Communication Completion

- Single completion
  - completion of a send operation indicates that the sender is now free to update the locations in the send buffer
  - completion of a receive operation indicates that the receive buffer contains the received message

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
int MPI_Test( MPI_Request *request,
              int *flag, MPI_Status *status)
```

Communication Completion

- Multiple Completions (ANY)
  - A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations.

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )
int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```

Communication Completion

- Multiple Completions (SOME)
  - A call to MPI_WAITSOME or MPI_TESTSOME can be used to wait for the completion of at least one out of several operations.

```
int MPI_Waitsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
int MPI_Testsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
```

Communication Completion

- Multiple Completions (ALL)
  - A call to MPI_WAITALL or MPI_TESTALL can be used to wait for the completion of all operations.

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )
int MPI_Testall( int count, MPI_Request *array_of_requests,
                 int *flag, MPI_Status *array_of_statuses )
```

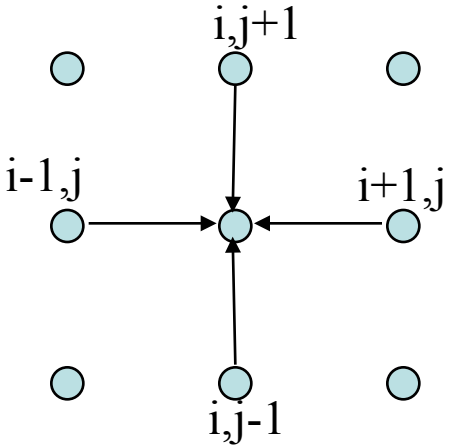
Persistent Communications

- A communication with the same argument list repeatedly executed within the inner loop of a parallel computation
  - Allow MPI implementations to optimize the data transfers
- All communication modes (buffered, synchronous and ready) can be applied

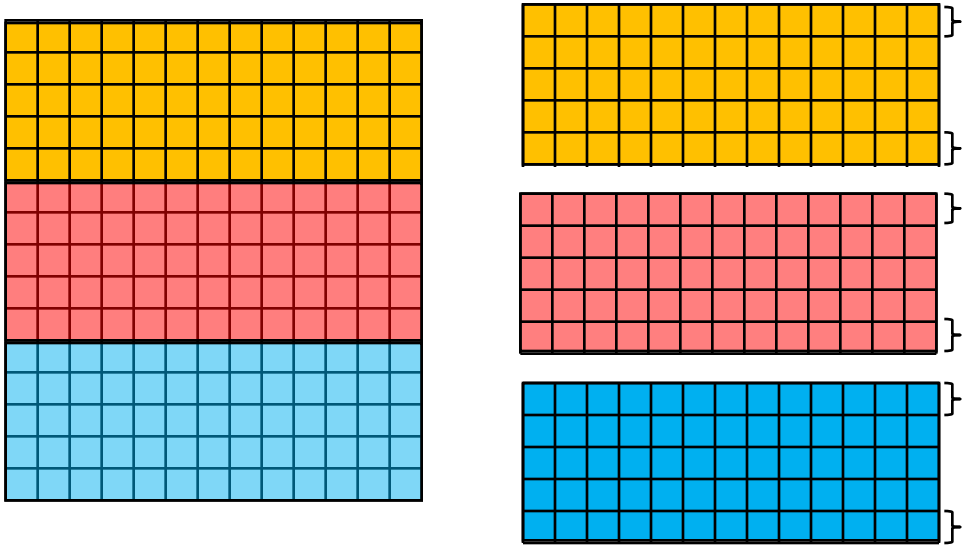
```
int MPI_[B,S, R,]Send_init( void* buf, int count, MPI_Datatype datatype,
                            int dest, int tag, MPI_Comm comm,
                            MPI_Request *request )
int MPI_Recv_init( void* buf, int count, MPI_Datatype datatype,
                            int source, int tag, MPI_Comm comm,
                            MPI_Request *request )
int MPI_Start( MPI_Request *request )
int MPI_Startall( int count, MPI_Request *array_of_requests )
```

$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$

Laplace's equation – MPI




```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                       +   U(i,j-1) + U(i,j+1))

  end for
end for
```



