NTNU – Trondheim
Norwegian University of
Science and Technology

Derived data types

Jan.Christian.Meyer@ntnu.no

The primitive types

- All we've seen so far are messages containing some number of contiguous elements, of types like
  - MPI_INT64_T
  - MPI_DOUBLE
  - MPI_CHAR

    ...etc…
- It's fine for sending rows of consecutive array elements, but it quickly gets restrictive
  - What if you want to send columns?
  - What if you want to send the contents of a struct that has different types of elements inside?

Solution #1: DIY packing

- You can always marshal a serialized version of your own objects

```
struct { int i; int j; double v; } my_struct;              // Structured data
uint8_t my_buffer [ 2*sizeof(int)+sizeof(double) ];   // Just some bytes
*((int *)&my_buffer[0]) = my_struct.i;                     // Count bytes
*((int *)&my_buffer[sizeof(int)]) = my_struct.j;
*((double *)&my_buffer[2*sizeof(int)]) = my_struct.v;
```

  send/receive the contents of my_buffer, and manually un-marshal the whole mess at the receiving end
- This requires a lot of extra code, and it's kind of messy
- There are functions MPI_Pack and MPI_Unpack that dispense with most of the pointer arithmetic
  - They still create about as much additional work, though

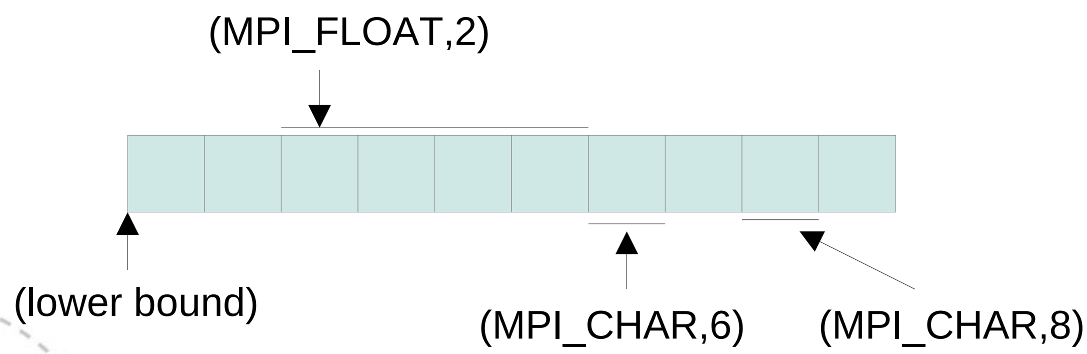
Derived types

- Most of the problems that can be solved by packing can also be handled more elegantly using derived types
- A derived type combines some other types (whether predefined or derived) in a structure which describes their layout in memory
- Derived types must be constructed and commited to MPI, so that it can “compile” an efficient representation of them
- After that, you can use them just like the primitive types

Types, in general

$\{ (t_0,d_0), (t_1,d_1), \ldots , (t_{n-1},d_{n-1}) \}$ ← this is an MPI_Datatype

- It consists of
  - a *type signature* $[t_0, t_1, \ldots, t_{n-1}]$ (i.e. a list of types), and
  - a list of *displacements* $[d_0, d_1, \ldots , d_{n-1}]$
- The displacements are all memory offsets relative to an arbitrary base address (called the *lower bound*)
- A type of one float and two chars may look like this



Note: there can be gaps between displacements

Memory locations and integers

- MPI doesn't demand much from its target platforms
  - It allows for the possibility that a memory address may not fit in an integer
  - Same as how we can have a 64-bit pointer to a 32-bit int
- Therefore, MPI has the MPI_Aint type, with the requirement that it can hold a memory address
- Displacements have this type
- It's mostly a general wrapper for points, so if you pass pointers where Aints are expected, no function call is likely to complain
  - The abstraction is more helpful in Fortran
  - Fortran's intrinsic pointers are ~~demented~~ unusual

