The actual performances depend on the MPI implementation
3. **Derived datatypes are also needed for getting the most out of MPI-I/O.**



Derived datatypes

What are?

MPI Derived datatypes are datatypes that are built from the basic MPI datatypes (e.g. MPI_INT, MPI_REAL, …)

Why datatypes?

- Specifying application-oriented layout of data in memory
  - can reduce memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available
- Specifying application-oriented layout of data on a file can reduce systems calls and physical disk I/O

Definition

A **general datatype** is an **opaque object** able to describe a buffer layout in memory by specifying:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements.

Typemap = {(type 0, displ 0), … (type n-1, displ n-1)}

– pairs of basic types and displacements (in byte)

Type signature = {type 0, type 1, … type n-1}

– list of types in the typemap

– gives size of each elements

– tells MPI how to interpret sent and received bits

Displacement:

– tells MPI where to get (on sending) or put (on receiving) data

Typemap

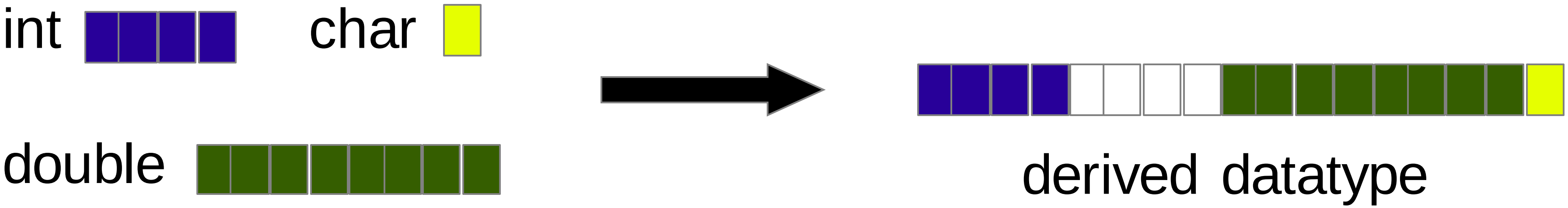
Example:

Basic datatypes are particular cases of a general datatype, and are predefined:

MPI_INT = {(int, 0)}

General datatype with typemap

Typemap = {(int,0), (double,8), (char,16)}



int

char

double

derived datatype

How to use

General datatypes are created and destroyed at run-time through calls to MPI library routines.

Implementation steps are:

1. Creation of the datatype from previous existing ones with a **datatype constructor**.
2. Allocation (**committing**) of the datatype before using it.
3. **Usage of the derived datatype** for MPI communications and/or for MPI-I/O
4. Deallocation (**freeing**) of the datatype after that it is no longer needed.

Construction of derived datatype

MPI_TYPE_CONTIGUOUS	contiguous datatype
MPI_TYPE_VECTOR	regularly spaced datatype
MPI_TYPE_CREATE_HVECTOR	like vector, but the stride is specified in byte
MPI_TYPE_INDEXED	variably spaced datatype
MPI_TYPE_CREATE_HINDEXED	like indexed, but the stride is specified in byte
MPI_TYPE_CREATE_INDEXED_BLOCK	particular case of the previous one
MPI_TYPE_CREATE_SUBARRAY	subarray within a multidimensional array
MPI_TYPE_CREATE_DARRAY	distribution of a ndim-array into a grid of ndim-logical processes
MPI_TYPE_CREATE_STRUCT	fully general datatype

Committing and freeing

MPI_TYPE_COMMIT (datatype)
INOUT datatype: datatype that is committed (handle)

New defined datatypes must be committed before its usage in any MPI communication

MPI_TYPE_FREE (datatype)
INOUT datatype: datatype that is freed (handle)

datatype will be deallocated when all pending operations are finished

MPI_TYPE_CONTIGUOUS

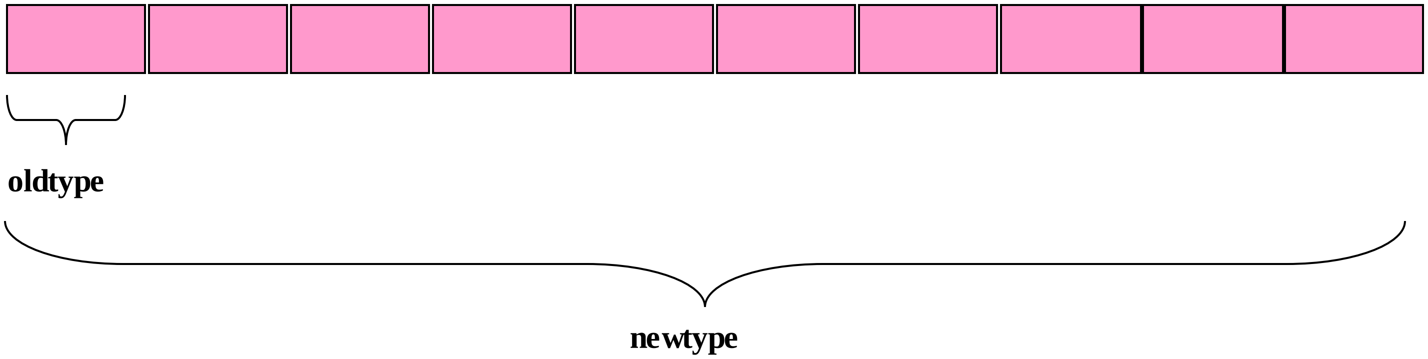
MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)

IN count: replication count (non-negative integer)

IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- MPI_TYPE_CONTIGOUS constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations.
- newtype is the datatype obtained by concatenating count copies of oldtype.



