Lecture 15 – Examples of MPI Derived Datatypes and Virtual Toplogies

- In the last lecture, we discussed the creation and use of MPI derived datatypes
- These derived datatypes describe (or map) the information that will ultimately be sent or received
- The derived datatypes do NOT pertain to the actual data, but are used by the MPI commands to describe the type of data being sent or received

Example: Sending a Section of a 3D Array

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

!     extract the section a(1:17:2, 3:11, 2:10)
!     and store it in e(:,:,:).   Note this is a 9x9x9 block.

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
!     find the extent (stride) of MPI_REAL in bytes on this computer
      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

!     create datatype for a 1D section
!     CALL MPI_TYPE_VECTOR( count, blocklength, stride, oldtype, newtype)
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)
!     creates the datatype for the first index of "a"
```

Note this new MPI routine

Example: Sending a Section of a 3D Array

```
!     extract the section a(1:17:2, 3:11, 2:10)
!     create datatype for a 2D section
!     CALL MPI_TYPE_HVECTOR( count, blocklength, stride(bytes), oldtype,
!                            newtype)
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)
!     creates the datatype for the first and second indicies of "a"

!     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, 1,
     &                       threeslice, ierr)
!     Creates the datatype for the first-third indicies of "a"

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
!     CALL MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,
!                       recvbuf,recvcount,recvtype,source, recvtag,
!                       comm,status,ierror)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0,
     &                  e, 9*9*9, MPI_REAL, myrank, 0,
     &                  MPI_COMM_WORLD, status, ierr)
```

Note that the stride is that of the original a array

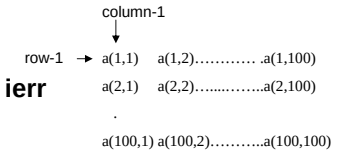
Example: Transpose of a Matrix

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

!     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
!     find the extent of MPI_REAL on this computer
      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

!     create datatype for one row
!     CALL MPI_TYPE_VECTOR( count, blocklength, stride, oldtype,
!                           newtype)
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```



Remember that a is stored in column-major order in Fortran

Example: Transpose of a Matrix