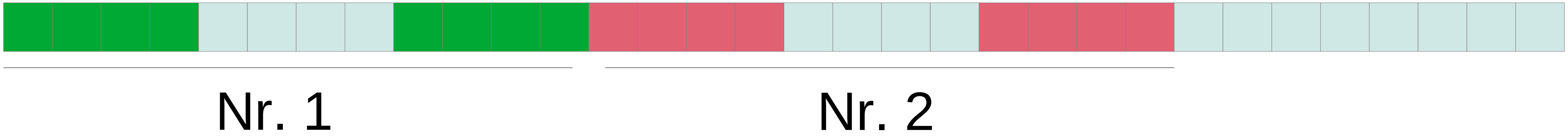
How big is a type?

- That depends on your point of view
- Our example type of 1 float and 2 chars can be said to consist of
  - 6 bytes (4 bytes of float data + 2 bytes of char data)

    or
  - 9 bytes (the distance from its origin to its end)
  - The last char is at displacement 8, so if we want to put two of these after one another, the 2nd begins at displacement 9 from the first
- This latter number is called the *extent* of the type
  - It includes gaps and spacing

Why care?

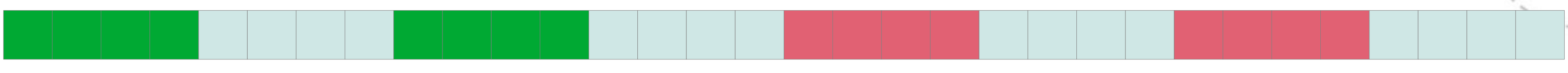
- The point of the distinction is that when MPI works out what "*count elements*" of an MPI_Datatype means (as seen in send, recv, and almost everywhere else), it uses the type's extent to read them
- Consider a type { (float, 0), (float, 8) }
- Two of these will have a memory footprint like this:



This is consistent with the idea of counting contiguous blocks of predefined types, Nr. 2 begins right after Nr.1

Tricks with the extent

- Since the elements of the example are separated by 4 bytes, an equally useful assumption might be that we want “2 consecutive elements” to look like this instead:



Nr. 1 Nr. 2

- This *can* be done by padding each element to include 4 floats and only use 2 of them, but it’s redundant
- Alternatively, we can set the extent of 1 element to be 16, instead of the default 12

Resizing types

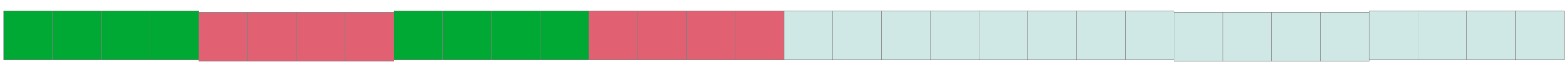
- int MPI_Type_create_resized (

```
    MPI_Datatype old_type,      // Type to start with
    MPI_Aint lower_bound,       // New value for lower bound
    MPI_Aint extent,            // New value for extent
    MIP_Datatype new_type       // Result comes out here
);
```

- With what we know already, we could at this point
  - Take a primitive type like MPI_INT64_T
  - Create a version of it that contains only every 8th consecutive int64_t in memory
  - Or something similar

Another use

- The extent is just the multiple to count “consecutive” copies in, padding it has no effect on memory contents
- If we adjust the extent to a shorter size than even the footprint of the data type, we can interleave data with it
- Here’s our example type again, in “2 consecutive elements” with extent 4:



Consecutive blocks & lengths

- So far, a type of consecutive float triplets becomes

  { (float,0), (float,4), (float,8) }
- That's a little redundant
- Since it's a consecutive block, it would suffice to count the number of consecutive elements

  { (float, 3 x sizeof(float)) }
- That's the notion of a *block length*, which comes up on the next few slides

