21st Summer School of PARALLEL COMPUTING

July 2 - 13, 2012 (Italian)
September 3 - 14, 2012 (English)

Derived datatypes

Giusy Muscianisi – g.muscianisi@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, …)

Any data layout can be described with them

General datatypes allow to transfer efficently heterogeneous and non-contiguous data

Derived datatypes

Why to use them?

Problem:

1. one often wants to pass messages that contain values with different datatype (e.g., an integer count, followed by a sequence of real numbers)
2. one often wants to send noncontiguous data (e.g. a sub block of a matrix)

First solution:

1. → Consecutive MPI calls to send and receive each element in turn
   → Copy data to a single buffer before sending it

2. → Consecutive MPI calls to send and receive each element in turn
   → Use MPI_BYTE and sizeof() to avoid the type matching rules

- Additional latency costs due to multiple calls
- Additional latency costs due to memory copy
- Not portable to a heterogeneous system

Derived datatypes

Why to use them?

Problem:

1. one often wants to pass messages that contain values with different datatype (e.g., an integer count, followed by a sequence of real numbers)
2. one often wants to send noncontiguous data (e.g. a sub block of a matrix)

Second solution:

MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers.

During the communication, the datatype tells MPI system where to take the data when sending or where to put data when receiving.

– – — – — – — – — – –

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

– gives size of each elements

– tells MPI how to interpret the bits it sends and received

Displacement:

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

Typemap

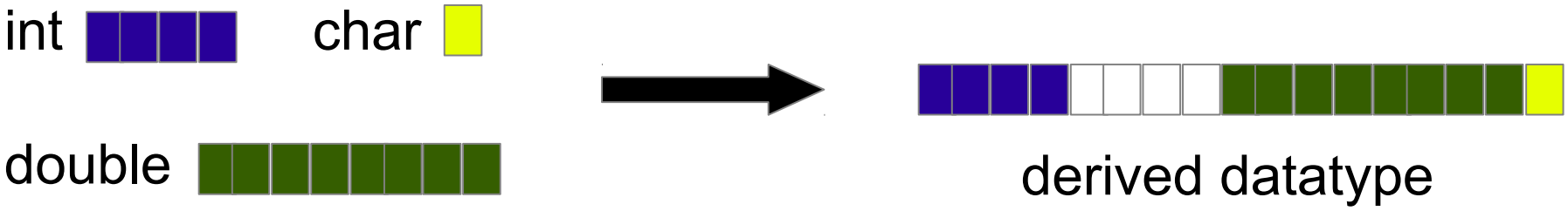
Example:

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 byte
• MPI_INDEXED	variably spaced datatype
• MPI_TYPE_CREATE_HINDEXED	like indexed, but the stride is specified in byte
• MPI_TYPE_CREATE_INDEXED_BLOCK	similar to mpi_type_create_hindex
• 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

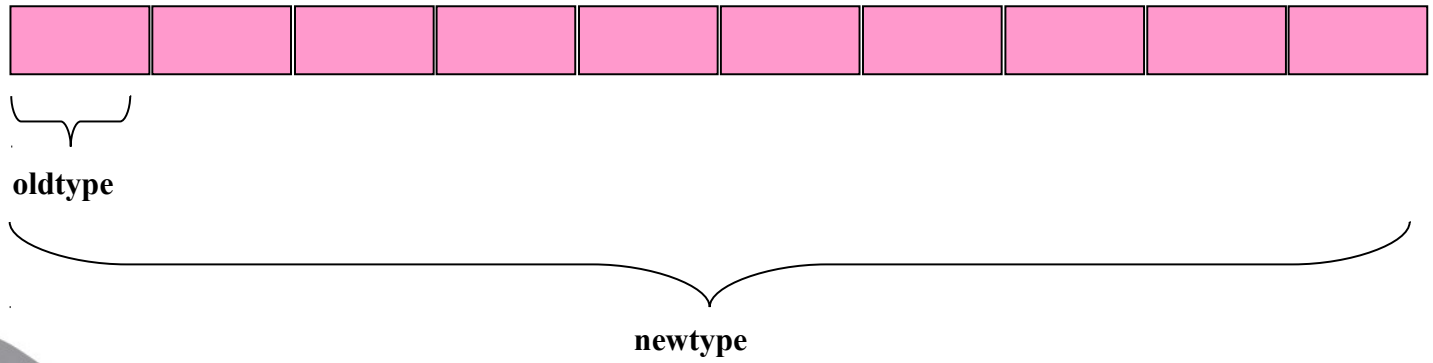
```
MPI_TYPE_FREE (datatype)
   INOUT datatype: datatype that is freed (handle)
```

