ICHEC

Irish Centre for High-End Computing

PRACE Course: Intermediate MPI

9-11 November 2022

MPI Derived Datatypes

User Defined datatypes

- Basic types: Pre defined datatypes, i.e, `MPI_INT / MPI_INTEGER`, `MPI_FLOAT / MPI_REAL`
    - can only send a single block of contiguous data
- Derived types: User defined types, i.e, for struct in C/type in Fortran, vectors with gaps in it ex. subblock of a matrix
- Define new datatypes
    - Grouping data of different datatypes in terms of both basic types and other derived types
    - Grouping non contiguous datatype
    - Grouping larger messages, count is int in C

User Defined datatypes

- Send a single message that would have required multiple messages to send with basic datatypes
- Code is more compact and maintainable
- Needed for getting the most out of MPI I/O
- Allows optimizations by the MPI runtime
- Performance depends on the datatype

- User-defined datatypes can be used both in point-to-point communication and collective communication
- The datatype instructs where to take the data when sending or where to put data when receiving
- Non-contiguous data in sending process can be received as contiguous or vice versa

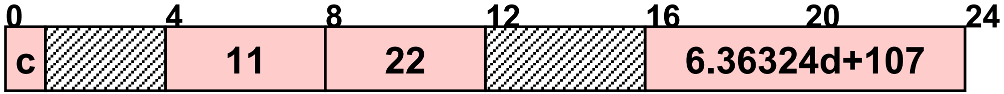
Dataype typemap

- Datatype stored by its type map:
  typemap = { ($type_0$,$disp_0$), ... , ($type_{n-1}$, $disp_{n-1}$) }
  - $type_i$: data types (typesig = { $type_0$, ... , $type_{n-1}$ })
  - $disp_i$: displacements in bytes
- Displacements are not required to be positive, distinct, or in increasing order.
- A derived datatype is logically a pointer to a list of entries: basic datatype as displacement.

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

Datatype typemap

Example:



derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g., structures, common blocks, module data subarrays, some variables in the memory

Describing Datatype Handle

- The type map, together with a base address buf, specifies a communication buffer
- They can be used in all send and receive operations.
  - `MPI_SEND(buf, 1, datatype,...)/MPI_RECV(buf, 1, datatype,...)`
- Have same status as predefined
  - can use in any message passing call
- Type matching rule isapplied:
  - type signature of sender and receiver has to match
  - the count argument has to match in Send/Recv operation
  - The message need not fill the whole receive buffer

Describing Datatype Handle

```
struct buff_layout
{int i_val[3];
double d_val[5];
}buffer;
```

Compiler

```
array_of_types[0] = MPI_INT;
array_of_blocklengths[0] = 3;
array_of_displacements[0] = 0;
array_of_types[1] = MPI_DOUBLE;
array_of_blocklengths[1] = 5;
array_of_displacements[1] = …;

MPI_Type_struct(2,
    array_of_blocklengths,
    array_of_displacements,
    array_of_types, &buff_datatype);

MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, …)
```

&buffer = the start address of the data

the datatype handle describes the data layout

MPI Derived datatypes

- Allows to create new contiguous data type consisting of an array of elements of another data type
- Vector data type consisting of regularly spaced blocks of elements of a component type or not regularly spaced data
- Specify an array of index locations for blocks of elements of a component type
- Struct data type to accommodate multiple data types.

- Datatype Constructors: `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_create_subarray`, `MPI_Type_create_struct`,

- `MPI_Type_dup()`: duplicate a datatype

MPI Derived datatypes

- Procedure to create and use a new Datatype
  1. Create your own datatype using Datatype Constructors
  2. Commit the new datatype.
  3. Use it in your communications routines
  4. Free your datatype. They take up memory.