Creating structured types

- In quite general terms, you can specify a type in all the gory details we've talked about, based on counts and block lengths of some other type(s):

  int MPI_Type_struct (

  ```
  int count,                              // How many parts do we have?
  int * array_of_blocklengths,            // What are their block lengths?
  MPI_Aint *array_of_displacements,       // What are their displacements?
  MPI_Datatype *array_of_types,           // What are their types?
  MPI_Datatype *new_type                  // Output: a brand new type
  ```

  );
- This gives explicit control of the whole type's layout

NTNU – Trondheim
Norwegian University of
Science and Technology

Committing types

- To make the translation from MPI types into the native addressing mechanism of your computer, we must *commit* them before use
- int MPI_Commit_type (MPI_Datatype *t);
  - This function does precisely that
- In practice,

```
MPI_Datatype my_type;
MPI_Type_struct ( foo, bar, baz, &my_type);
MPI_Commit_type ( &my_type );
MPI_Send ( ptr, 2, my_type, dst, tag, MPI_COMM_WORLD );
```

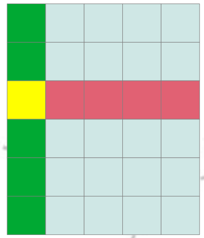
(Footnote: if you make intermediate types as steps to construct a really complicated one, it's only necessary to commit the final product)

Vector types

- The generality of structured types means they can also represent types which have a very regular layout
- Committing a structured type for lots of regularly spaced elements is repetitive and tedious
- Consider this layout:

- This would be a list of 11 displacements, even if we know they're all evenly separated
- This is a very common task
- Enter: vector types, consisting of
  - A count
  - A block length
  - A common *stride* between the blocks

Patterns in multidimensional arrays



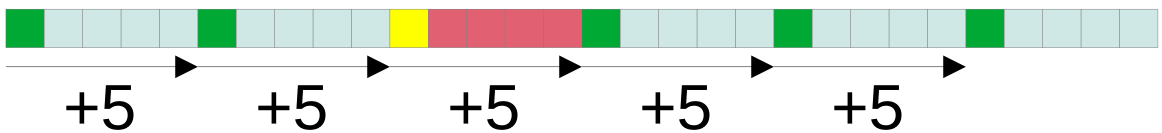
- Consider this 6x5 array, with a column and a row vector:
- In row-major order, it has the memory footprint (i,j)=i*5+j

- In column-major order, it's (i,j) = j*6+i

- If we do it in one way, the elements of columns are scattered out across memory
- If we do it in the other, the elements of rows are scattered instead

The common cause of stride

- Whether it’s this-major or that-major, indices along the minor axis are “strided”:



- That is the *stride* parameter of the vector type

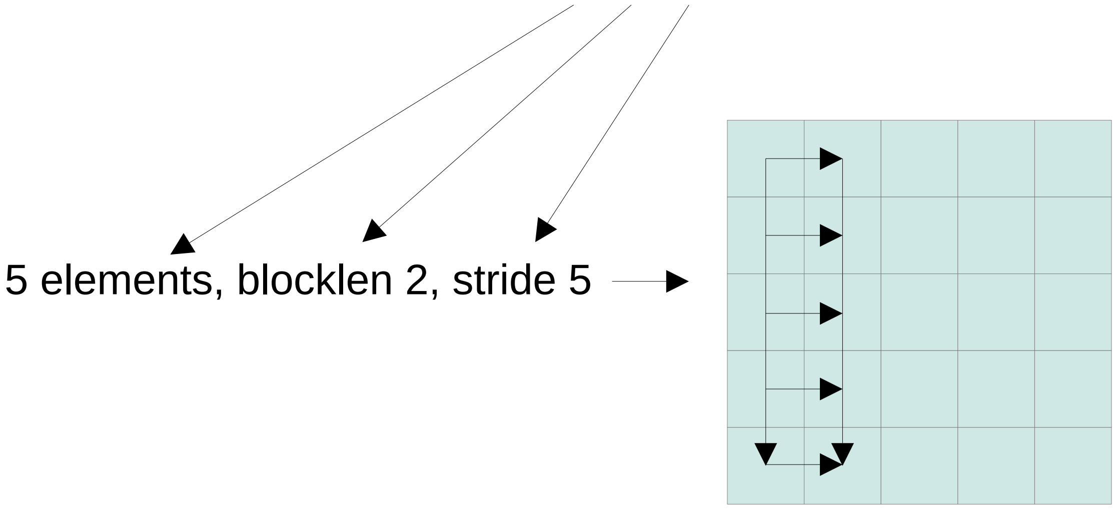
  (count and blocklength mean what they meant before)