```
!    create datatype for 100 rows (entire matrix) in row-major order
!    CALL MPI_TYPE_HVECTOR( count, blocklength, stride(bytes),
!                           oldtype,newtype)
     CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
!
     CALL MPI_TYPE_COMMIT( xpose, ierr)
!
!    send matrix in row-major order and receive in column-major order
!    CALL MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,
!                      recvbuf,recvcount,recvtype,source,recvtag,
!                      comm,status,ierror)
     CALL MPI_SENDRECV( a, 1, xpose, myrank, 0,
    &                   b, 100*100,MPI_REAL, myrank, 0,
    &                   MPI_COMM_WORLD,status, ierr)
```


Processor Topologies

- **When writing a parallel program, you sometimes can arrange the processors into some sort of topology for communication purposes.**
  - In complicated solvers, you may have a 3D grid (single block Navier-Stokes solver), or a 3D periodic grid (direct numerical simulation of turbulence in a periodic box), or a more general graph connecting processors in an irregular fashion.
- **As a programmer, you have to keep track of the processors (and their rank) that each processor needs to communicate with. Although this could be done by the programmer (as in projects-2 and 4), MPI has a number of functions/subroutines that facilitate this procedure greatly, especially in the case of n-dimensional structured (periodic or non-periodic) Cartesian topologies.**
- **The more general case of a graph is not discussed in the notes, but can be found in the documentation in the web.**

Processor Topologies

- **In Project-2, you could break up the single block into multiple blocks with a Cartesian topology.**
- **In Project-4, you could partition (decompose) the blocks in that Cartesian topology into multiple processors**
  - However, to be more general, we will not use MPI Cartesian routines in projects-4 and 5.
- **These 2 steps could alternatively be performed using MPI virtual topology routines in conjunction with MPI derived datatype utilities.**
  - Some of these routines overlay a Cartesian structure of processors onto a Cartesian topology of blocks
  - Other of these routines are more general in that they overlay a set of processors onto a general graphed domain (useful for unstructured-grid topologies)
  - These routines only set up the processor topology. It is up to the programmer to perform decomposition of the domain onto this resulting processor topology. MPE routines may be used for this.

Virtual Topologies

- **Virtual topologies in MPI have the dual advantage that**
  - In the midst of communication, it may be easier to refer to the processors that we need to communicate with through MPI built-in functions than by calculating the processor ranks ourselves.
  - *If* (and this is a BIG if) the MPI implementation is *smart* it can take advantage of the virtual topology constructors to infer the processor arrangement you would like to use and map processes more efficiently to the underlying hardware. (Not generally the case, however)
- **Note that despite the fact that you may have defined a virtual topology that only has connections to nearest neighbors, you can still communicate with all the processors in the communicator if you continue to use processor ranks as the arguments to the point-to-point functions.**

Virtual Topologies

- **The functions MPI_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies of processors, respectively. These topology creation functions are *collective*. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.**

- **The topology creation functions take as input an existing communicator, comm_old, which defines the set of processes on which the topology is to be mapped. A new communicator comm_topol is created that carries the topological structure as cached information.**
  - In analogy to function MPI_COMM_CREATE (which creates a new communicator out of a sub-group of processes), no cached information propagates from comm_old to comm_topol. We will discuss this later in the course.

MPI_CART_CREATE

- **Let's say that we wanted to take your single block heat conduction code from project-1 and simply assign a NxM set of processors to it. We could use MPI_CART_CREATE to describe how the processor topology.**
- **Note that this creates the topology but does not map the computational grid to the topology.**

MPI_CART_CREATE

- **MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension.**
  - For each coordinate direction, one specifies whether the processor structure is periodic or not.
  - Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary.
  - The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI_CART_CREATE

- **MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached.**
  - If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine).
  - If the total size of the Cartesian processor grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL. The call is erroneous if it specifies a processor grid that is larger than the group size.

MPI_CART_CREATE

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

[IN comm_old] input communicator (handle)

[IN ndims] number of dimensions of Cartesian grid (integer)

[IN dims] integer array of size ndims specifying the number of processes in each dimension

[IN periods] logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension

[IN reorder] ranking may be reordered (true) or not (false) (logical)

[OUT comm_cart] communicator with new Cartesian topology (handle)

C:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                    int *periods, int reorder, MPI_Comm *comm_cart)
```

Fortran 90/95:

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
                COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```


Cartesian Topology Utility Functions

- The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian processor topology information that was associated with a communicator by MPI_CART_CREATE.

MPI_CARTDIM_GET(comm, ndims)

[IN comm] communicator with Cartesian structure (handle)

[OUT ndims] number of dimensions of the Cartesian structure (integer)

C:

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Fortran 90/95:

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
INTEGER COMM, NDIMS, IERROR
```


MPI_CART_GET

MPI_CART_GET(comm, maxdims, dims, periods, coords)

[IN comm] communicator with Cartesian structure (handle)

[IN maxdims] length of vector dims, periods, and coords in the calling program (integer)

[OUT dims] number of processes for each Cartesian dimension (array of integer)

[OUT periods] periodicity (true/ false) for each Cartesian dimension (array of logical)

[OUT coords] coordinates of calling process in Cartesian structure (array of integer)

C:

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                 int *periods, int*coords)
```

Fortran 90/95:

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS,
             IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)
```


MPI_CART_RANK

- For a process group with Cartesian structure, the function MPI_CART_RANK takes the process coordinates in the coords array and returns its rank in rank.
- For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0 or coords(i) >dims(i), it is shifted back to the interval 0:coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

MPI_CART_RANK(comm, coords, rank)

[IN comm] communicator with Cartesian structure (handle)

[IN coords] integer array (of size ndims) specifying the Cartesian coordinates of a process

[OUT rank] rank of specified process (integer)

C:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Fortran 90/95:

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
INTEGER COMM, COORDS(*), RANK, IERROR
```

MPI_CART_COORDS

- MPI_CART_COORDS takes the rank of the process rank and returns its Cartesian coordinates in the array coords (of length maxdims).