- There is an overhead to defining a derived type. So, avoid:

```
do loop = 1, n
    define type
    use type
    free type
end do
```

Contiguous Datatype

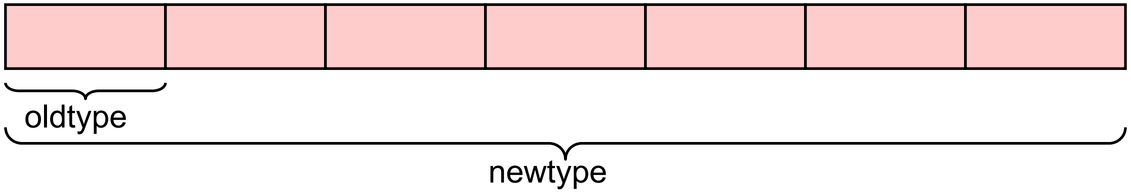
- C:

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

- Fortran:

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)
   TYPE(MPI_Type) :: oldtype, newtype
   INTEGER :: count; INTEGER, OPTIONAL :: ierror
```

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype
- Count elements of the same datatype forming a contiguous chunk in memory
- May also be useful intermediate stage in building more complicated types



Committing and Freeing a Datatype

- C:

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

- Fortran:

```
MPI_TYPE_COMMIT(datatype, ierror)
    TYPE(MPI_Type) :: datatype
    INTEGER, OPTIONAL :: ierror
```

- Before a datatype handle is used in message passing communication, it needs to be committed
- This must be done only once by each MPI process.
- If usage is over, one may call `MPI_TYPE_FREE()` to free a datatype and its internal resources.

Example – ring type contiguous

```
MPI_Datatype newtype;

ierror=MPI_Init(&argc,&argv);
ierror=MPI_Comm_size(MPI_COMM_WORLD,&uniSize);
ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

int sBuf[2]={myRank, 0};
int rBuf[2]={-1, 0};
int dest=(myRank < uniSize - 1) ? myRank + 1 : 0;
int src=(myRank > 0) ? myRank - 1 : uniSize - 1;

MPI_Type_contiguous(2, MPI_INT, &newtype);
MPI_Type_commit(&newtype);

while ( rBuf[0] != myRank ) {
  ierror=MPI_Sendrecv(sBuf, 1, newtype, dest, 100, rBuf, 1, newtype, src, 100,MPI_COMM_WORLD,
&status);
  sBuf[1] += rBuf[0];
  sBuf[0] = rBuf[0];
}

