23rd Summer School on PARALLEL COMPUTING

Derived Datatypes

Giusy Muscianisi – g.muscianisi@cineca.it
Luca Ferraro – l.ferraro@cineca.it
Maurizio Cremonesi – m.cremonesi@cineca.it

SuperComputing Applications and Innovation Department

CINECA

Derived datatypes

What are?

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

Why datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication)
- 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

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

Derived datatypes

Possible solutions:

1. make multiple MPI calls to send and receive each data element

   → If advantageous, copy data to a buffer before sending it
2. use MPI_pack/MPI_Unpack to pack data and send packed data (datatype MPI_PACKED)
3. use MPI_BYTE to get around the datatype-matching rules. Like MPI_PACKED, MPI_BYTE can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte.

- Additional latency costs due to multiple calls
- Additional latency costs due to memory copy
- Not portable to a heterogeneous system using MPI_BYTE or MPI_PACKED

Packing data

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 (initially zero) in the output buffer to be used for packing, updated when exiting from MPI_Pack
  - the communicator "comm" to be used in sending has to be specified

Unpacking data

- **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

Using MPI_Pack/Unpack

```
int   position;   int   n;   float  a, b;   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);
}
```

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

Why using Packed data

- **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 allows for the implementation of efficient MPI patterns

Derived datatypes

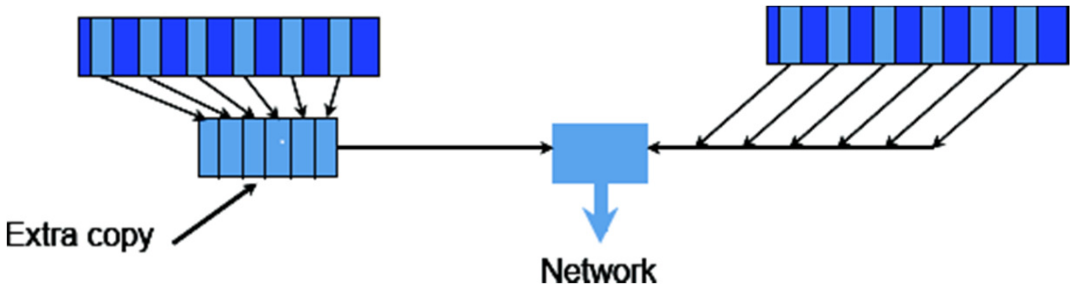
You may need to send messages that contain

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

Derived datatypes

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.**



Definition

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

- 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
– give size of each element
– tells MPI how to interpret the bits it sends and receives

Displacement:

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

Typemap

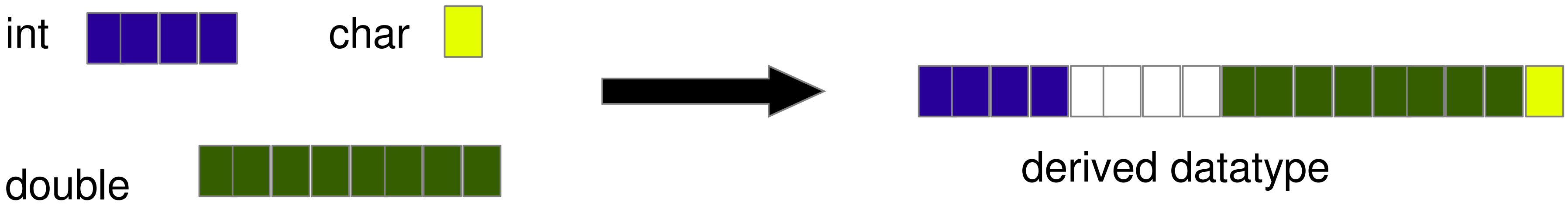
Example:

Basic datatype 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)}



How to use

General datatypes (differently from C or Fortran) are created (and destroyed) at run-time through calls to MPI library routines.