- 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_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_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);
```

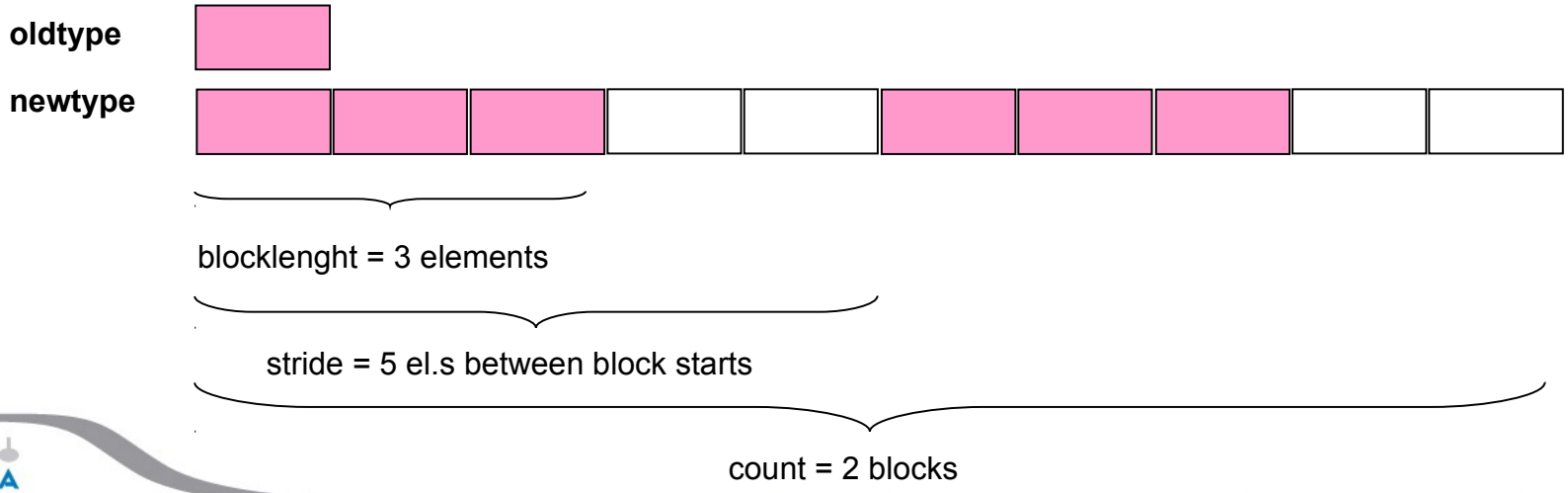
9.0	10.0	11.0	12.0

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

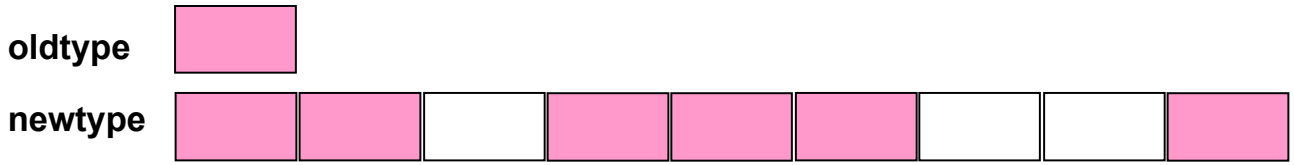
```
MPI_TYPE_CREATE_HVECTOR (count, blocklength, stride, oldtype, newtype)