MPI Derived Datatypes

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

MPI Datatypes

- Abstract representation of underlying data
  - Handle type: MPI_Datatype
- Pre-defined handles for intrinsic types
  - E.g., C: MPI_INT, MPI_FLOAT, MPI_DOUBLE
  - E.g., Fortran: MPI_INTEGER, MPI_REAL
  - E.g., C++: MPI::BOOL
- User-defined datatypes
  - E.g., arbitrary / user-defined C structs

MPI Data Representation

- Multi platform interoperability
- Multi languages interoperability
  - Is MPI_INT the same as MPI_INTEGER?
  - How about MPI_INTEGER[1,2,4,8]?
- Handling datatypes in Fortran with MPI_SIZEOF and MPI_TYPE_MATCH_SIZE

Multi-Platform Interoperability

- Different data representations
  - Length 32 vs. 64 bits
  - Endianness conflict
- Problems
  - No standard about the data length in the programming languages (C/C++)
  - No standard floating point data representation
    - IEEE Standard 754 Floating Point Numbers
      - Subnormals, infinities, NANs …
    - Same representation but different lengths

How About Performance?

- Old way
  - Manually copy the data in a user pre-allocated buffer, or
  - Manually use MPI_PACK and MPI_UNPACK
- New way
  - Trust the [modern] MPI library
  - High performance MPI datatypes

MPI Datatypes

- MPI uses “datatypes” to:
  - Efficiently represent and transfer data
  - Minimize memory usage
- Even between heterogeneous systems
  - Used in most communication functions (MPI_SEND, MPI_RECV, etc.)
  - And file operations
- MPI contains a large number of pre-defined datatypes

Some of MPI's Pre-Defined Datatypes

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER*2
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

Datatype Matching

- Two requirements for correctness:
  - Type of each data in the send / recv buffer matches the corresponding type specified in the sending / receiving operation
  - Type specified by the sending operation has to match the type specified for receiving operation
- Issues:
  - Matching of type of the host language
  - Match of types at sender and receiver

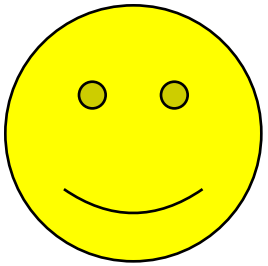
Datatype Conversion

- "Data sent = data received"
- 2 types of conversions:
  - Representation conversion: change the binary representation (e.g., hex floating point to IEEE floating point)
  - Type conversion: convert from different types (e.g., int to float)

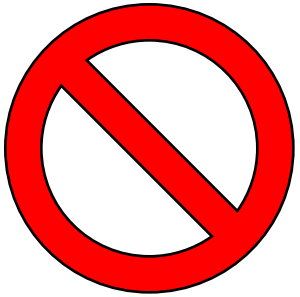
➔ Only representation conversion is allowed

Datatype Conversion

```
if( my_rank == root )
    MPI_Send( ai, 1, MPI_INT, … )
else
    MPI_Recv( ai, 1, MPI_INT, … )
```

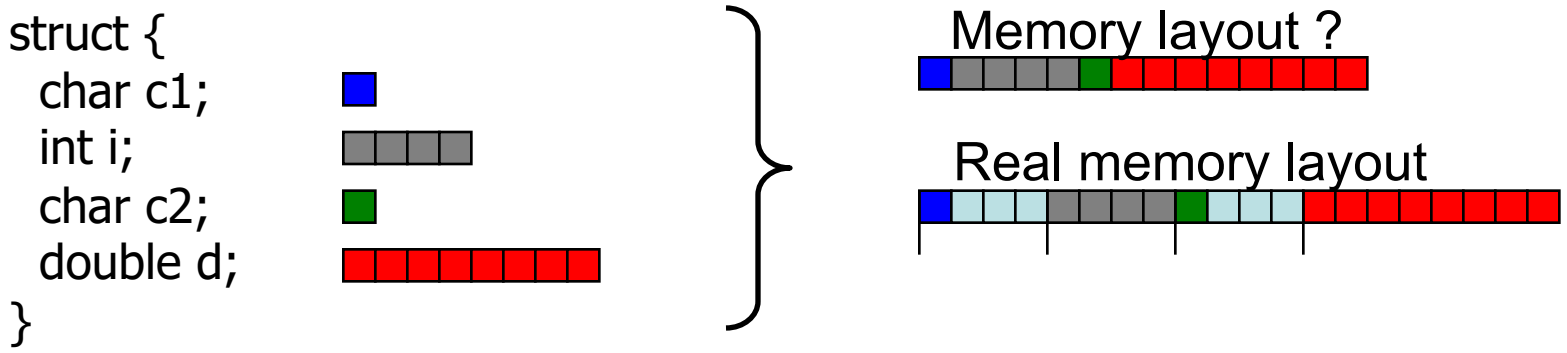


```
if( my_rank == root )
    MPI_Send( ai, 1, MPI_INT, … )
else
    MPI_Recv( af, 1, MPI_FLOAT, … )
```