Implementation steps are:

1. Creation of the datatype from 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 **bytes**
• MPI_TYPE_INDEXED	**variably spaced** datatype
• MPI_TYPE_CREATE_HINDEXED	like indexed, but the stride is specified in **bytes**
• 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)

- Before it can be used in a communication or I/O call, each derived datatype has to be committed

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

- Mark a datatype for deallocation
- Datatype will be deallocated when all pending operations are finished

MPI_TYPE_CONTIGUOUS

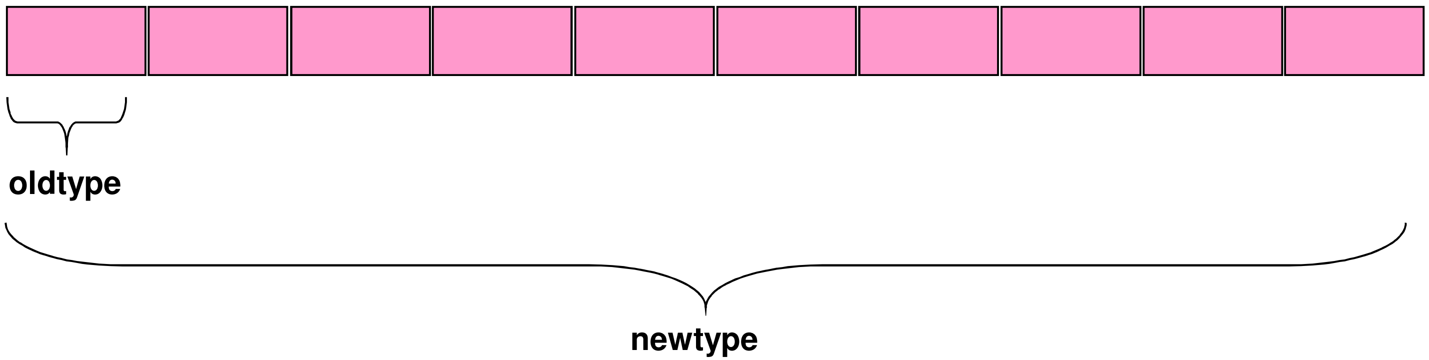
MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)

IN count: replication count (non-negative integer)

IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- MPI_TYPE_CONTIGUOUS 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_contiguous(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

a[4][4]

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

9.0	10.0	11.0	12.0

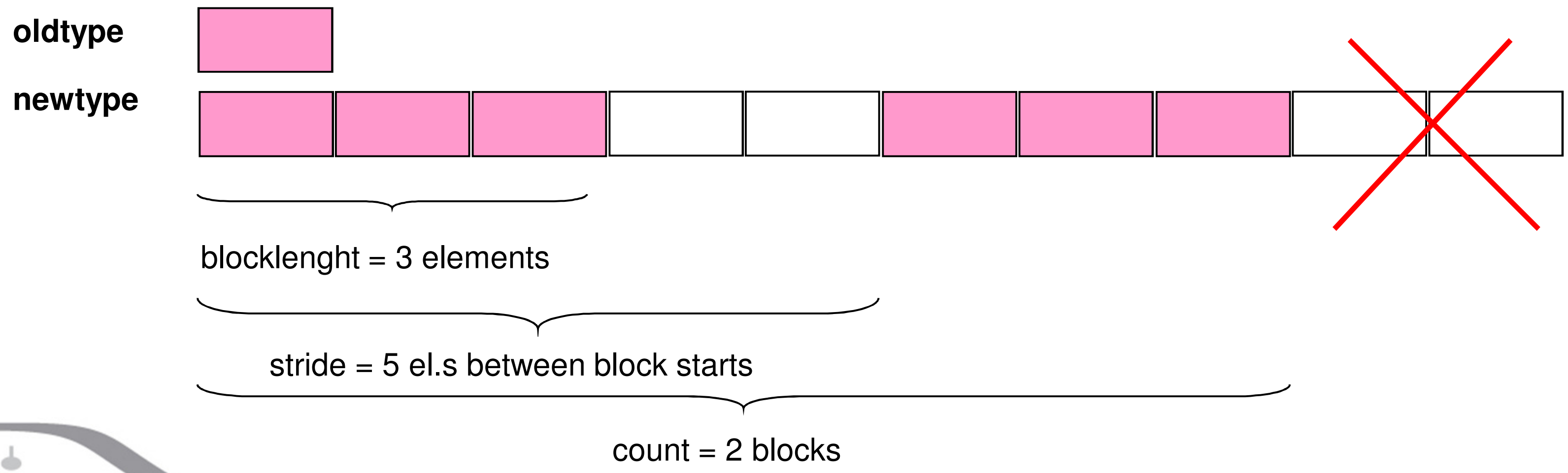
1 element of rowtype

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)