printf("Rank %d, sums %d\n", myRank, sBuf[1] );
MPI_Type_free(&newtype)
```

Vector Datatype

- C:

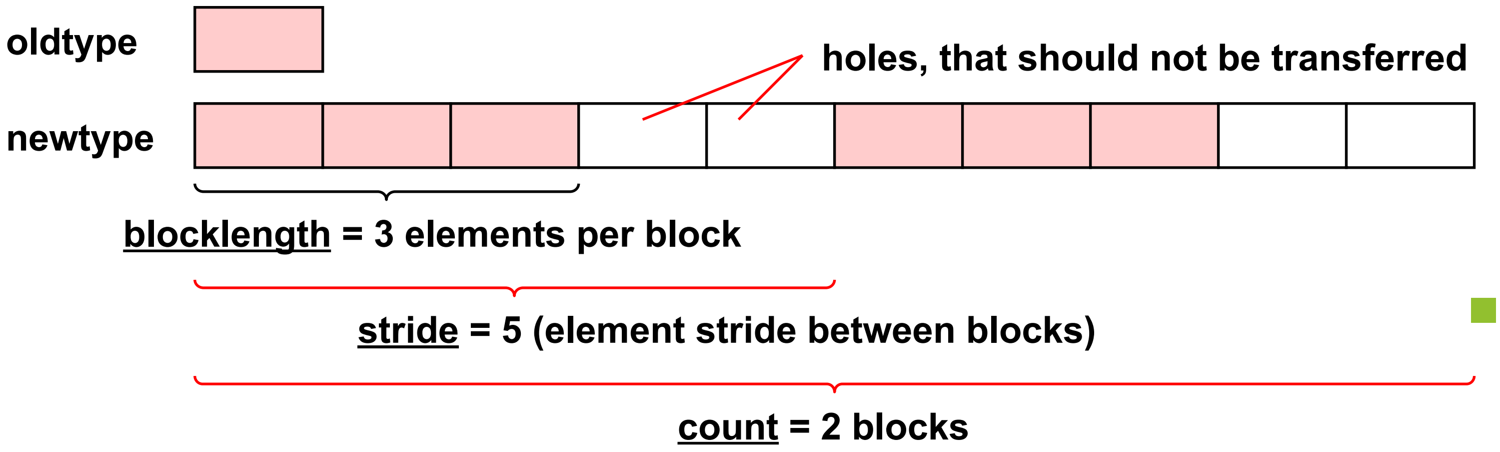
```
int MPI_Type_vector(int count,  int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_CONTIGUOUS(count, blocklength, stride, oldtype,
                    newtype, ierror)
    TYPE(MPI_Type) :: oldtype, newtype
    INTEGER :: count, blocklength, stride
    INTEGER, OPTIONAL :: ierror
```

- countblocks of blocklength elements of the same datatype
- Between the start of each block there are stride elements of the same datatype
- `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)` is equivalent to a call to `MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype)`

Vector Datatype



- Pattern with blocks and gaps: corresponds to a subsection of a 2D array
- Which row/column you are really sending depends on the pointer which you pass to the according MPI_Send routine.

Array Memory layout

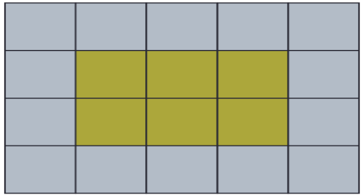
- Data is contiguous in memory
- You can choose to draw arrays however you like

First index i	Second index j	Format
right	up	coordinates
down	right	matrix
right	down	graphics

- Regardless of how you draw them, the layout in memory is
  - x[i][j] is followed by x[i][j+1] in C
  - x(i,j)is followed by x(i+1,j) in Fortran
- Depending on how you draw them, this can appear “row major” or “column major”

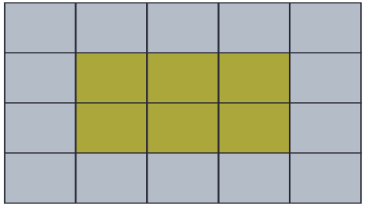
Array Memory layout

C: **`x[5][4]`**



- A 3 x 2 subsection of a 5 x 4 array

F: **`x(5,4)`**



- A 3 x 2 subsection of a 5 x 4 array

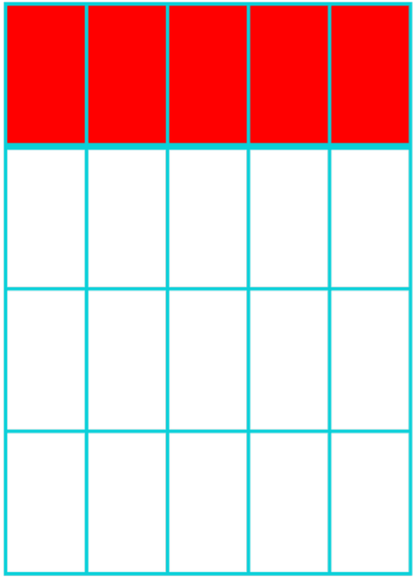
Sending a row using MPI_TYPE_vector

- C

```
MPI_Type_vector(1, 5, 1, MPI_INT, ARR_ROW)
```

- Fortran

```
MPI_Type_vector(5, 1, 4, MPI_INT, ARR_ROW)
```



```
MPI_Type_Commit(ARR_ROW)
MPI_Send(&buf …, ARR_ROW, …)
MPI_Recv(&buf …, ARR_ROW, …)
```

Sending a column using MPI_TYPE_vector

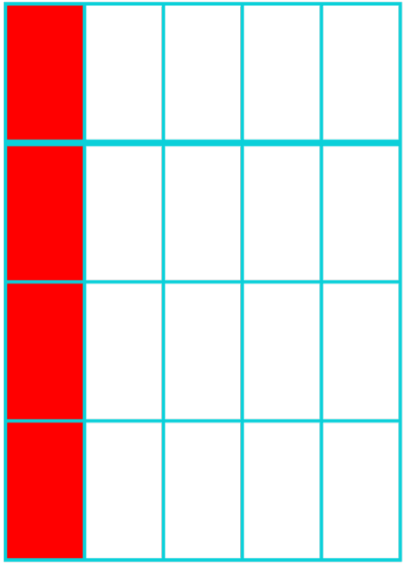
- C

