HIGH PERFORMANCE
COMPUTING CINECA
ITALY

Parallel Programming with MPI

Part III - Derived datatypes

HPC SCHOOL
COSTA RICA

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

DERIVED DATATYPES
Definition, typemap, how to use

DERIVED DATATYPE CONSTRUCTORS
Contiguous, vector and other examples

MANAGING STRUCTS WITH CONSTRUCTORS
MPI_Type_create_struct, extent, using displacements

KABRÉ
SUPERCOMPUTADORA

Derived datatypes

Need for 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

Possible solution:

Make multiple MPI calls to send and receive each data element

→ If advantadgeous, copy data to a buffer before sending it

Additional latency costs due to multiple calls
Additional latency costs due to memory copy

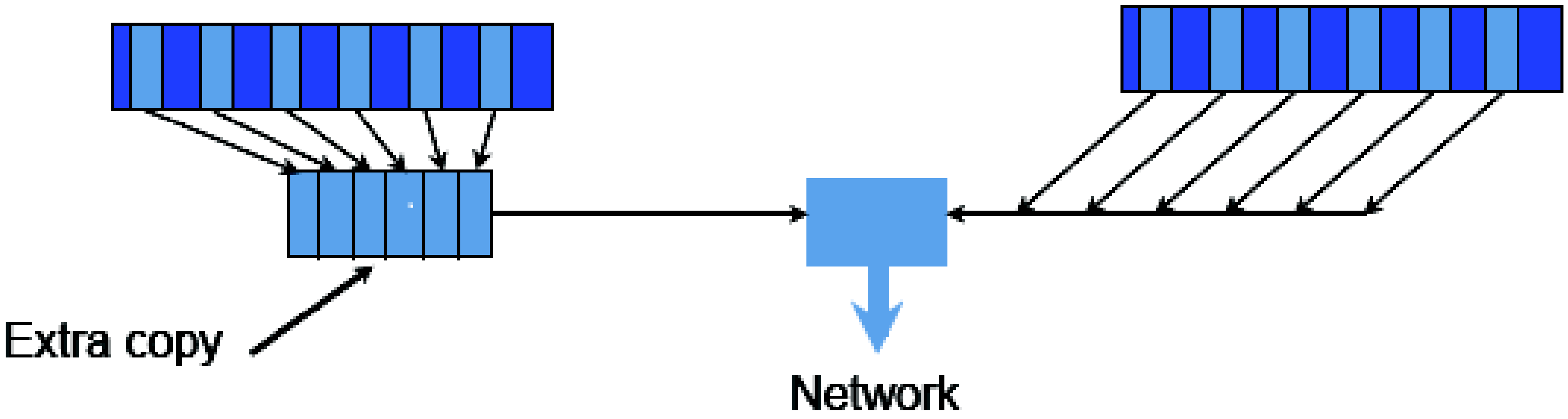
Datatype Solution

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.

The actual performances depend on the MPI implementation

Derived datatypes are also needed for getting the most out of MPI-I/O.



Definitions

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 bytes)

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

- list of types in the typemap
- gives size of each elements and tells MPI how to interpret the bits it sends and receives

Displacement:

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

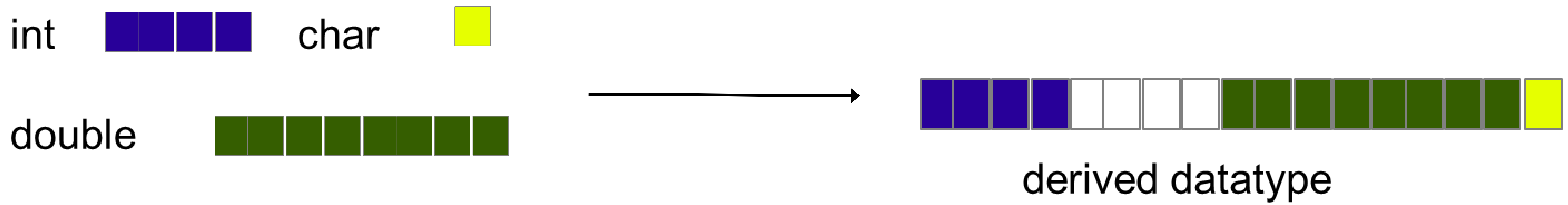
Typemap

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 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;