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



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

a[4][4]

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

2.0	6.0	10.0	14.0

1 element of columntype

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)

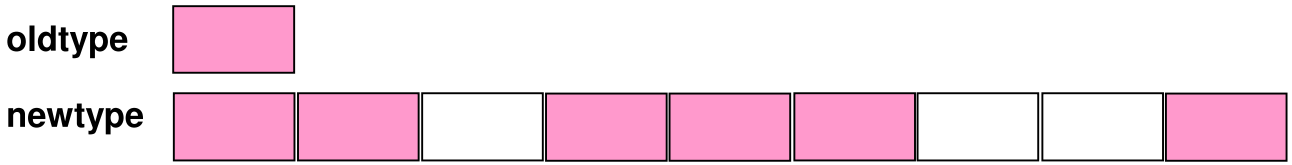
- Similar to MPI_TYPE_VECTOR, but stride is given in bytes, rather than in elements
- “H” stands for heterogeneous

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)

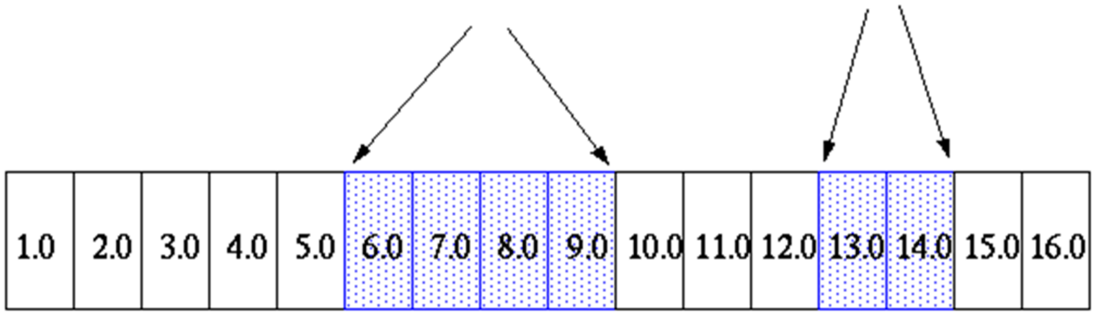
- Creates a new type from blocks comprising identical elements
- The size and displacements of the blocks can vary



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

Example 1

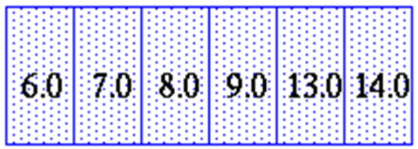
count = 2; blocklengths[0] = 4; blocklengths[1] = 2;
displacements[0] = 5; displacements[1] = 12;



a[16]

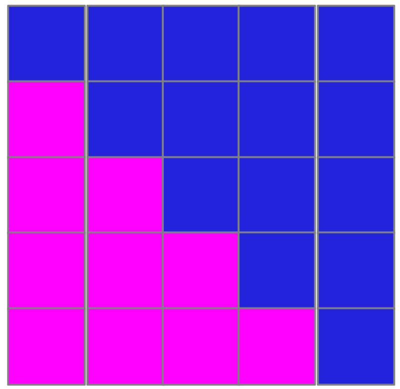
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

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



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 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.

