Lecture 14 – MPI with Derived Datatypes and Virtual Topologies

- **Limitations of Point-to-Point Communication:**
- **So far, all point-to-point communication has involved contiguous buffers containing elements of the same fundamental types provided by MPI such as MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, etc. This is rather limiting since one often wants to:**
  - Send messages with different kinds of datatypes (say a few integers followed by a few reals.)
  - Send non-contiguous data (such as a sub-block of a matrix or an unstructured subset of a one-dimensional array.)
- **These two goals could easily be accomplished by either sending a number of messages with all data of one type and by copying the non-contiguous data to a contiguous array that is later sent in a single message.**

Point-to-Point Communication

- **There are several disadvantages:**
  - A larger number of messages than necessary may need to be sent.
  - Additional inefficient memory-to-memory copying may be required to fill contiguous buffers.
  - Resulting code is needlessly complicated.
- **MPI *derived datatypes* allow us to carry out these operations: we can send/receive messages with non-contiguous data of different types without the need for multiple messages or additional memory-to-memory copying.**
- **MPI *derived datatypes* also allow us to send/receive derived datatypes consisting of mixed-type words/arrays**
  - Note that the basic MPI_Send/Receive commands covered thus far will not allow communication of derived datatypes without constructing an MPI derived datatype

Definition of an MPI Derived Datatype

- **A *general datatype* is an object that specifies two things:**
  - A sequence of basic datatypes.
  - A sequence of integer (byte) displacements.
- **The displacements do not need to be positive, distinct, or in any particular order.**
- **The sequence of pairs of datatype-displacement is called a *type map*, while the sequence of datatypes alone is called the *type signature*.**
- **NOTE that the derived datatype does not pertain to the actual data but rather a descriptor (map) of the data to be sent/received**

Definition of an MPI Derived Datatype

- **For example:**

Typemap =(type_0,disp_0),(type_1,disp_1),...,(type_n,disp_n)

can be such a map, whose signature would be

Typesig = type_0, type_1,..., type_n

- **Derived datatypes can be used in all communication routines (MPI_SEND, MPI_RECV, ...) in place of the basic datatypes by defining them with MPI data type constructors.**
- **In addition, by using MPI's derived datatype constructors, you can automatically select data of interest such as sub-portions of a block of data, both in structured and unstructured fashion.**

MPI Datatype Constructors

- **MPI has 3 basic constructors that allow the user to define a general datatype**
  - MPI_Type_contiguous: builds a derived type whose elements are contiguous entries
  - MPI_Type_vector: builds a derived type whose elements are equally spaced entries of an array
  - MPI_Type_indexed: builds a derived type whose elements are arbitrary entries of an array

MPI_TYPE_CONTIGUOUS

- **The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype element into contiguous locations.**

- **Example: Say that we have created a datatype in our program**

```
TYPE :: GRID
      REAL*8 :: X,Y,Z
      REAL*8 :: TEMP
END TYPE GRID
```

- **And say that we want to pass the GRID datatype to another processor**
- **We must first create an MPI datatype that describes it. In this case, we have 4 double-precision words that create the GRID datatype. We can construct the MPI datatype, GRIDTYPE, by concatenating 4 double-precision words:**

```
CALL MPI_Type_contiguous( 4, MPI_DOUBLE_PRECISION, GRIDTYPE)
```


MPI_TYPE_CONTIGUOUS

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
[IN count] replication count (nonnegative integer)
[IN oldtype] old datatype (handle)
[OUT newtype] new datatype (handle)

C:

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

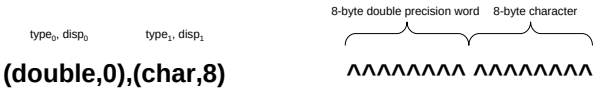
Fortran 90/95:

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE,
                    IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

- **NEWTYPE is the datatype obtained by concatenating count copies of oldtype.**

Another Example: MPI_TYPE_CONTIGUOUS

- **Say we have created a derived datatype (old_type) that has a map:**



(double,0),(char,8)

and we want to replicate it by count = 3.