Memory Layout

- How to describe a memory layout ?



Using iovecs (list of addresses)
<pointer to memory, length>
<baseaddr c1, 1>,<addr_of_i, 4>,
<addr_of_c2, 1>, <addr_of_d, 8>
•Waste of space
•Not portable …

Using displacements from base addr
<displacement, length>

<0, 1>, <4, 4>, <8, 1>, <12, 8>
•Sometimes more space efficient
•And nearly portable
•What are we missing ?

Datatype Specifications

- Type signature
  - Used for message matching

    { $type_0$, $type_1$, …, $type_n$ }
- Type map
  - Used for local operations

    { ($type_0$, $disp_0$), ($type_1$, $disp_1$),…, ($type_n$, $disp_n$) }

➔ It's all about the memory layout

User-Defined Datatypes

- Applications can define unique datatypes
  - Composition of other datatypes
  - MPI functions provided for common patterns
    - Contiguous
    - Vector
    - Indexed
    - …

➔ Always reduces to a type map of pre-defined datatypes

Handling datatypes

- MPI impose that all datatypes used in communications or file operations should be committed.
  - Allow MPI libraries to optimize the data representation

MPI_Type_commit(MPI_Datatype*)

MPI_Type_free(MPI_Datatype*)

- All datatypes used during intermediary steps, and never used to communicate does not need to be committed.

Contiguous Blocks

- Replication of the datatype into contiguous locations.

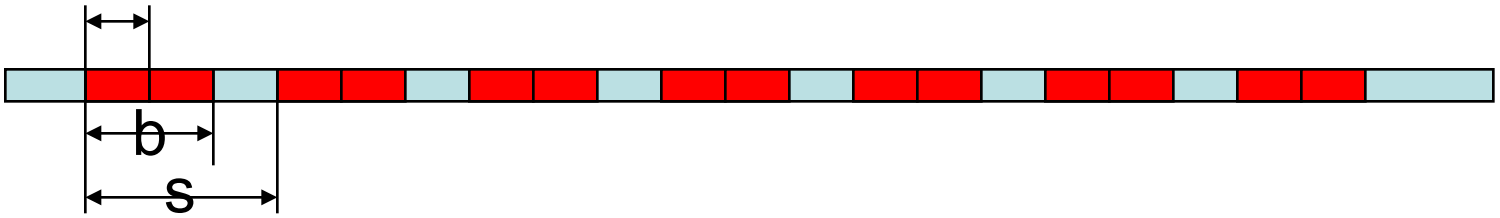
  MPI_Type_contiguous(3, oldtype, newtype)

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)		
IN	count	replication count(positive integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Vectors

- Replication of a datatype into locations that consist of equally spaced blocks

  MPI_Type_vector(7, 2, 3, oldtype, newtype)



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

IN	count	number of blocks (positive integer)
IN	blocklength	number of elements in each block (positive integer)
IN	stride	number of elements between start of each block (integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

- Replication of an old datatype into a sequence of blocks, where each block can contain a different number of copies and have a different displacement

MPI_TYPE_INDEXED(count, array_of_blocks, array_of_displs, oldtype, newtype)