Example

```
count = 4;
MPI_Type_countiguous(count, MPI_FLOAT,  &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	**10.0**	**11.0**	**12.0**
13.0	14.0	15.0	16.0

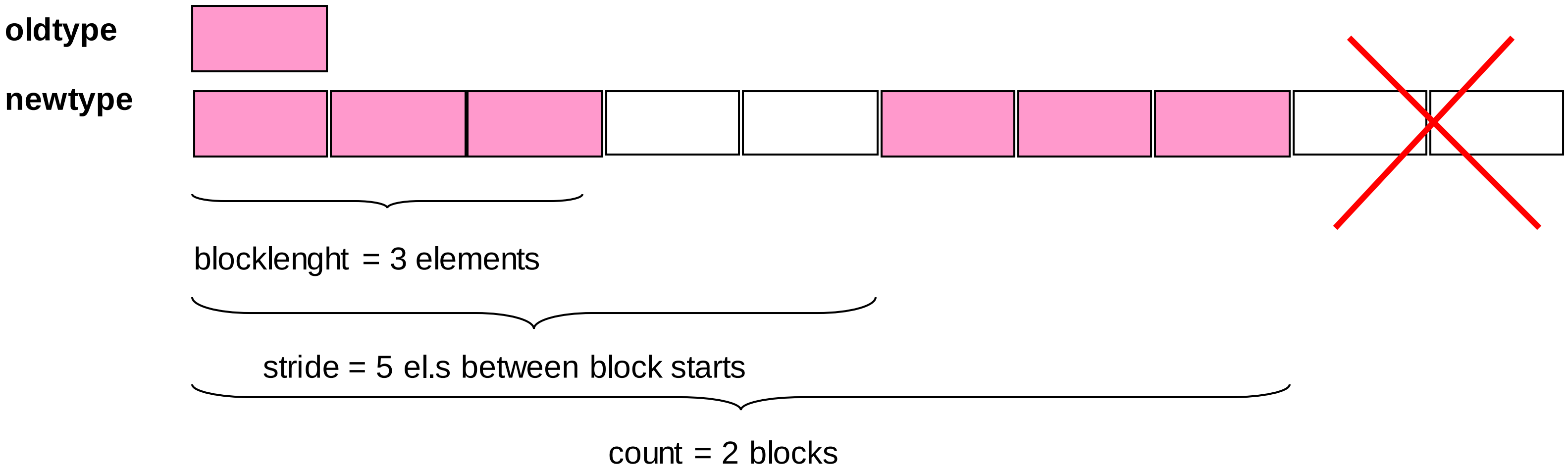
```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	**10.0**	**11.0**	**12.0**

MPI_TYPE_VECTOR

MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
IN count: Number of blocks (non-negative integer)
IN blocklen: Number of elements in each block (non-negative integer)
IN stride: Number of elements (NOT bytes) between start of each block (integer)
IN oldtype: Old datatype (handle)
OUT newtype: New datatype (handle)

- Consists of a number of elements of the same datatype repeated with a certain stride



oldtype

newtype

blocklenght = 3 elements

stride = 5 el.s between block starts

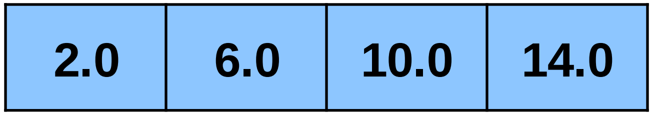
count = 2 blocks

Example

```
count = 4; blocklength = 1; stride = 4;
MPI_Type_vector (count, blocklength, stride, MPI_FLOAT, &columntype);
```

1.0	**2.0**	3.0	4.0
5.0	**6.0**	7.0	8.0
9.0	**10.0**	11.0	12.0
13.0	**14.0**	15.0	16.0

```
MPI_Send(&a[2][0], 1, columntype, dest, tag, comm);
```



2.0	**6.0**	**10.0**	**14.0**

MPI_TYPE_CREATE_HVECTOR

MPI_TYPE_CREATE_HVECTOR (count, blocklength, stride, oldtype, newtype)

IN count: Number of blocks (non-negative integer)
IN blocklen: Number of elements in each block (non-negative integer)
IN stride: Number of bytes between start of each block (integer)
IN oldtype: Old datatype (handle)
OUT newtype: New datatype (handle)

- It’s identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements
- “H” stands for heterogeneous

- In C, use MPI_Aint to specify offsets
- In Fortran, use type INTEGER(KIND=MPI_ADDRESS_KIND) to specify offsets

MPI_TYPE_INDEXED

MPI_TYPE_INDEXED (count, array_of_blocklengths, array_of_displacements,oldtype, newtype)

IN count: number of blocks – also number of entries in
array_of_blocklenghts and array_of_displacements
(non-negative integer)