Committing and freeing

```
int MPI_Type_commit(MPI_Datatype *type);
```

- Before it can be used in a communication or I/O call, each derived datatype has to be committed
- New datatypes have to be declared as MPI_Datatype

```
int MPI_Type_free(MPI_Datatype *type);
```

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

Derived datatype constructors

MPI_Type_contiguous

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
          MPI__Datatype *newtype);
```

count number of consecutive elements of the old datatype
oldtype the previous datatype to be used as base
newtype the new derived datatype

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**.

MPI_Type_contiguous

```
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

```
int MPI_Type_vector(int count, int blocklength, int stride,
         MPI_Datatype oldtype, MPI__Datatype *newtype);
```

count number of blocks
blocklength number of elements in each block
Stride number of elements (NOT bytes) between the start of each block
oldtype the previous datatype to be used as base
newtype the new derived datatype

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

MPI_Type_vector

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_indexed

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
    int *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

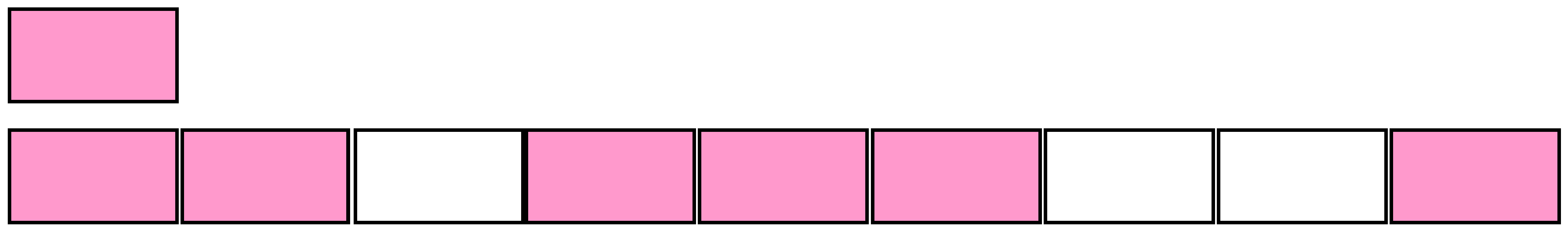
count	number of blocks
array_of_blocklengths	number of elements in each block
array_of_displacements	displacement for each block
oldtype	the previous datatype to be used as base
newtype	the new derived datatype

Creates a new type from blocks comprising identical elements

The size and displacements of the blocks can vary

oldtype

newtype



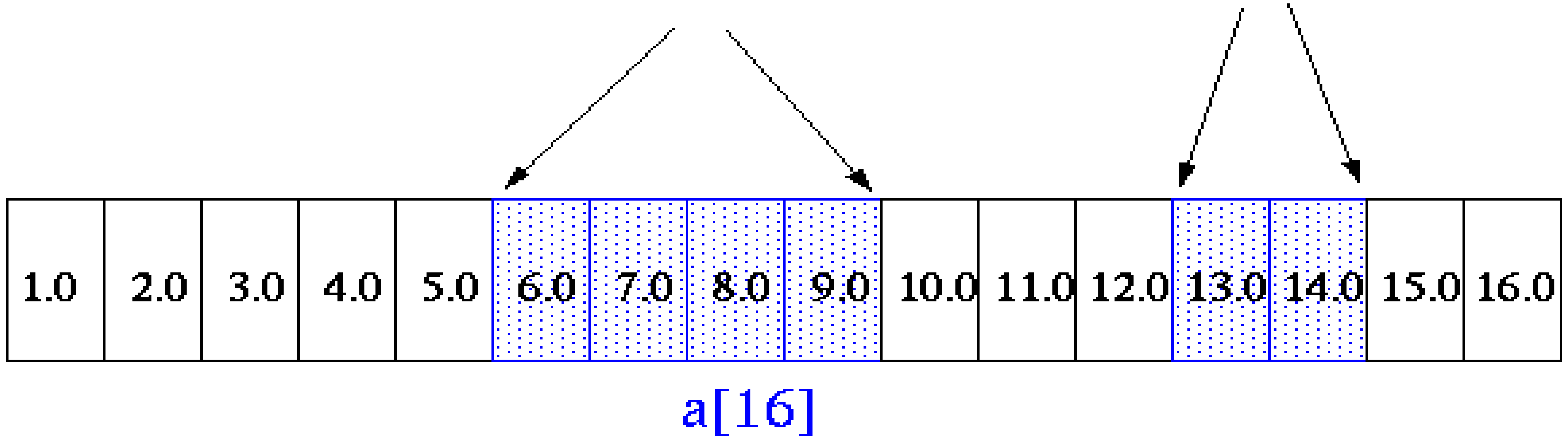
count=3, array_of_blocklenghths=(/2,3,1/), array_of_displacements=(/0,3,8/)