   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)
```

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

MPI_TYPE_INDEXED

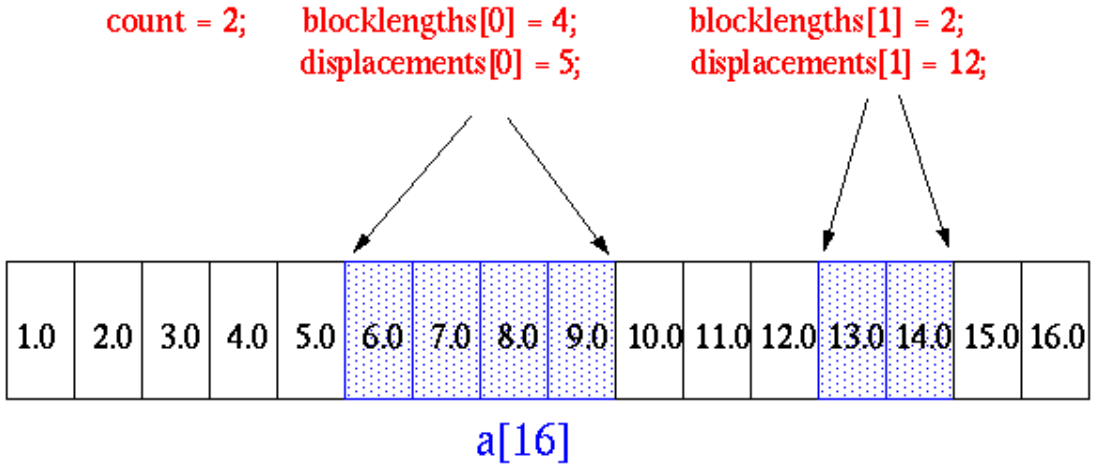
```
MPI_TYPE_INDEXED (count, array_of_blocklengths, array_of_displacements,
                  oldtype, newtype)
   IN count: number of blocks – also number of entries in
             array_of_blocklenghts and array_of_displacements
             (non-negative integer)
   IN array_of_blocklengths: number 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



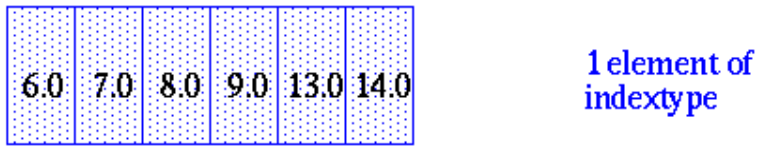
count=3, array_of_blocklenghths=(/2,3,1/), array_of_displacements=(/0,3,8/)

Example 1



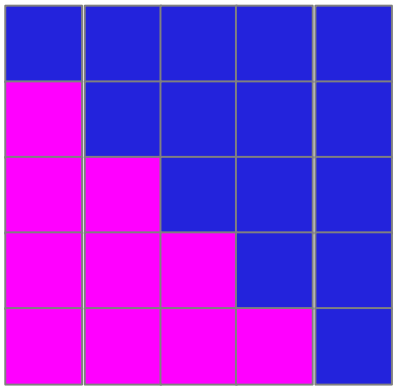

```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

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



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)
```

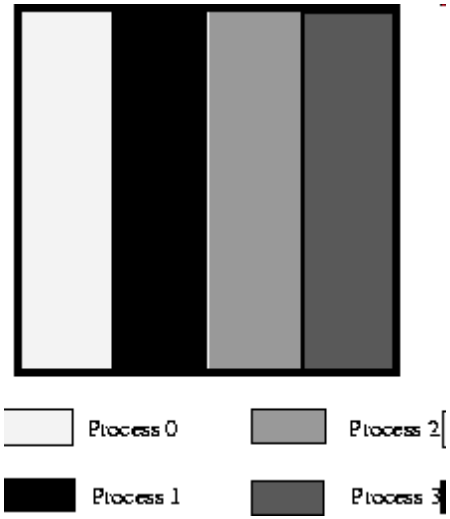
- Similar to MPI_TYPE_INDEXED, except that the block-length is the same for all blocks.
- There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). This function allows for constant blocksize and arbitrary displacements.

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)

```
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);
```

MPI_TYPE_CREATE_DARRAY

MPI_TYPE_CREATE_DARRAY (size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, 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)