```
MPI_Type_vector(4, 1, 5, MPI_INT, ARR_COL)
```

- Fortran

```
MPI_Type_vector(1, 4, 1, MPI_INT, ARR_COL)
```

```
MPI_Type_Commit(ARR_COL)
MPI_Send(&buf …, ARR_COL, …)
MPI_Recv(&buf …, ARR_COL, …)
```



Sending a sub-matrix using MPI_TYPE_vector

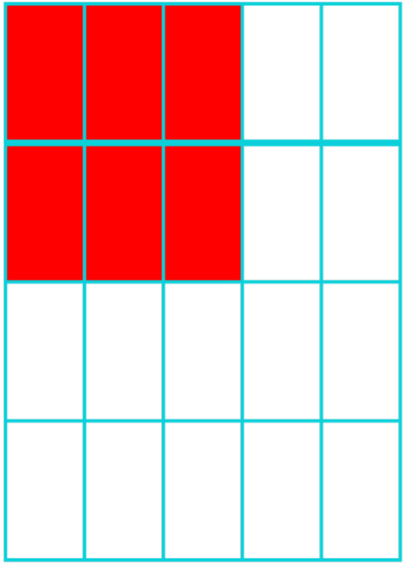
- C

```
MPI_Type_vector(2, 3, 5, MPI_INT, SUBMAT)
```

- Fortran

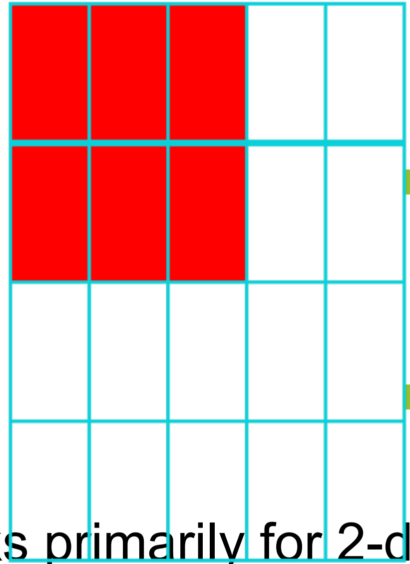
```
MPI_Type_vector(3, 2, 4, MPI_INT, SUBMAT)
```