MPI_Type_indexed

count = 2;

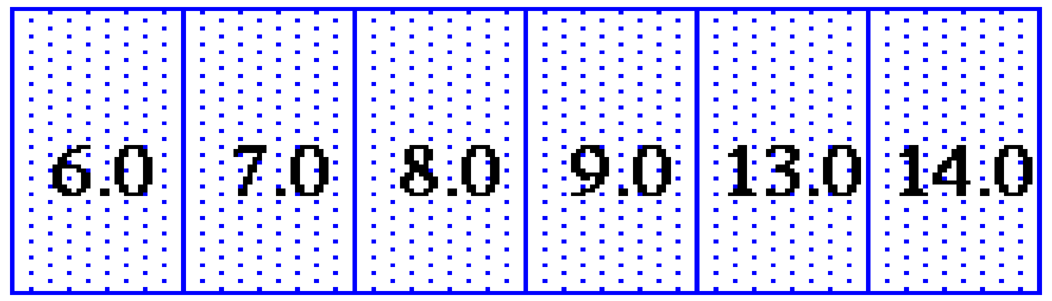
blocklengths[0] = 4;
displacements[0] = 5;

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




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

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



1 element of indextype

MPI_Type_indexed example

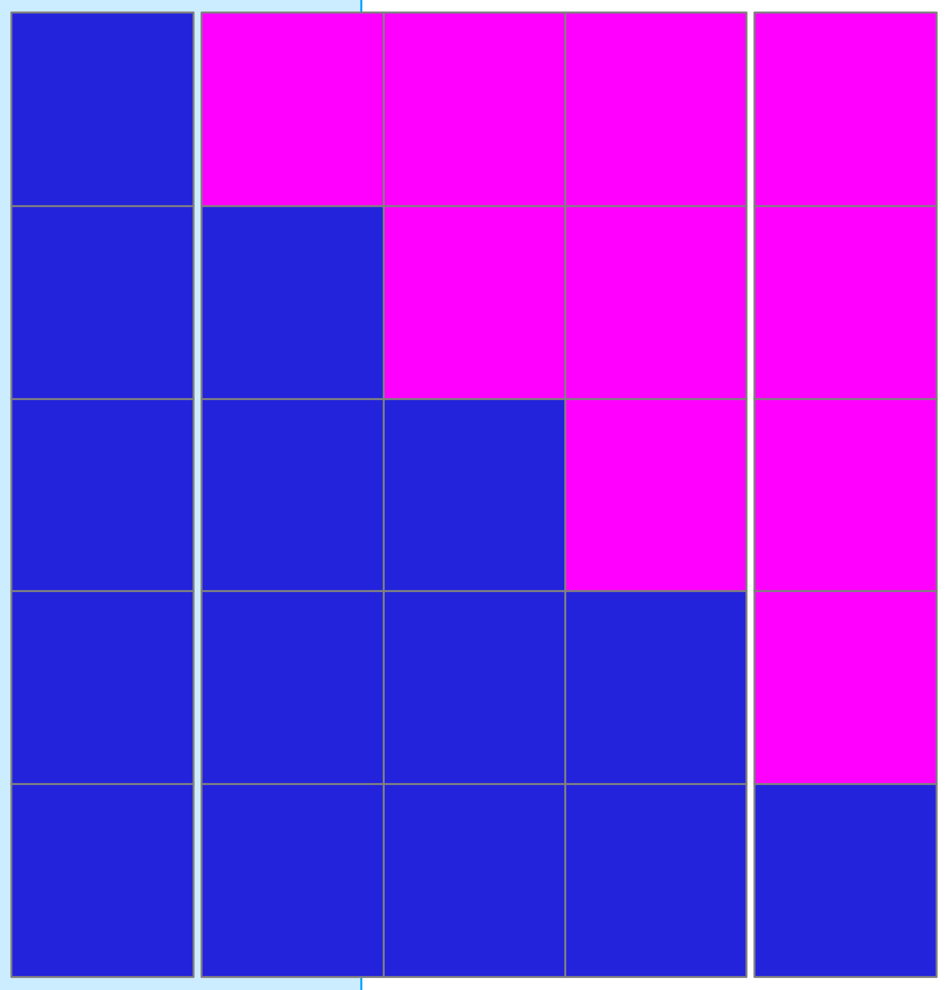
```
/* Lower Triangular Matrix (LTM) */
MPI_Datatype lower;
float *a;
int displ[100], blocklen[100];
a = malloc(100*100*sizeof(float));

/* Compute start and size of the rows */
for (int i=0, i < 100; i+) {
     displ[i]=100*(i-1);
     blocklen[i]=i;
     }

/* Create and commit a datatype for the LTM */
MPI_Type_indexed(100, blocklen, disp, MPI_FLOAT, &lower);
MPI_Type_commit(&lower);

/* Use it… */
MPI_Send(a, 1, lower, dest, tag, MPI_COMM_WORLD);

/* Free it after use */
MPI_Type_free(&lower);
```