- It’s the distance between neighbors in a direction we’ve chosen to project into sequential memory
- The scheme extends naturally to 3D, 4D, *etc.* by making successively larger jumps between neighboring elements along each new axis

Using vector types

- Given, *e.g*. a 5x5 matrix of doubles,

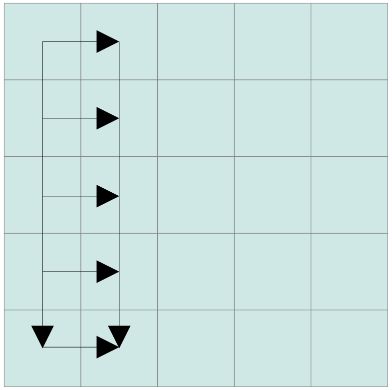
    MPI_Type_vector (5, 2, 5, MPI_DOUBLE, &my_type);



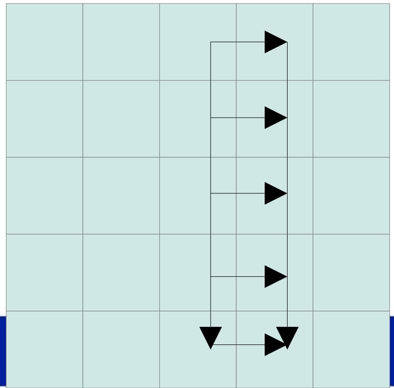
NTNU – Trondheim
Norwegian University of Science and Technology

It’s independent of position

- MPI_Send (&ARRAY(0,0), …, my_type, …); sends these elements



- MPI_Send (&ARRAY(0,2), …, my_type, …); sends these instead



With similar offsets, you don’t need an own type for every vector

Subarrays

- We can also construct types for internal regions of arrays

  int MPI_Type_create_subarray (

```
    int ndims,                          // How many dimensions in array?
    const int array_of_sizes[],         // How big is the entire array?
    const int array_of_subsizes[],      // How big is our slice of it?
    const int array_of_starts[]         // Where is the origin of the slice?
    int order, MPI_Datatype old_type, MPI_Datatype *new_type
);
```

A 2D example

- Using

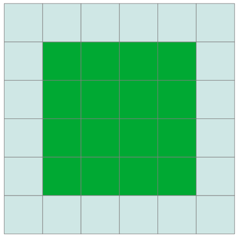
  ndims=2

  array_of_sizes = (int[2]) { 6, 6 }

  array_of_subsizes = (int[2]) { 4, 4 }

  array_of_starts = (int[2]) { 1, 1 }

  we get this slice of a 6x6 array:



- Nice for separating domain interiors from halos

There are a couple of more conveniences

- int MPI_Type_contiguous (
  int count, MPI_Datatype oldtype, MPI_Datatype *newtype
  );
  - This is just a block of memory
- int MPI_Type_indexed (

```
int count,                  // Nr. of parts
int *block_lengths,         // List of blocklengths for the parts
int *displacements,         // List of their displacements
MPI_Datatype oldtype,       // What kind of elements?
MPI_Datatype *newtype
);
```

  - This is like MPI_Type_struct, except that all the struct members have the same type