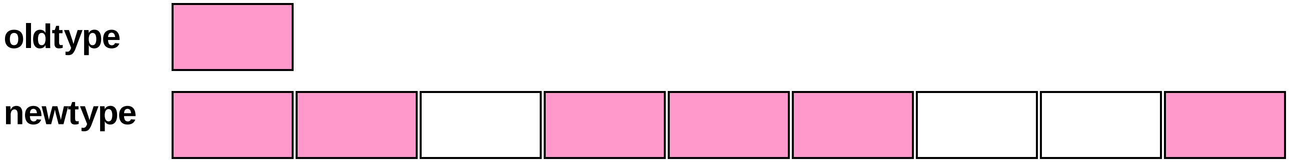
IN array_of_blocklengths: number of elements per block
(array of non-negative integers)

IN array_of_displacements: displacement for each block, in multiples of oldtype extent
(array of integer)

IN oldtype: old datatype (handle)

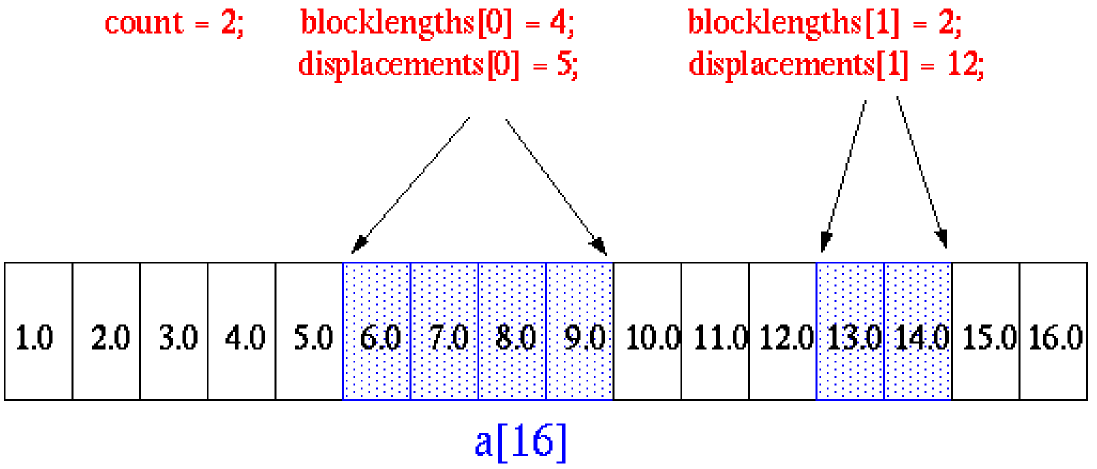
OUT newtype: new datatype (handle)

- Creates a new type from blocks comprising identical elements
- The size and displacements of the blocks can vary



count=3, array_of_blocklenghths=(/2,3,1/), array_of_displacements=(/0,3,8/)

Example 1



count = 2;

blocklengths[0] = 4;
displacements[0] = 5;

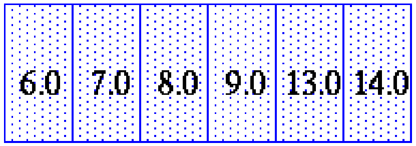
blocklengths[1] = 2;
displacements[1] = 12;

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0

a[16]

```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

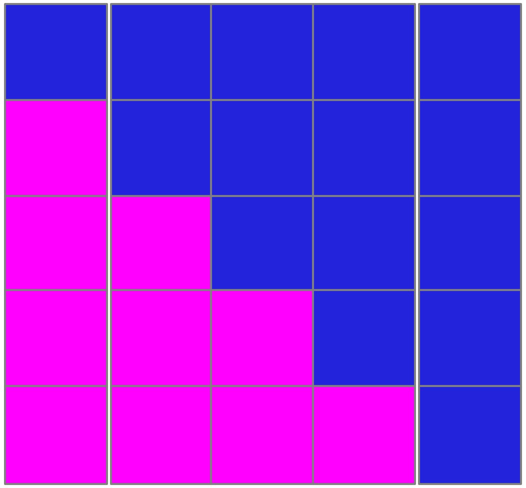
```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



6.0	7.0	8.0	9.0	13.0	14.0

1 element of
indextype

Example 2



```
/* upper triangular matrix */
double a[100][100];
int displ[100], blocklen[100], int i;
MPI_Datatype upper;

/* compute start and size of the rows */
for (i=0; i<100; i++){
        displ[i] = 100*i+i;
        blocklen[i] = 100-i;
}

/* create and commit a datatype for upper triangular matrix */
MPI_Type_indexed (100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit (&upper);

/* … send it ...*/
MPI_Send (a, 1, upper, dest, tag, MPI_COMM_WORLD);
MPI_Type_free (&upper);
```

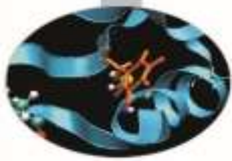
MPI_TYPE_CREATE_HINDEXED

MPI_TYPE_CREATE_HINDEXED (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN count: number of blocks – also number of entries in array_of_blocklengths and array_of_displacements (non-negative integer)
IN array_of_blocklengths: number of elements in each block (array of non-negative integers)
IN array_of_displacements: byte displacement of each block (array of integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)

- This function is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather that in multiples of the oldtype extent

MPI_TYPE_CREATE_INDEXED BLOCK



MPI_TYPE_CREATE_INDEXED_BLOCK (count, blocklengths,
array_of_displacements, oldtype, newtype)

IN count: length of array of displacements (non-negative integer)
IN blocklengths: size of block (non-negative integer)
IN array_of_displacements: array of displacements (array of integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)

- Similar to MPI_TYPE_INDEXED, except that the block-length is the same for all blocks.
- There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). This function allows for constant blocksize and arbitrary displacements.