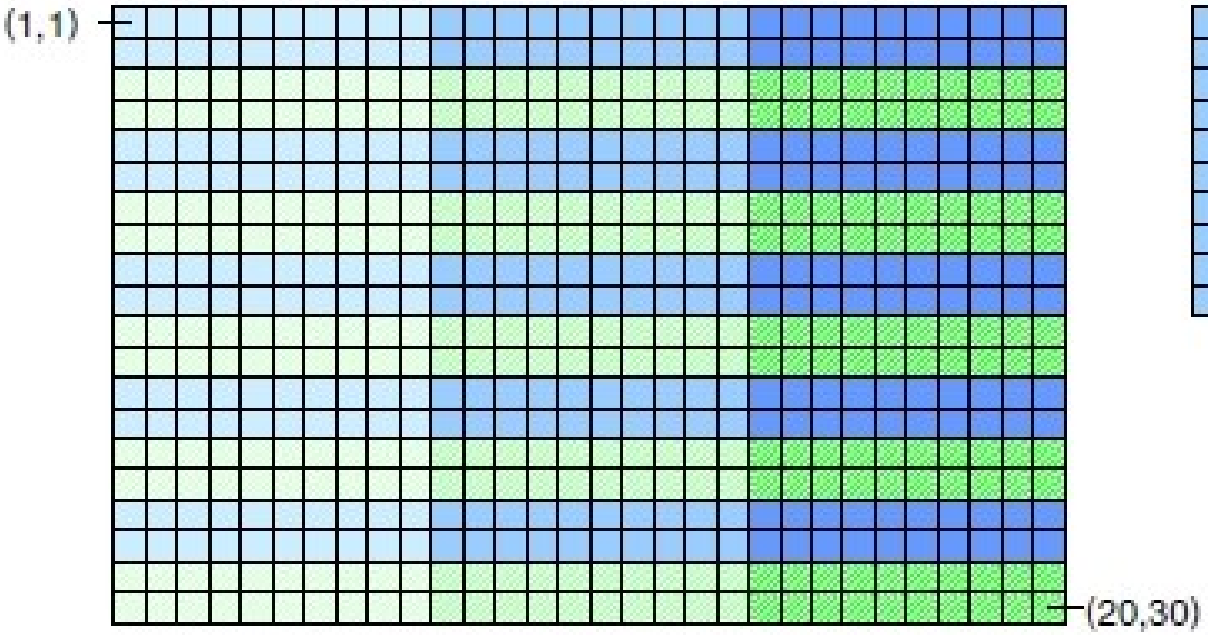
IN order: array storage order flag (state, i.e. MPI_ORDER_C or MPI_ORDER_FORTRAN)

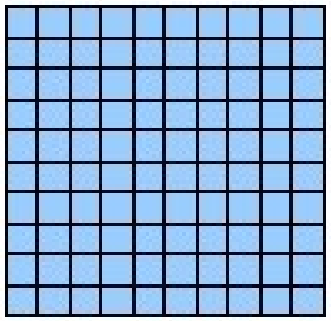
IN oldtype: old datatype (handle)

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):

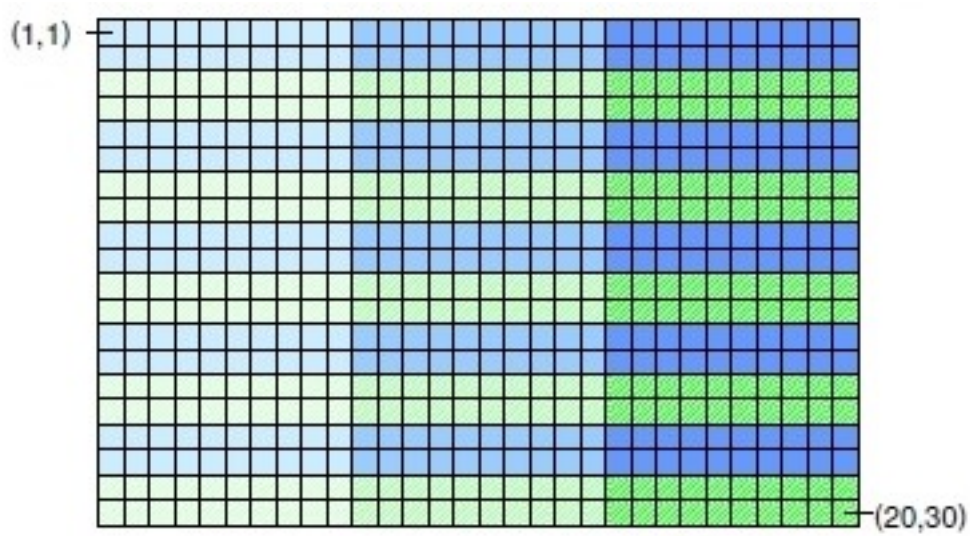




MPI_TYPE_CREATE_DARRAY

```
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_gsize[2] = (/20,30/)
array_of_distribs[2] = (/MPI_DISTRIBUTE_CICLIC, MPI_DISTRIBUTE_BLOCK/)
array_of_dargs[2] = (/2, MPI_DISTRIBUTE_DFLT_DARG/)
array_of_psize[2] = (/2,3/)
```