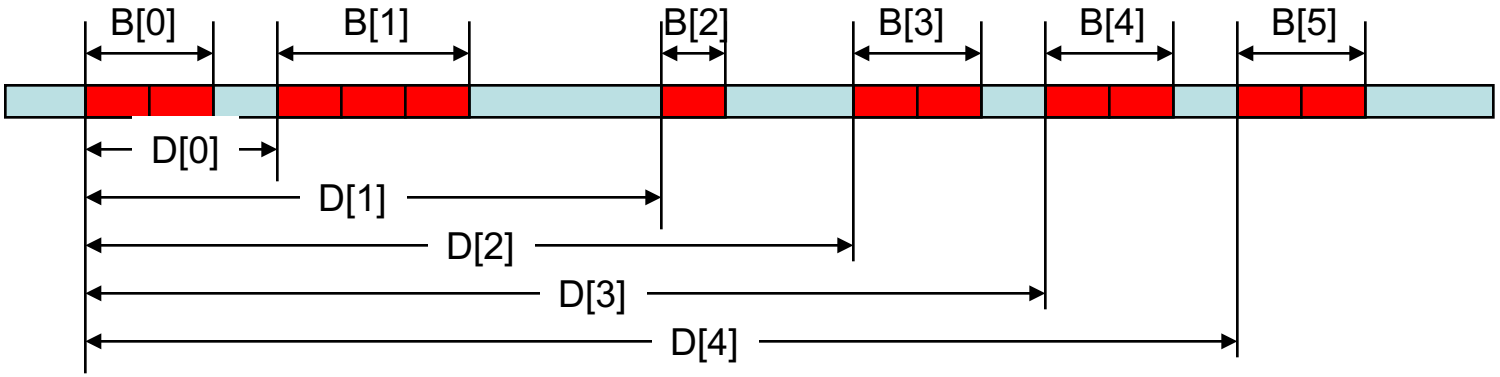
IN	count	number of blocks (positive integer)
IN	a_of_b	number of elements per block (array of positive integer)
IN	a_of_d	displacement of each block from the beginning in multiple multiple of oldtype (array of integers)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

array_of_blocklengths[] = { 2, 3, 1, 2, 2, 2 }

array_of_displs[] = { 0, 3, 10, 13, 16, 19 }

MPI_Type_indexed(6, array_of_blocklengths,
array_of_displs, oldtype, newtype)



Datatype Composition

- Each of the previous functions are the super set of the previous

  CONTIGUOUS < VECTOR < INDEXED

- Extend the description of the datatype by allowing more complex memory layout
  - Not all data structures fit in common patterns
  - Not all data structures can be described as compositions of others

“H” Functions

- Displacement is not in multiple of another datatype
- Instead, displacement is in bytes
  - MPI_TYPE_HVECTOR
  - MPI_TYPE_HINDEX
- Otherwise, similar to their non-”H” counterparts

Arbitrary Structures

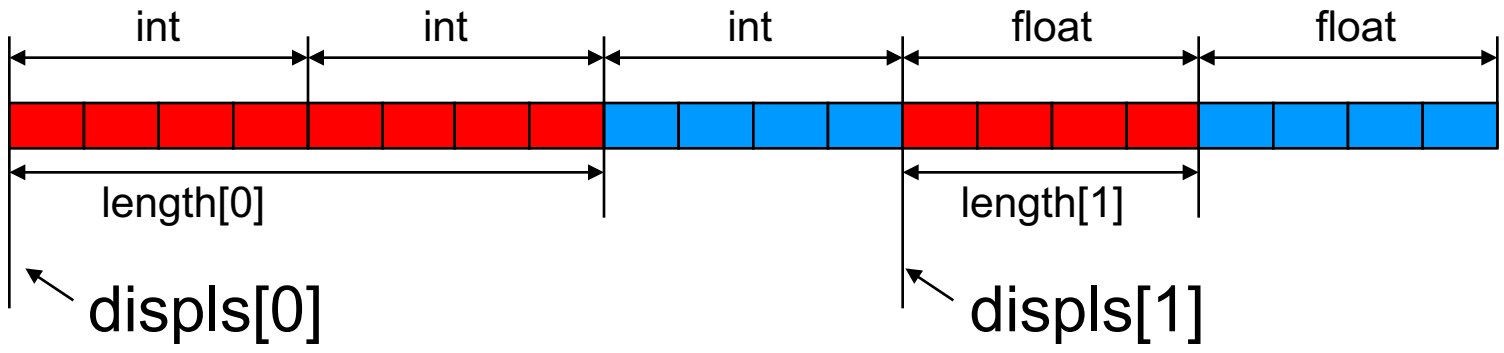
- The most general datatype constructor
- Allows each block to consist of replication of different datatypes

```
MPI_TYPE_CREATE_STRUCT( count, array_of_blocklength,
                        array_of_displs, array_of_types, newtype )
IN     count       number of entries in each array ( positive integer)
IN     a_of_b      number of elements in each block (array of integers)
IN     a_of_d      byte displacement in each block (array of Aint)
IN     a_of_t      type of elements in each block (array of MPI_Datatype handle)
OUT newtype        new datatype (MPI_Datatype handle)
```