MPI_CART_COORDS(comm, rank, maxdims, coords)

[IN comm] communicator with cartesian structure (handle)

[IN rank] rank of a process within group of comm (integer)

[IN maxdims] length of vector coord in the calling program (integer)

[OUT coords] integer array (of size ndims) containing the cartesian coordinates of specified process (integer)

C:

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int*coords)
```

Fortran 90/95:

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```


MPI_GRAPH_NEIGHBORS

- MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general, graph topology (like that used in an unstructured data-structure).

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

[IN comm] communicator with graph topology (handle)

[IN rank] rank of process in group of comm (integer)

[OUT nneighbors] number of neighbors of specified process (integer)

C:

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
                              int *nneighbors)
```

Fortran 90/95:

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS,
                          IERROR)
INTEGER COMM, RANK, NNEIGHBORS, IERROR
```


MPI_GRAPH_NEIGHBORS

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

[IN comm] communicator with graph topology (handle)

[IN rank] rank of process in group of comm (integer)

[IN maxneighbors] size of array neighbors (integer)

[OUT neighbors] ranks of processes that are neighbors to specified process (array of integer)

C:

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int
                        maxneighbors, int*neighbors)
```

Fortran 90/95:

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS,
                    NEIGHBORS, IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*),
    IERROR
```


Cartesian Shift Coordinates

- If the process topology is a Cartesian structure, a MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send.

```
MPI_SendRecv(sendbuf,sendcount,sendtype,dest,sendtag,
             recvbuf,recvcount,recvtype,source,recvtag,
             comm,status,ierror)
```

[IN sendbuf] data to be sent (choice)

[IN sendcount] number of elements in sendbuf (integer)

[IN sendtype] type of data contained in sendbuf (handle)

[IN dest] processor rank of destination (integer)

[IN sendtag] tag of send message (integer)

[OUT recvbuf] data to be sent (choice)

[IN recvcount] number of elements in recvbuf (integer)

[IN recvtype] type of data contained in recvbuf (handle)

[IN source] processor rank of source (integer)

[IN recvtag] tag of recv message (integer)

Cartesian Shift Coordinates

- **If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.**

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

[IN comm] communicator with Cartesian structure (handle)

[IN direction] coordinate dimension of shift (integer)

[IN disp] displacement (> 0: upwards shift, < 0: downwards shift) (integer)

[OUT rank_source] rank of source process (integer)

[OUT rank_dest] rank of destination process (integer)

Cartesian Shift Coordinates

C:

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int*rank_source, int *rank_dest)
```

Fortran 90/95:

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
               RANK_DEST, IERROR)
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST,
   IERROR
```

- **The direction argument indicates the dimension of the shift, i.e., the coordinate which value is modified by the shift. The coordinates are numbered from 0 to ndims-1, when ndims is the number of dimensions.**

- **Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.**

Cartesian Shift Coordinates

- **Example: The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of REALs is *stored one element per process*, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.**

```
! find process rank
    CALL MPI_COMM_RANK(comm, rank, ierr))
! find Cartesian coordinates
    CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
! compute shift source and destination
! MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest, ierr)
    CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
! skew array
! MPI_SENDRECV_REPLACE(buf,count,datatype,dest,sendtag,source,
!                      recvtag,comm,status,ierror)
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source,
   &                          0, comm, status, ierr)
```


Project-3

- **Write a multi-block solver for your simulation problem that will run on a single processor**
  - This is an intermediate step prior to distributing the blocks over P-processors and adding message-passing to allow parallel computing
  - Use the data structure and boundary-condition data files that you constructed under Project-2
  - Use one of the methods (“on the fly”, “accumulation operators”, or “halo/ghost cells”) to deal with inter-block boundaries
- **This code should read the multi-block grid plot3d (or other format) files along with the connectivity file, initialize the temperature (or read a multi-block initial temperature file), and run.**
- **Demonstrate that you can get the same solution and same convergence as the 101x101 Dirichlet Project 1 solution with the decompositions that you generated in Project-2**

Project-3

- **Due Friday, November 8th**
  - A description of your equations, program, and method for dealing with neighbor information
  - A listing of your multi-block simulation code for the single processor
  - A plot of your multi-block solution for the sheet metal problem (pick one of your decompositions from project-2)
  - A direct comparison of your convergence rates between the single-block and multi-block solvers for the plate problem. *A plot of the single-block and multi-block convergence histories is required*.
  - A direct comparison of your solution times between the single-block and multi-block solvers for the plate problem. Use WOPR for both!