MPI_TYPE_CREATE_SUBARRAY

MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN ndims: number of array dimensions (positive integer)

IN array_of_sizes: number of elements of type oldtype in each dimension of the full array (array of positive integers)

IN array_of_subsizes: number of elements of type oldtype in each dimension of the subarray (array of positive integers)

IN array_of_starts: starting coordinates of the subarray in each dimension (array of non-negative integers)

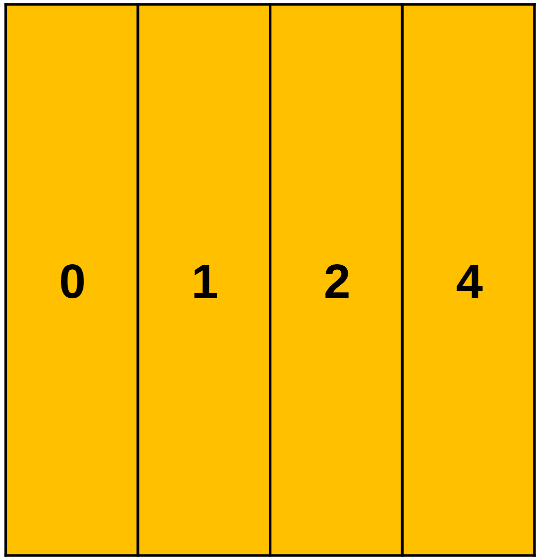
IN order: array storage order flag (state: MPI_ORDER_C or MPI_ORDER_FORTRAN)

IN oldtype: array element datatype (handle)

OUT newtype: new datatype (handle)

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

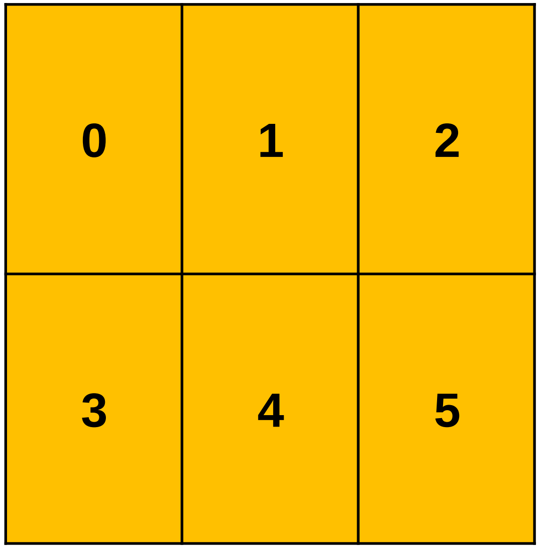
Example



MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

```
MPI_Datatype submatrix;
int gsizes[2], lsizes[2], starts[2];
int ndims[2] = { 4, 1 }; // MPI Cartesian process topology

gsizes[0] = 100; gsizes[1] = 100; // global matrix sizes

lsizes[0] = gsizes[0] / ndims[0]; // no reminder here
lsizes[1] = gsizes[1] /  ndims[1]; //no reminder here
starts[0] = 0;
starts[1] = rank*lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, starts,
    MPI_ORDER_C, MPI_DOUBLE, &submatrix);

MPI_Type_commit(&submatrix);
```

Example



0	1	2
3	4	5

MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

```
MPI_Datatype submatrix;
int gsizes[2], lsizes[2], starts[2];
int ndims[2] = { 2, 3}; // MPI Cartesian process topology

lsizes[0] = 200; lsizes[1] = 100; // local submatrix sizes

MPI_Cart_cords(comm, rank, 2, cords);

sizes[0] = ndims[0] * lsizes[0];
sizes[1] = ndims[1] * lsizes[1];
starts[0] = cords[0] * subsizes[0];
starts[1] = cords[1] * subsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, starts,
    MPI_ORDER_C, MPI_DOUBLE, &submatrix);

MPI_Type_commit(&submatrix);
```

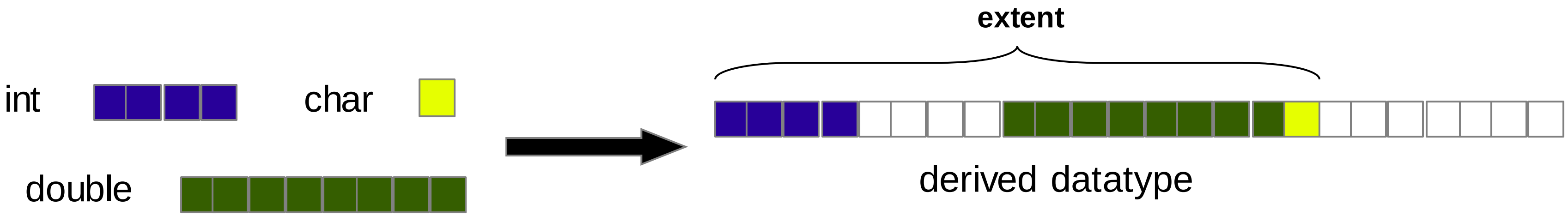
Size and Extent

The MPI datatype for structures – MPI_TYPE_CREATE_STRUCT – requires dealing with memory addresses and further concepts:

Typemap: pairs of basic types and displacements

Extent: The **extent** of a datatype is the span from the lower to the upper bound with data **(including mid "holes")**

Size: The **size** of a datatype is the net number of bytes to be transferred **(without "holes")**



Size vs. extent of a datatype

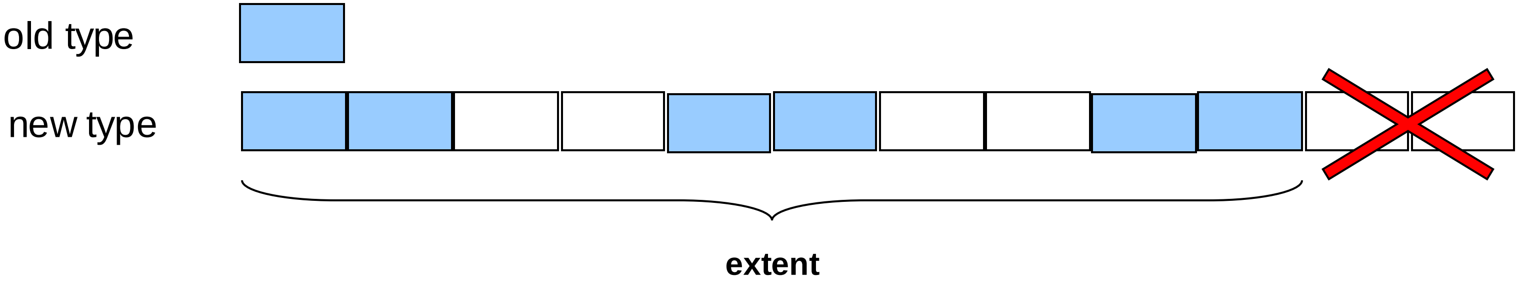
Basic datatypes:

- size = extent = number of bytes used by the compiler

Derived datatypes:

- extent include holes but...
- ... final holes are a figment of our imagination

MPI_TYPE_VECTOR (3, 2, 4, old_type, new_type)



- size = 6 x size of “old type”
- extent = 10 x extent of “old type”

Query size and extent of datatype

- Returns the total number of bytes of the entry datatype

MPI_TYPE_SIZE (datatype, size)

IN datatype: datatype (handle)

OUT size: datatype size (integer)

- Returns the lower bound and the extent of the entry datatype

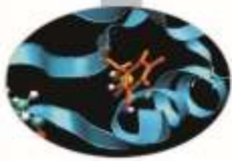
MPI_TYPE_GET_EXTENT (datatype, lb, extent)

IN datatype: datatype to get information on(handle)

OUT lb: lower bound of datatype (integer)

OUT extent: extent of datatype (integer)

Use of Datatype in Communications



- The extent of a datatype controls how the datatype is used with the **count > 1** field in MPI communications

```
call MPI_Send(buf,count,datatype,...)
```

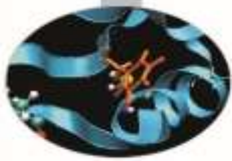
- What does actually send?

```
do i=0,count-1
    call MPI_Send(bufb(1+i*extent(datatype)),1,datatype,...)
enddo
```

where *bufb* is a byte type like *integer*1*

- *extent* is used to decide where to send from or where to receive to for count>1 datatype communications
  - Normally, this is right after the last byte used for (i-1)

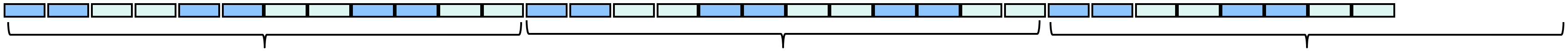
Use of Datatype in Communications



- Were we to use a derived datatype (i.e: MPI_TYPE_VECTOR) in a communication with count>1 of this derived datatype

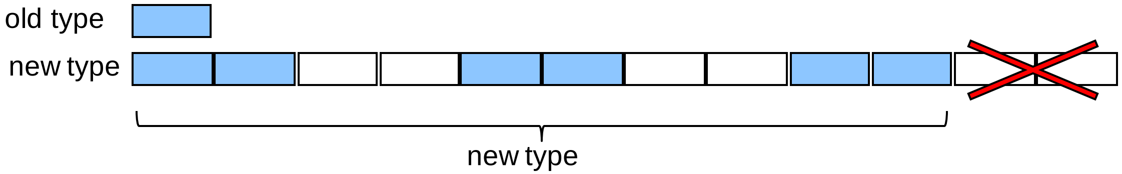
MPI_Send (buffer, 1000, new_type, ...)