```
MPI_Type_Commit(SUBMAT)
MPI_Send(&buf …, SUBMAT, …)
MPI_Recv(&buf …, SUBMAT, …)
```



MPI_Type_create_subarray

- Extracts a 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.
- All the rest are holes



ndims=2
sizes(0)=2; sizes(1)=3
subsizes(0)=4; subsizes(1)=5
starts(0)=0; starts(1)=0
order=MPI_ORDER_C

- Ideal for halo exchange in nd Cartesian
- Similar to MPI_Type_vector(), which works primarily for 2-dim arrays

Struct Datatype

- C:

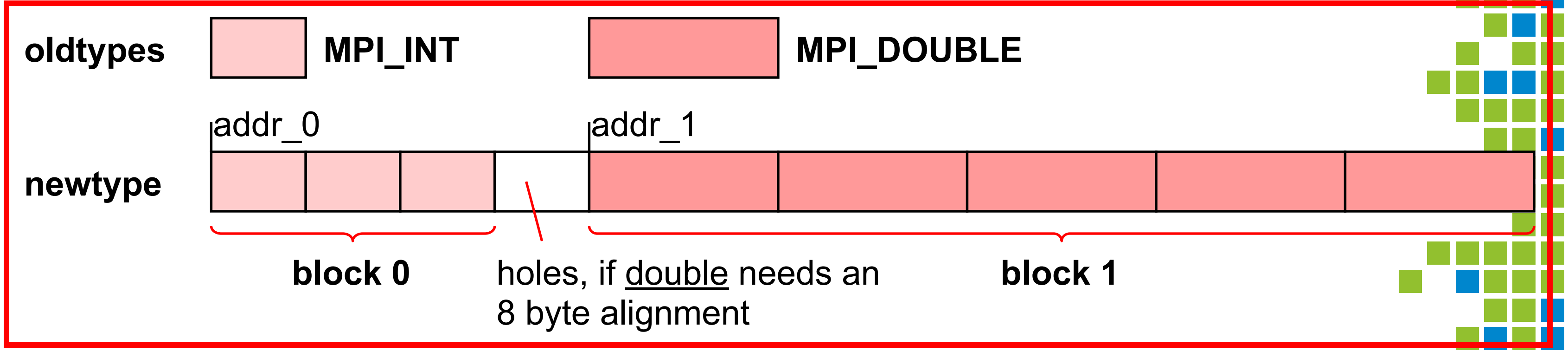
```
int MPI_Type_create_struct(int count,int *array_of_blocklngths,
MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths,
array_of_displacements, array_of_types, newtype, ierror)
    TYPE(MPI_Type) :: oldtype, newtype
    INTEGER :: count, blocklength, stride
    INTEGER, OPTIONAL :: ierror
```

- The most general derived datatype. It allows grouping of different datatypes.
- count: the number of elements in the derived type. (integer)
- array_of_blocklengths: the number of the entries in each element (array of integer)
- array_of_displacements : byte displacement of each element (array of address integers) (`INTEGER(KIND=MPI_ADDRESS_KIND)` in Fortran)
- array_of_types : type of elements in each block (array of handles to datatype objects)

Struct Datatype



count = 2
array_of_blocklengths = (3, 5)
array_of_displacements = (0, addr_1 – addr_0)
array_of_types = (MPI_INT, MPI_DOUBLE)

Memory Layout of Struct Datatypes

- C:

```
struct buf{
    int      i_val[3];
    double   d_val[5];
 }sBuf, rBuf
```

Fortran:

```
type buff
   sequence
   integer, dimension(3):: i_val
   real, dimension(5):: d_val
end type buff
```

- Storage format NOT defined by the language
  - different compilers do different things
  - e.g. insert arbitrary padding between successive elements
  - need to tell MPI the byte displacements of every element
- Explicitly compute memory addresses of every member
- Subtract addresses to get displacements from origin

How to compute the displacement

- C:

```
int MPI_Get_address(void* location, MPI_Aint *address)
```

- Fortran:

```
MPI_GET_ADDRESS(location, address, ierror)
        TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
        INTEGER(KIND=MPI_ADDRESS_KIND) :: address
        INTEGER, OPTIONAL :: ierror
```

- Returns the address of the memory location referenced by location
- `MPI_Aint` is an integer that is big enough to store an address
- Displacements are considered to be relative offsets: displacement[0] = 0 in most cases
- Displacements are not required to be positive, distinct or in increasing order
- *array_of_displacements[i] := address(block_i) –address(block_0)*

How to compute the displacement

- Absolute addresses: displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant `MPI_BOTTOM`. (`MPI_BOTTOM` is an address, i.e., cannot be assigned to a Fortran variable.)
- Relative displacement between two absolute addresses:

C: `MPI_Aint  MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)`

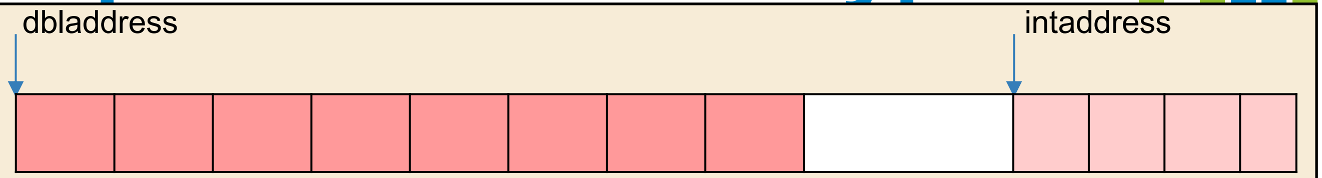
Fortran: `MPI_AINT_DIFF(addr1, addr2)`

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: addr1, addr2
```

- New absolute address as sum of an absolute base address and a relative displacement: `MPI_AINT_ADD()`.