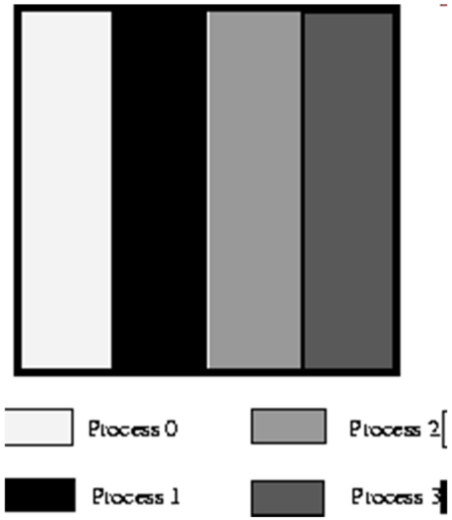
MPI_TYPE_CREATE_SUBARRAY

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)

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts,
            MPI_ORDER_C, MPI_DOUBLE, &filetype);

MPI_Type_commit(&filetype);
```

Summer School on PARALLEL COMPUTING

MPI_TYPE_CREATE_DARRAY

- Creates a data type corresponding to a distributed, multidimensional array
- N-dimensional distributed/strided sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran and C calls expect indices starting from 0

- An example is provided in the MPI2-I/O presentation.

CINECA

MPI_TYPE_CREATE_DARRAY

int **MPI_Type_create_darray** (int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN size: size of process group (positive integer)

IN rank: rank in process group (non-negative integer)

IN ndims: number of array dimensions as well as process grid dimensions(positive integer)

IN array_of_gsizes: number of elements of type oldtype in each dimension of global array (array of positive integers)

IN array_of_distribs: distribution of array in each dimension (array of state: **MPI_DISTRIBUTE_BLOCK** - Block distribution, **MPI_DISTRIBUTE_CYCLIC** – Cyclic distribution, **MPI_DISTRIBUTE_NONE** - Dimension not distributed.)

IN array_of_dargs: distribution argument in each dimension (array of positive integers, **MPI_DISTRIBUTE_DFLT_DARG** specifies a default distribution argument)

IN array_of_psizes: size of process grid in each dimension (array of positive integers)

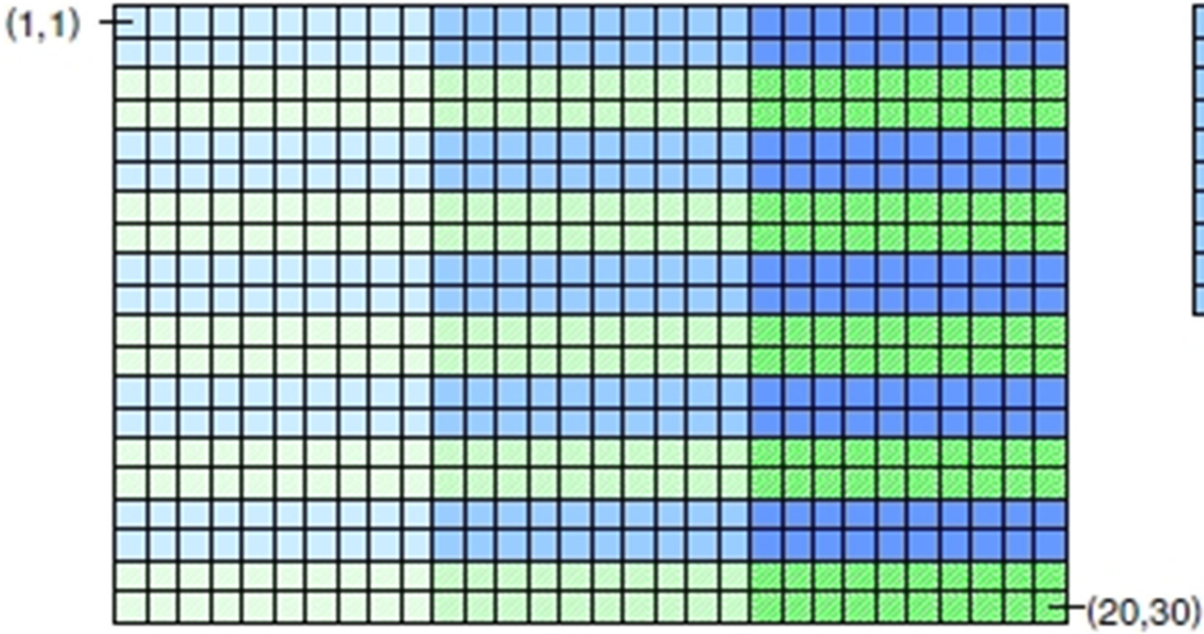