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)
```

- 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.

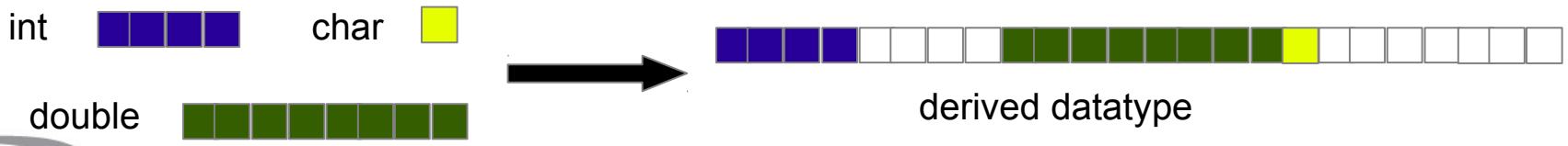
To understand the MPI_TYPE_CREATE_STRUCT … … …

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 inner "holes"). When creating new types, holes at the end of the new type are not countered to the extent

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

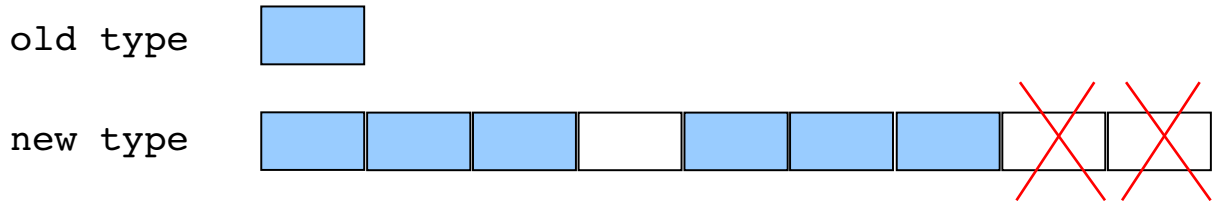


Size vs. extent of a datatype

Basic datatypes:

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

Derived datatypes (example)



- size = 6 x size of “old type”
- extent = 7 x extent of “old type”

Query size and extent of a datatype

```
MPI_TYPE_SIZE (datatype, size)
   IN datatype: datatype (handle)
   OUT size: datatype size (integer)
```

- Returns the total number of bytes 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)
```

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

MPI_TYPE_CREATE_STRUCT

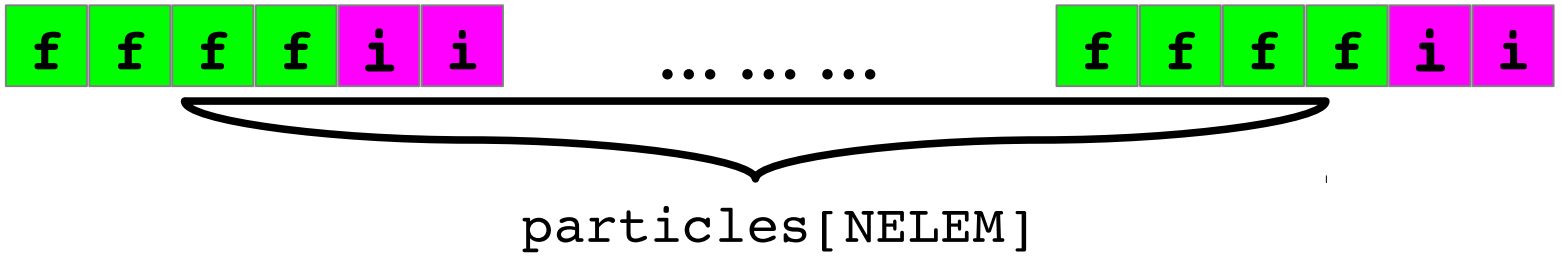
```
MPI_TYPE_CREATE_STRUCT (count, array_of_blocklengths,
            array_of_displacements, array_of_oldtypes, newtype )

   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.

Example 1/2

```
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);
```

Example 2/2

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

Particle particles[NELEM]
```

```
int count, blockcounts[0];
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_struct (count, blockcounts, displ,
oldtypes,  &particletype);

MPI_Type_commit(particletype);

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

MPI_Free(particletype);
```

Determining displacements

```
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 for determining relative dispacements
- The correct displacements can be determined
- Use this function guarantees portability

Ex - determining displacements

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

```
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 displacemnts 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,ParticleYype,dest,tag,comm);
MPI_Type_free (&ParticleType);
```

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”.
- Sets new lower and upper bound markers
- Allow for correct stride in creation of new derived datatypes:
  - Holes at the end of datatypes do not initially count to the extent
  - Successive datatypes (e.g. contiguous, vector) would not be defined as intended

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);
```

Performance

- Performance depends on the datatype – more general datatypes are often slower
- Overhead is potentially reduced by:
  - Sending one long message instead of many small messages
  - Avoiding the need to pack data in temporary buffers
- Some implementation are slow

QUESTIONS ???