Arbitrary Structures

```
struct {
  int i[3];
  float f[2];
} array[100];
```

```
Array_of_lengths[] = { 2, 1 };
Array_of_displs[] = { 0, 3*sizeof(int) };
Array_of_types[] = { MPI_INT, MPI_FLOAT };
MPI_Type_struct( 2, array_of_lengths,
  array_of_displs, array_of_types, newtype );
```



Portable Vs. non portable

- The portability refer to the architecture boundaries
- Non portable datatype constructors:
  - All constructors using byte displacements
  - All constructors with H<type>, MPI_Type_struct
- Limitations for non portable datatypes
  - One sided operations
  - Parallel I/O operations

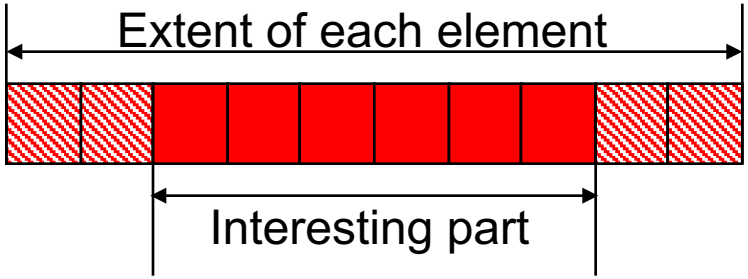
MPI_GET_ADDRESS

- Allow all languages to compute displacements
  - Necessary in Fortran
  - *Usually* unnecessary in C (e.g., “&foo”)

```
MPI_GET_ADDRESS( location, address )
IN      location   location in the caller memory (choice)
OUT     address    address of location (address integer)
```

And Now the Dark Side…

- Sometimes more complex memory layout have to be expressed



ddt

```
Struct {
  int gap[2];
  int i[6];
  int gap2[2];
}
```

`MPI_Send( buf, 3, ddt, … )`

What we just did …

And what we expected to do…

Lower-Bound and Upper-Bound Markers

- Define datatypes with “holes” at the beginning or end
- 2 pseudo-types: MPI_LB and MPI_UB
  - Used with MPI_TYPE_STRUCT

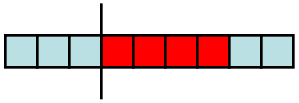
$$\text{Typemap} = \{ (\text{type}_0, \text{disp}_0), \ldots, (\text{type}_n, \text{disp}_n) \}$$

$$\text{lb(Typemap)} \begin{cases} \text{Min}_j \text{ disp}_j & \text{if no entry has type lb} \\ \text{min}_j(\{\text{disp}_j \text{ such that type}_j = \text{lb}) & \text{otherwise} \end{cases}$$

$$\text{ub(Typemap)} \begin{cases} \text{Max}_j \text{ disp}_j + \text{sizeof}(\text{type}_j) + \text{align} & \text{if no entry has type ub} \\ \text{Max}_j\{\text{disp}_j \text{ such that type}_j = \text{ub}\} & \text{otherwise} \end{cases}$$