...what we want



- Remember that trailing holes are discarded in datatype creation:

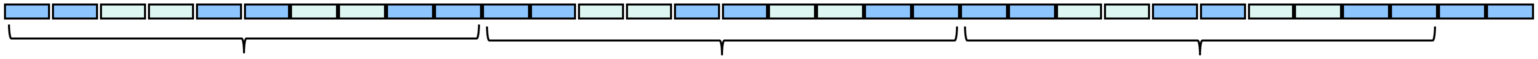
MPI_TYPE_VECTOR (3, 2, 4, old_type, new_type)



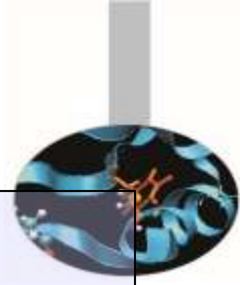
- beware of using communication of derived datatypes with count > 1

MPI_Send (buffer, 1000, new_type, ...)

...what we get ☹



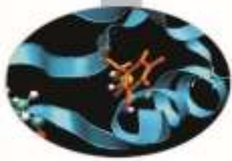
Resizing datatypes



MPI_TYPE_CREATE_RESIZED (oldtype, newlb, newextent, newtype)
IN oldtype: input datatype (handle)
IN newlb: new lower bound of datatype (integer, in terms of bytes)
IN newextent: new extent of datatype (integer, in term of bytes)
OUT newtype: output datatype (handle)

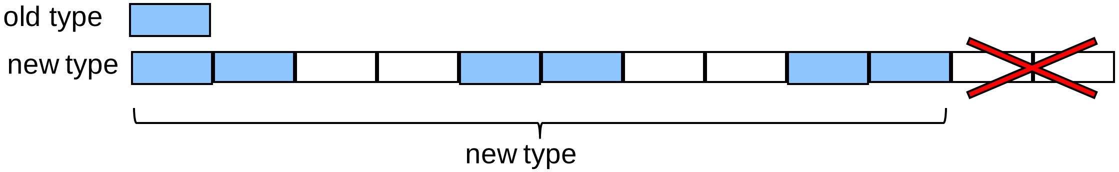
- Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be “lb”, and its upper bound is set to be “lb + extent”.
- Modifying extent is useful to handle alignments of the last items of structs
  - crucial when communicating more than one derived data-type
- Modifying also the lower bound can be confusing, use with particular care

Extend Datatype for Communications



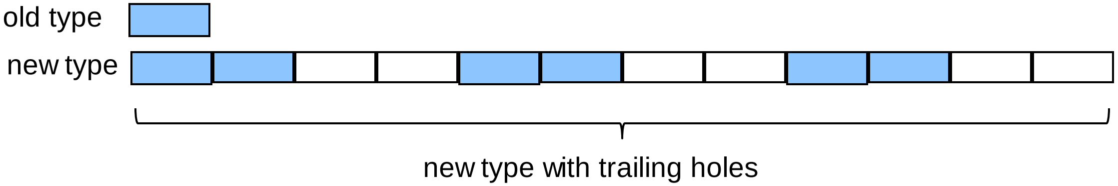
- If you want to use a derived datatype as a basic template in MPI communications, remember to take extent into account
  - trailing holes are discarded in datatype creation

MPI_TYPE_VECTOR (3, 2, 4, old_type, vec_type)



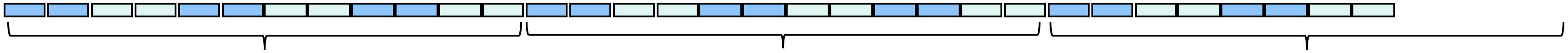
- extend the created datatype with the MPI_TYPE_CREATE_RESIZED

MPI_TYPE_CREATE_RESIZED (vec_type, 0, 12*old_type, vec_type_extended)



- perform MPI communication as usual

MPI_Send (buffer, 1000, vec_type_extended, ...)



MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_STRUCT (count, array_of_blocklengths,
array_of_displacements, array_of_oldtypes, newtype)

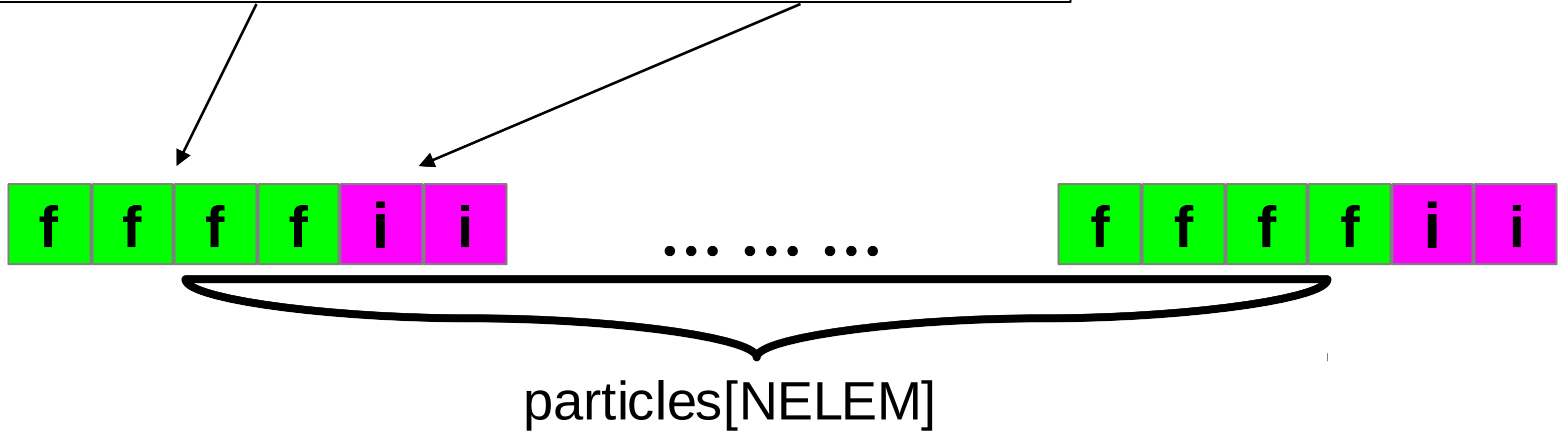
IN count: number of blocks (non-negative integer) -- also number of entries the following arrays
IN array_of_blocklenghts: number of elements in each block
(array of non-negative integer)
IN array_of_displacements: byte displacement of each block
(array of integer)
IN array_of_oldtypes: type of elements in each block
(array of handles to datatype objects)
OUT newtype: new datatype (handle)