MPI_Type_subarray

```
int MPI_Type_subarray(int ndims, int *array_of_sizes,
    int *array_of_subsizes, int *array_of_starts, int order,
    MPI_Datatype oldtype, MPI_Datatype *newtype);
```

ndims	number of array dimensions
array_of_sizes	number of elements of type “oldtype” in each dimension of the full array
array_of_subsizes	number of elements of type “oldtype” in each dimension of the subarray
array of starts	starting coordinates of the subarray in each dimension
order	array storage order flag (MPI_ORDER_C or MPI_ORDER_FORTRAN)
oldtype	the previous datatype to be used as base
newtype	the new derived datatype

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.

MPI_Type_subarray example

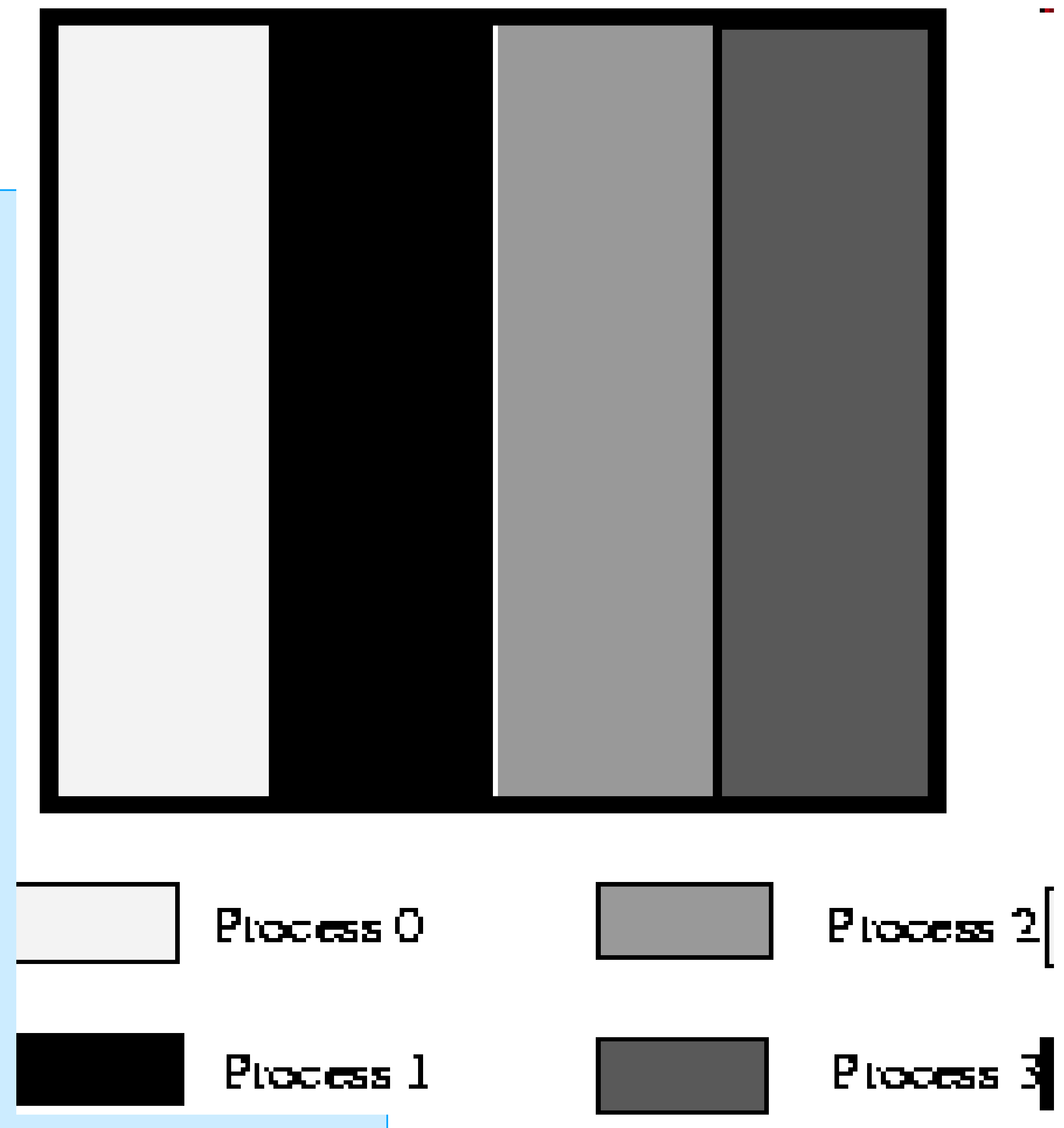
```
double matrix[100][100];
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

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_send(matrix,1,filetype,dest,tag,MPI_COMM_WORLD);
MPI_Type_free(&filetype);
```



Managing structs with constructors

MPI_Type_struct

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