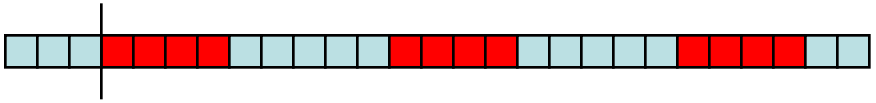
MPI_LB and MPI_UB

```
displs = ( -3, 0, 6 )
blocklengths = ( 1, 1, 1 )
types = ( MPI_LB, MPI_INT, MPI_UB )
MPI_Type_struct( 3, displs, blocklengths,
  types, type1 )
```



Typemap= { (lb, -3), (int, 0), (ub, 6) }

```
MPI_Type_contiguous( 3, type1, type2 )
```



Typemap= { (lb, -3), (int, 0), (int, 9), (int, 18), (ub, 24) }

MPI 2 Solution

- Problem with the way MPI-1 treats this problem: upper and lower bound can become messy, if you have derived datatype consisting of derived dataype consisting of derived datatype consisting of… and each of them has MPI_UB and MPI_LB set
- There is no way to erase LB and UB markers once they are set !!!
- MPI-2 solution: reset the extent of the datatype

```
MPI_Type_create_resized ( MPI_Datatype datatype,
MPI_Aint lb, MPI_Aint extent, MPI_Datatype*newtype );
```

- Erases all previous lb und ub markers

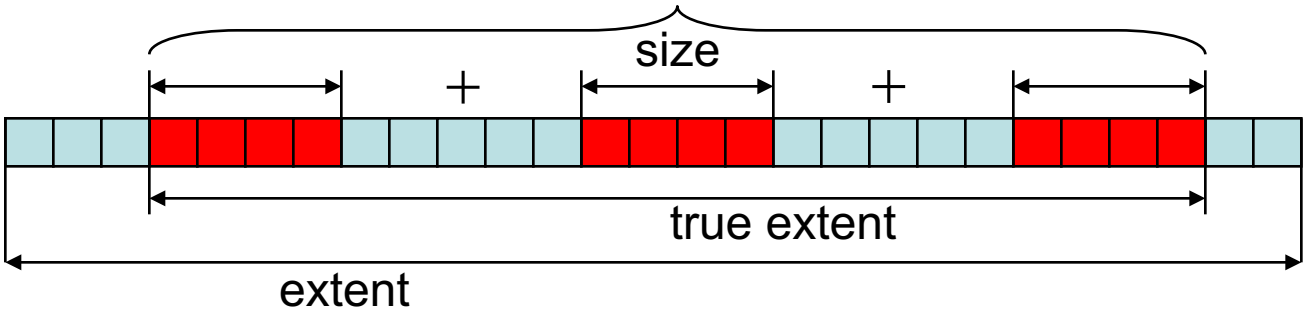
True Lower-Bound and True Upper-Bound Markers

- Define the real extent of the datatype: the amount of memory needed to copy the datatype inside
- TRUE_LB define the lower-bound ignoring all the MPI_LB markers.

$$Typemap = \{ (type_0, disp_0), \ldots, (type_n, disp_n) \}$$

$$true\_lb(Typemap) = \min_j \{ disp_j : type_j \neq lb \}$$
$$true\_ub(Typemap) = \max_j \{ disp_j + sizeof(type_j) : type_j \neq ub \}$$

Information About Datatypes



MPI_TYPE_GET_{TRUE_}EXTENT(datatype, {true_}lb, {true_}extent)

IN	datatype	the datatype	(MPI_Datatype handle)
OUT	{true_}lb	{true} lower-bound of datatype	(MPI_AINT)
OUT	{true_}extent	{true} extent of datatype	(MPI_AINT)

MPI_TYPE_SIZE(datatype, size)

IN	datatype	the datatype	(MPI_Datatype handle)
OUT	size	datatype size	(integer)

Decoding a datatype

- Sometimes is important to know how a datatype was created (eg. Libraries developers)
- Given a datatype can I determine how it was created ?
- Given a datatype can I determine what memory layout it describe ?