- This subroutine returns a new datatype that represents count blocks. Each block is defined by an entry in array_of_blocklengths, array_of_displacements and array_of_types.
- **Displacements are expressed in bytes** (since the type representation can change!)
- To gather a mix of different datatypes scattered at many locations in space into one datatype that can be used for the communication.

Using extents (not safe)

```
struct {
   float x, y, z, velocity;
   int n, type;
} Particle;

Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);

count = 2;
blockcounts[0] = 4;           blockcount[1] = 2;
oldtypes[0]= MPI_FLOAT;       oldtypes[1] = MPI_INT;
displ[0] = 0;                 displ[1] = 4*extent;
```



```
MPI_Type_struct (count, blockcounts, displ, oldtypes, &particletype);
MPI_Type_commit(&particletype);
```

Using extents (not safe)/ 2

```
struct {
   float x, y, z, velocity;
   int n, type;
} Particle;

Particle particles[NELEM];
```

```
int count, blockcounts[2];
MPI_Aint displ[2];
MPI_Datatype particletype, oldtypes[2];

count = 2;
blockcounts[0] = 4; blockcount[1] = 2;
oldtypes[0]= MPI_FLOAT; oldtypes[1] = MPI_INT;

MPI_Type_extent(MPI_FLOAT, &extent);
displ[0] = 0; displ[1] = 4*extent;

MPI_Type_create_struct (count, blockcounts, displ, oldtypes,
                                              &particletype);

MPI_Type_commit(&particletype);

MPI_Send (particles, NELEM, particletype, dest, tag,
                     MPI_COMM_WORLD);

MPI_Free(&particletype);
```

Mind the alignments!

WARNING: C structs may be padded by the compiler, e.g.

```
struct mystruct {
    char a;
    int b;
    char c;
} x
```

⇨

```
struct mystruct {
    char a;
    char gap_0[3];
    int b;
    char c;
    char gap_1[3];
} x
```

Using extents to handle structs is not safe! Get the addresses

MPI_GET_ADDRESS (location, address)

IN location: location in caller memory (choice)

OUT address: address of location (integer)

- The address of the variable is returned, which can then be used to determine the correct relative displacements
- Using this function helps portability

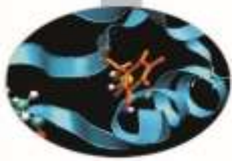
Using displacements (in bytes)

```
struct PartStruct {
   char class;
   double d[6];
   int b[7];
} particle[100];
```

- **MPI_Get_address can handles inner padding...**
- **... but not trailing padding (important when sending more than one struct)**

```
MPI_Datatype ParticleType;
int count = 3;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3]; // used to track address offsets in MPI

MPI_Get_address(&particle[0].class, &disp[0]);
MPI_Get_address(&particle[0].d, &disp[1]);
MPI_Get_address(&particle[0].b, &disp[2]);

/* Make displacements relative */
disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;

MPI_Type_create_struct (count, blocklen, disp, type,
      &ParticleType);
MPI_Type_commit (&ParticleType);

MPI_Send(particle, 100, ParticleType, dest, tag, comm);
MPI_Type_free (&ParticleType);
```

Extend Structure for Communications



```
/* Sending an array of structs portably */
struct PartStruct particle[100];
MPI_Datatype ParticleType;
…

/* check that the extent is correct */
MPI_Type_get_extent(ParticleType, &lb, &extent);
If ( extent != sizeof(particle[0]) ) {
        MPI_Datatype old = ParticleType;
        MPI_Type_create_resized ( old, 0, sizeof(particle[0]), &ParticleType);
        MPI_Type_free(&old);
}
MPI_Type_commit ( &ParticleType);
```

Fortran types

- According to the fortran standard, memory layout for variable allocation is much more liberal than C language
- An array of types, may be implemented as 5 arrays of scalars!

```
type particle
   real :: x,y,z,velocity
   integer :: n
end type particle
type(particle) :: particles(Np)
```

```
type particle
   sequence
   real :: x,y,z,velocity
   integer :: n
end type particle
type(particle) :: particles(Np)
```