    int *array_of_displacements, MPI_Datatype array_of_oldtypes,
    MPI_Datatype *newtype);
```

count	number of blocks
array_of_blocklengths	number of elements in each block
array_of_displacements	BYTE displacement for each block
oldtype	type of elements for each block
newtype	the new derived datatype

- 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_oldtypes*.
- 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 extent

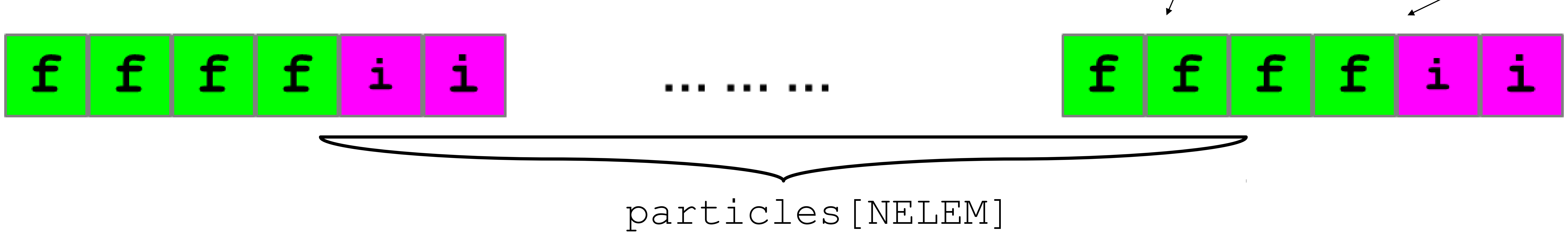
Extent:

The extent of a datatype is the span from the lower to the upper bound in bytes (including "holes")

```
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 extent

```
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_Type_free(&particletype);
```

Using 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 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);
```

Performance

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

Some implementations are slow.

Current MPI Standard to be found at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
Credits to A. Emerson and many other colleagues from CINECA for the slides