MPI_Type_get_enveloppe

MPI_Type_get_envelope (MPI_Datatype datatype,
int *num_integers, int *num_addresses,
int *num_datatypes, int *combiner);

- The combiner field returns how the datatype was created, e.g.
  - MPI_COMBINER_NAMED: basic datatype
  - MPI_COMBINER_CONTIGUOS: MPI_Type_contiguous
  - MPI_COMBINER_VECTOR: MPI_Type_vector
  - MPI_COMBINER_INDEXED: MPI_Type_indexed
  - MPI_COMBINER_STRUCT: MPI_Type_struct
- The other fields indicate how large the integer-array, the datatype-array, and the address-array has to be for the following call to MPI_Type_get_contents

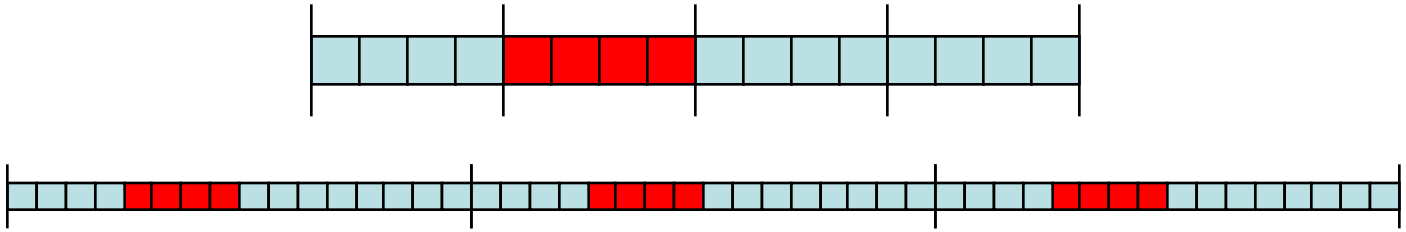
MPI_Type_get_contents

MPI_Type_get_contents (MPI_Datatype datatype,
int max_integer, int max_addresses, int max_datatypes,
int *integers, int *addresses, MPI_Datatype *dts);

- Call is erroneous for a predefined datatypes
- If returned data types are derived datatypes, then objects are duplicates of the original derived datatypes. User has to free them using MPI_Type_free
- The values in the integers, addresses and datatype arrays are depending on the original datatype constructor

One Data By Cache Line

- Imagine the following architecture:
  - Integer size is 4 bytes
  - Cache line is 16 bytes
- We want to create a datatype containing the second integer from each cache line, repeated three times



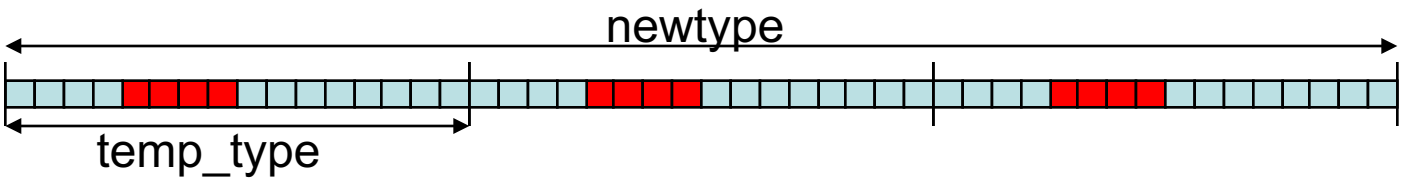
- How many ways are there?

Solution 1

```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_INT, MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 0, 0, 0, 0 };
int array_of_lengths[] = { 1, 1, 1, 1 };
struct one_by_cacheline c[4];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1].int[1], &(array_of_displs[1]) );
MPI_Get_address( &c[2].int[1], &(array_of_displs[2]) );
MPI_Get_address( &c[3], &(array_of_displs[3]) );

for( i = 0; i < 4; i++ ) Array_of_displs[i] -= start;

MPI_Type_create_struct( 4, array_of_lengths,
                                array_of_displs, array_of_types, newtype )
```

Solution 2

```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 4, 16 };
int array_of_lengths[] = { 1, 1 };
struct one_by_cacheline c[2];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1], &(array_of_displs[1]) );

Array_of_displs[0] -= start;
Array_of_displs[1] -= start;
MPI_Type_create_struct( 2, array_of_lengths,
                    array_of_displs, array_of_types, temp_type )
MPI_Type_contiguous( 3, temp_type, newtype )
```



Exercise

- Goals:
  - Create a datatype describing a matrix diagonal
  - What's different between C and Fortran ?