- The memory layout is guaranteed using sequence or bind(C) type attributes
  - Or by using the (old style) commons...
- With Fortran 2003, MPI_Type_create_struct may be applied to common blocks, sequence and bind(C) derived types
  - it is implementation dependent how the MPI implementation computes the alignments (sequence, bind(C) or other)
- The possibility of passing **particles** as a type depends on MPI implementation: try **particle%x** and/or study the MPI standard and Fortran 2008 constructs

Fortran type example

```
call MPI_GET_ADDRESS(foo%i,  disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%l,  disp(4), ierr)
base = disp(1)
disp(1) = disp(1) – base ; disp(2) = disp(2) - base
disp(3) = disp(3) – base ; disp(4) = disp(4) - base
blocklen(1) = 1 ; blocklen(2) = 1
blocklen(3) = 1 ; blocklen(4) = 1
type(1) = MPI_INTEGER ;                   type(2) = MPI_REAL
type(3) = MPI_DOUBLE_PRECISION ; type(4) = MPI_LOGICAL
call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
call MPI_TYPE_COMMIT(newtype, ierr)
call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
! or
call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
! expects that base == address(foo%i) == address(foo)
call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
extent = disp(2) - disp(1) ; lb = 0
call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
call MPI_TYPE_COMMIT(newarrtype, ierr)
call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
```

Why using Packed data

- Derived datatypes allow, in most cases, to avoid explicit packing and unpacking
  - the user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer.
- But packed data are provided for many reasons
- **Compatibility**
  - with previous libraries
  - development of additional communication libraries layered on top of MPI
- **Dynamic behaviour**
  - a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part
  - In MPI_Unpack, the count argument specifies the actual number of items that are unpacked (in MPI_Recv the count argument specifies the maximum number of items that can be received)
- **Buffering**
  - outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy
  - buffering is not the evil: explicit buffering may allow to implement efficient MPI patterns

Packing data for heterogenous types

- The user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it.
- **MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm)**
  - pack: "incount" data of type "datatype" from buffer "inbuf"
  - to: contiguous storage area "outbuf" containing "outsize" **bytes**
  - "position" is the first location in the output buffer to be used for packing, updated when exiting from MPI_Pack
  - the communication "comm" to be used in the next has to be specified
- **MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm)**
  - Unpacks a message into the receive buffer specified by "outbuf, outcount, datatype" from the buffer space specified by "inbuf and insize"
- **MPI_PACKED** is the datatype of packed data, to be send/received

MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position, comm)

inbuf - input buffer start (choice)
incount - number of input data items (integer)
datatype - datatype of each input data item (handle)
outbuf - output buffer start (choice)
outsize - output buffer size, in bytes (integer)
position - current position in buffer, in bytes (integer)
comm - communicator for packed message (handle)

MPI_UNPACK (inbuf, insize, position, outbuf, outcout, datatype, comm)

inbuf - input buffer start (choice)
insize - size of input buffer, in bytes (integer)
position - current position in buffer, in bytes (integer)
outbuf - output buffer start (choice)
outcount - output buffer size, in bytes (integer)
datatype - datatype of each input data item (handle)
comm - communicator for packed message (handle)

Using MPI_Pack/Unpack

```
int     n;  float   a, b;
int     position; char    buffer[100];

if (myrank == 0){
     n = 4;      a = 1.;       b = 2.;   position = 0;
     // packing
     MPI_Pack(&a, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
     MPI_Pack(&b, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
     MPI_Pack(&n, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);
     // communication
     MPI_Bcast(buffer, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
 } else {
    // communication
    MPI_Bcast(buffer, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
    position = 0;
    // unpacking
    MPI_Unpack(buffer, 100, &position, &a, 1, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack(buffer, 100, &position, &b, 1, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack(buffer, 100, &position, &n, 1, MPI_INT, MPI_COMM_WORLD);
}
```

Hands-on: using GALILEO

- Connect to GALILEO front-end:

```
# ssh <user_name>@login.galileo.cineca.it
```

- Load the modules to use compiler and MPI

```
# module av openmpi
# module load autoload openmpi
```

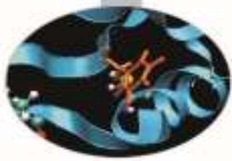
- write your programs using editors such as nano, emacs or vim
- compile you programs using the MPI compiler wrapper:

```
# mpicc my_source.c -o my_executable  # for C codes
# mpifort my_source.F90 -o my_executable  # for FORTRAN codes
```

- execute the program using N processes (if required):

```
# mpirun  -np <N>  ./my_executable
```

Hands-on: vectors and extents



Write a program working only with 2 MPI processes

- for each process, define a matrix A (nxn)
- rank=0 fills A with 0, while rank=1 fills with 1
- define a datatype to handle a column (C) or a row (Fortran) of A
- **extract size and extent of this type: is what you expect?**
- rank=0 sends to rank=1 the first column/row overwriting the A column/row
- check the results printing the matrix on screen
- modify the code to send the first nb columns/rows of A: **do you have to change the type? can you send two items of the new type?**

0	0	1	1	1
0	0	1	1	1
0	0	1	1	1
0	0	1	1	1
0	0	1	1	1

Final A for rank=1
n=5 ; nb=2

⇐ C

Fortran ⇒

0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Rights & Credits

Slides and examples were authored by:

- Luca Ferraro (l.ferraro@cineca.it)
- Francesco Salvadore (f.salvadore@cineca.it)