IN order: array storage order flag (state, i.e. **MPI_ORDER_C** or **MPI_ORDER_FORTRAN**)

IN oldtype: old datatype (handle)

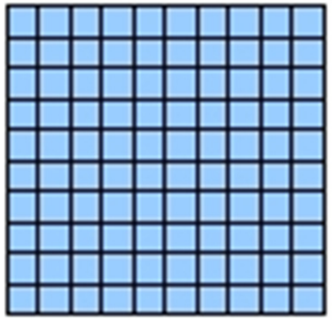
OUT newtype: new datatype (handle)

MPI_TYPE_CREATE_DARRAY

- Distribution scheme: (CYCLIC(2), BLOCK)
- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- distribution of global garray onto the larray in each of the 2x3 processes 
- garray on the file:



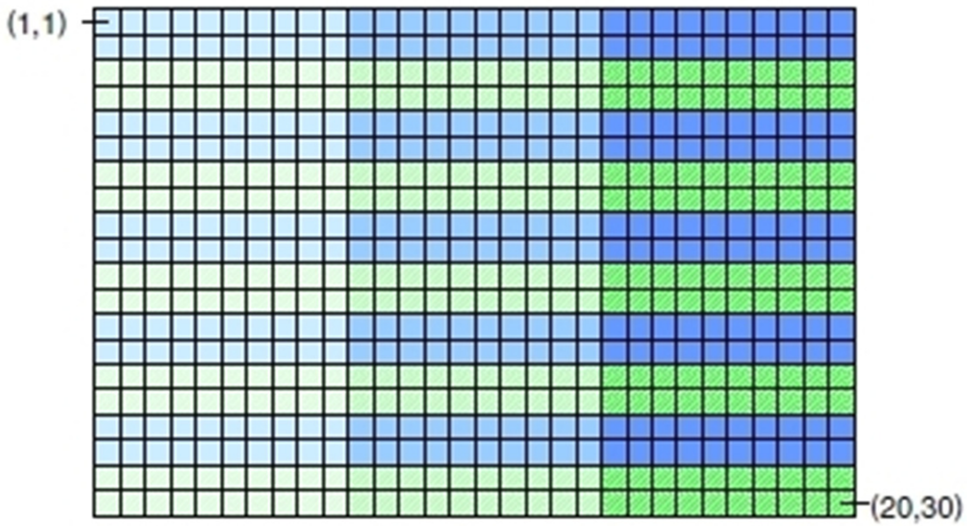
- e.g., larray on process (0,1):



MPI_TYPE_CREATE_DARRAY

int **MPI_Type_create_darray** (int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

array_of_gsizes[2] = (/20,30/)
array_of_distribs[2] = (/MPI_DISTRIBUTE_CYCLIC, MPI_DISTRIBUTE_BLOCK/)
array_of_dargs[2] = (/2, MPI_DISTRIBUTE_DFLT_DARG/)
array_of_psizes[2] = (/2,3/)



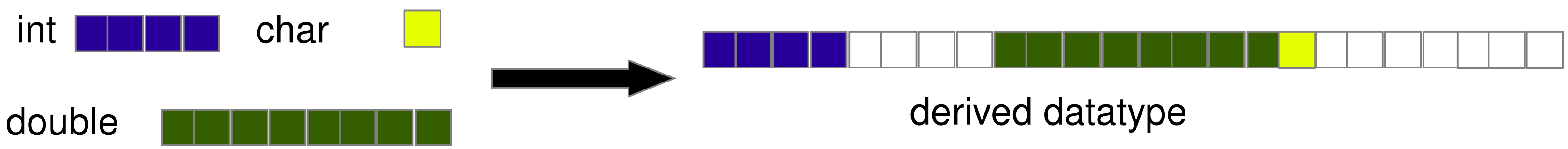
Size and Extent

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 **(including "holes")**

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



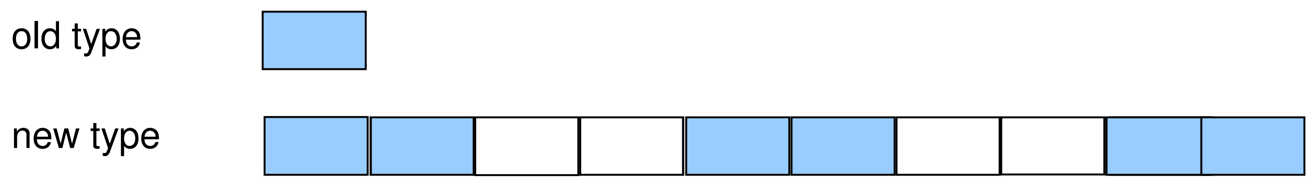
Size vs. extent of a datatype

Basic datatypes:

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

Derived datatypes:

- **extent include holes but...**
- **beware of the type vector: final holes are a figment of our imagination**



- 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)

Extent

- Extent controls how a datatype is used with the count field in a send and similar MPI operations
- Consider

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

- What actually gets sent?