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node82.htm

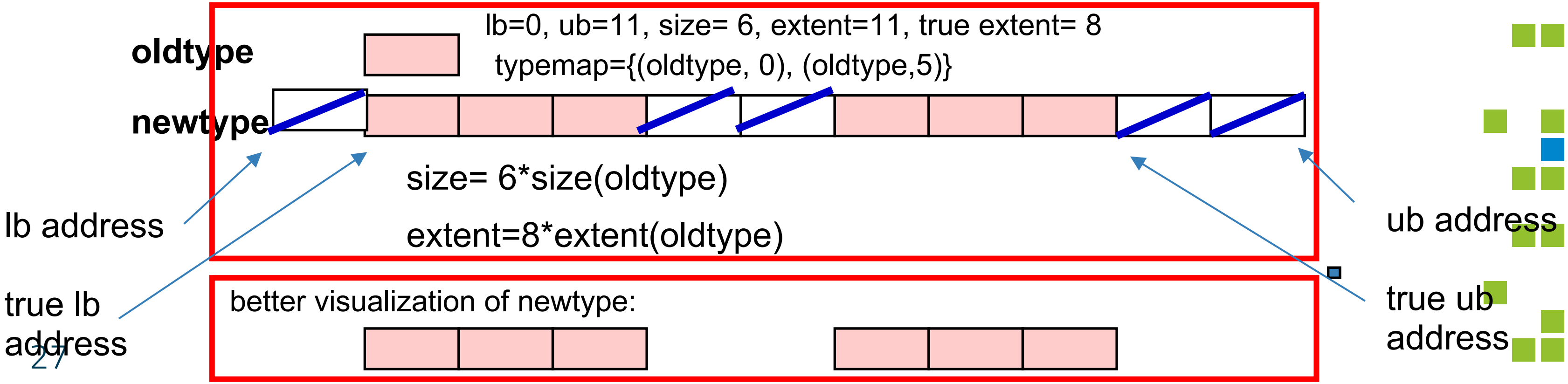
Example – struct datatype




```
struct buf{
   double dblrank;
   int intrank;
}sBuf, rBuf, sum;

array_of_blocklengths[0] = 1;
array_of_blocklengths[1] = 1;
MPI_Get_address(&sBuf.dblrank, &dbladdress);
MPI_Get_address(&sBuf.intrank, &intaddress);
array_of_displacements[0] = (MPI_Aint) 0;
array_of_displacements[1] = MPI_Aint_diff(intaddress, dbladdress);
array_of_types[0] = MPI_DOUBLE;
array_of_types[1] = MPI_INT;

MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements, array_of_types, &newtype);
MPI_Type_commit(&newtype);

sBuf.dblrank=(double)myRank;  sBuf.intrank=myRank;  sum.dblrank=0.0;  sum.intrank=0;
for( i = 0; i < uniSize; i++) {
  ierror=MPI_Isend(&sBuf, 1, newtype, dest, 100, MPI_COMM_WORLD, &request);
  ierror=MPI_Recv(&rBuf, 1, newtype, src, 100, MPI_COMM_WORLD, &status);
  ierror=MPI_Wait(&request, &status);
  sBuf=rBuf;
  sum.dblrank += rBuf.dblrank;
  sum.intrank += rBuf.intrank;}

MPI_Type_free(&newtype);
```

Size and Extent of a Datatype

- When sending multiple datatypes
  - datatypes are read from memory separated by their extent
  - extent = spans from first to last byte. (upper bound-lower bound)
  - Padding may occur due to data alignment reasons
  - lower bound = describes where the datatype starts
- For basic datatypes, extent is the size of the object
  - size = number of bytes that really transferred.