- **A call to MPI_Type_contiguous(count,old_type,new_type,ierror) would produce a new_type with map:**

(double,0),(char,8),(double,16),(char,24),(double,32),(char,40)

MPI_TYPE_VECTOR

- The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks.
- Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

MPI_TYPE_VECTOR

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

[IN count] number of blocks (nonnegative integer)

[IN blocklength] number of elements in each block (nonnegative integer)

[IN stride] number of elements between start of each block (integer)

[IN oldtype] old datatype (handle)

[OUT newtype] new datatype (handle)

C:

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

Fortran 90/95:

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
                NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
    IERROR
```


Example: MPI_TYPE_VECTOR

- Let's say that you want to send a single column of an array, A, that has dimensions IMAX x JMAX to another processor. You can create an MPI datatype that describes the elements of A to be sent

```
CALL MPI_Type_Vector(JMAX, 1, IMAX, MPI_DOUBLE_PRECISION,
                     COLUMNTYPE, IERROR)
```

- Then you could send say the imax-1 column of A to proc-1 with

```
CALL MPI_TYPE_COMMIT(COLUMNTYPE,IERROR)
CALL MPI_SEND(A(imax-1,1),1,COLUMNTYPE,1,1,
              MPI_COMM_WORLD,IERROR)
```


Another Example: MPI_TYPE_VECTOR

- Assume again that the oldtype has map

  (double,0),(char,8)

- A call to

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

MPI_TYPE_VECTOR(2,3,4,oldtype,newtype,ierror)
will create the newtype derived datatype with type map:

(double,0),(char,8), (double,16),(char,24), (double,32),(char,40),

(double,64),(char,72), (double,80),(char,88), (double,96),(char,104)

MPI_TYPE_VECTOR and MPI_TYPE_HVECTOR

- A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to

  MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to

  MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), where n is arbitrary.

- The function MPI_TYPE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements.

MPI_TYPE_INDEXED

- The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths,
array_of_displacements, oldtype, newtype)

[IN count] number of blocks (nonnegative integer)

[IN array_of_blocklengths] number of elements per block (array of nonnegative integers)

[IN array_of_displacements] displacement for each block, in multiples of oldtype extent (array of integers)

[IN oldtype] old datatype (handle)

[OUT newtype] new datatype (handle)

MPI_TYPE_INDEXED

C:

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

Fortran 90/95:

```
MPI_TYPE_INDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,
                 ARRAY_OF_DISPLACEMENTS, OLDTYPE,
                 NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*),OLDTYPE, NEWTYPE, IERROR
```


MPI_TYPE_COMMIT and MPI_TYPE_FREE

- An MPI derived datatype object *has to be committed* before it can be used in a communication. A committed datatype can still be used as a argument in datatype constructors.
- There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

[INOUT datatype] datatype that is committed (handle)

C:

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

Fortran 90/95:

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR
```

- The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

MPI_TYPE_COMMIT and MPI_TYPE_FREE

- **MPI_Type_Free marks the derived datatype object associated with datatype for deallocation and sets the derived datatype to MPI_DATATYPE_NULL.**
- **Any communication that is currently using this datatype will complete normally. Derived datatypes that were defined from the freed datatype are not affected.**

MPI_TYPE_FREE(datatype)
[INOUT datatype] datatype that is freed (handle)
C:
int MPI_Type_free(MPI_Datatype *datatype)
Fortran 90/95:
MPI_TYPE_FREE(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

MPI_TYPE_COMMIT and MPI_TYPE_FREE

- **For example,**

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
      ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
      ! now type1 can be used for communication
type2 = type1
      ! type2 can be used for communication
      ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 5, MPI_REAL, type1, ierr)
      ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
      ! now type1 can be used anew for communication
```

Note that the derived datatype is declared an integer

Creates an MPI datatype of 5 real contiguous numbers

Creates an MPI datatype consisting of 3 blocks (rows) of 5 real numbers (elements) with a stride of 5 numbers (20 bytes) between successive elements

- **Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.**