```
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 in MPI_Recv) for count>1
  - Normally, this is right after the last byte used for (i-1)

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 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.

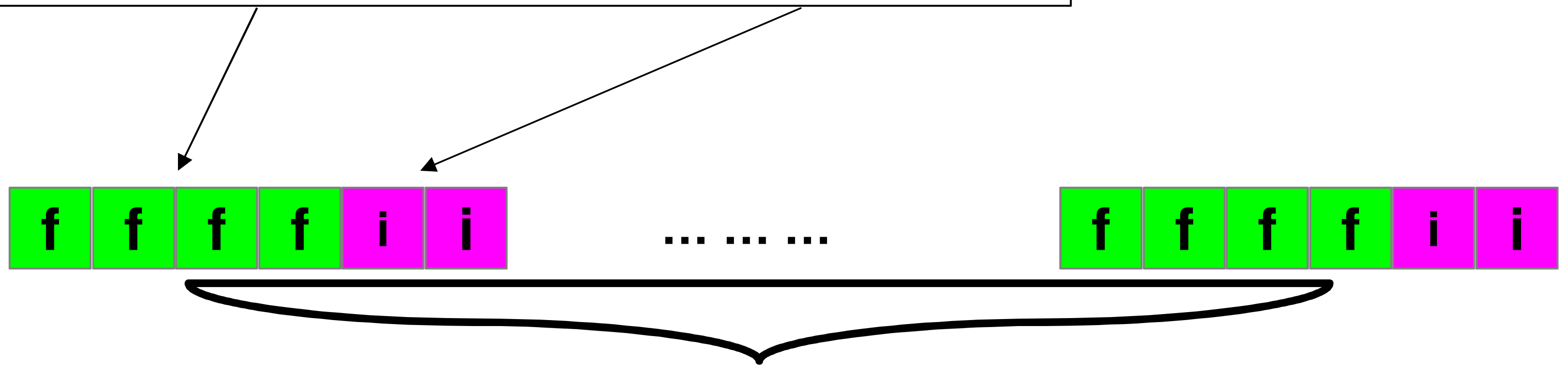
Using extents (not safe)

```
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;
```



particles[NELEM]

```
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!

- C struct may be automatically 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 dispacements
- Using this function helps wrt portability

Using displacements


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

- **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];

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);
```

Resizing datatypes

- Using addresses, is still unsafe for arrays of struct because of possible alignments of the last member of the structure

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

Example

```
/* 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 standard the memory layout of Fortran derived data is much more liberal
- 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)
```

Performances

- Performance depends on the datatype – more general datatypes are often slower
  - some MPI implementations can handle important special cases: e.g., constant stride, contiguous structures
- Overhead is potentially reduced by:
  - Sending one long message instead of many small messages
  - Avoiding the need to pack data in temporary buffers
- Some implementations are slow

QUESTIONS ???