Size and Extent of a Datatype

- Using addresses, is still unsafe for arrays of struct because of possible alignments of the last member of the structure.
- If you have derived datatype consisting of derived dataype consisting of derived datatype consisting of… and each of them has lb and ub set already
  - No way to erase upper/lower bound markers once they are set

- Extent can be changed with routine: `MPI_Type_create_resized()`
  - The new, resized datatype must be committed before use

- Extent of a datatype can be obtained: `MPI_Type_get_extent()`
- True extent can be obtained: `MPI_Type_get_true_extent()` (ignoring UB and LB markers, all gaps in the middle are still considered)
- Total number of bytes of the entry datatype: `MPI_Type_size()`

MPI_Type_create_resized

- C:

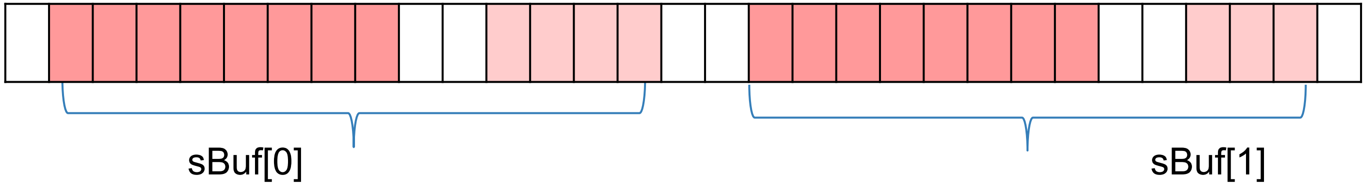
```
int MPI_Type_create_resized(MPI_Datatype oldtype,MPI_Aint newlb,
                            MPI_Aint newextent, MPI_Datatype *newtype)
```

- Fortran:

```
 MPI_TYPE_CREATE_RESIZED(oldtype, newlb, newextent, newtype,
ierror)
    TYPE(MPI_Type) :: oldtype, newtype
    INTEGER (KIND=MPI_ADDRESS_KIND) :: newlb, newextent
    INTEGER, OPTIONAL :: ierror
```

- 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".

Example – array of structs datatype




```
struct buf{
     double dblrank;
     int intrank;
 }sBuf[2], rBuf[2], sum[2];

 if(myRank==0){
    int bufsize;
    MPI_Aint buflb, bufextent, buftrueextent;
    MPI_Type_size(newtype, &bufsize);
    MPI_Type_get_extent(newtype, &buflb, &bufextent);
    MPI_Type_get_true_extent(newtype, &buflb, &buftrueextent);
    printf("Size= %d, Extent=%d, True extent=%d\n", bufsize, (int)bufextent,
(int)buftrueextent);
 }
```

```
Size= 12, Extent=16, True extent=12
```

Example – resize

```
  MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements,
array_of_types, &newtype);
  MPI_Type_create_resized(newtype, (MPI_Aint) 0, (MPI_Aint) sizeof(sBuf[0]),
&newtype_resized);
  MPI_Type_commit(&newtype_resized);


for( i = 0; i < uniSize; i++) {
    ierror=MPI_Isend(&sBuf, 2, newtype_resized, dest, 100, MPI_COMM_WORLD, &request);
    ierror=MPI_Recv(&rBuf, 2, newtype_resized, src, 100, MPI_COMM_WORLD, &status);
    ierror=MPI_Wait(&request, &status);
    for(j=0; j<2; j++){
      sBuf[j]=rBuf[j];
      sum[j].dblrank += rBuf[j].dblrank;
      sum[j].intrank += rBuf[j].intrank;
    }
  }


 MPI_Type_free(&newtype_resized);
```

Other Derived Datatype Creation Routines

`MPI_Type_create_hvector()`	like vector, but the stride is specified in byte
`MPI_Type_create_darray()`	distribution of a ndim-array into a grid of ndim-logical processes
`MPI_Type_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_INDEXED, except that the block-length is the same for all blocks.
`MPI_Type_create_hindexed_block()`	Create an hindexed datatype with constant-sized blocks