

## **Lecture 10 – 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(kind=8) :: X,Y,Z
    REAL(kind=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:**

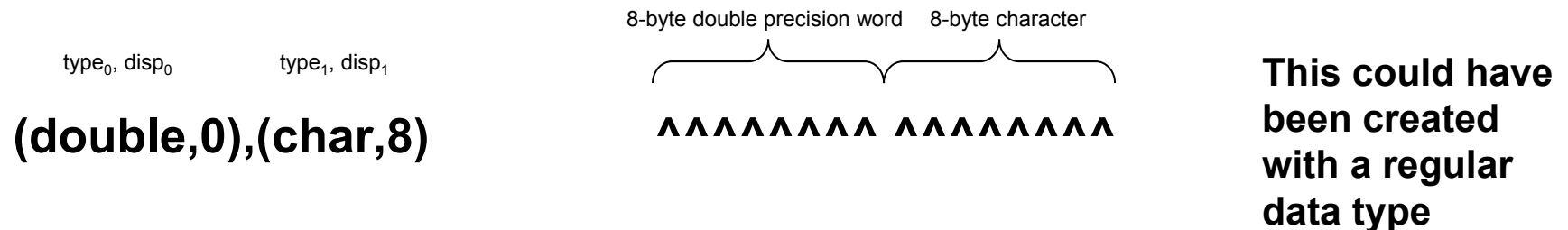
**MPI\_TYPE\_CONTIGUOUS(COUNT, OLDDTYPE, NEWTYPE,  
IERROR)**

**INTEGER COUNT, OLDDTYPE, 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:



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

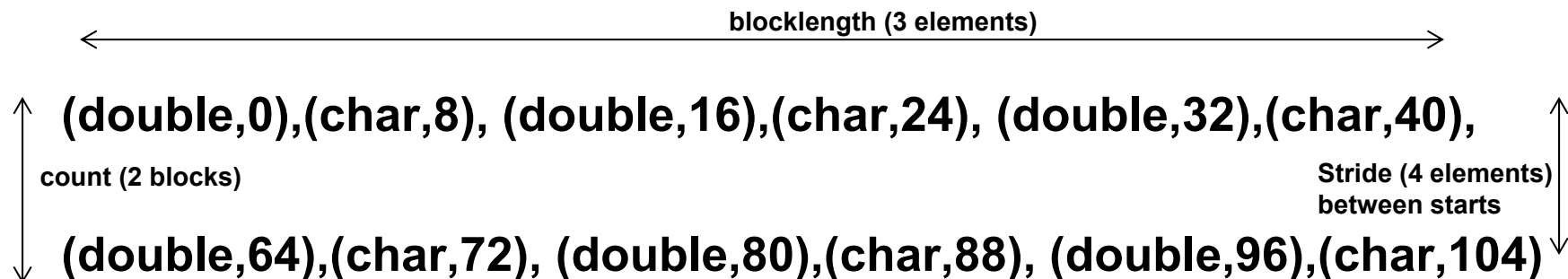
- Assume again that the oldtype has map

(double,0),(char,8)

8 bytes

8 bytes

- 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:



## **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 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 an 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**

Note that the derived datatype is declared an integer

**CALL MPI\_TYPE\_CONTIGUOUS(5, MPI\_REAL, type1, ierr)**

Creates an MPI datatype of 5 real contiguous numbers

**! 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)**

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

**! new uncommitted redifined type1 object created**

**CALL MPI\_TYPE\_COMMIT(type1, ierr)**

**! now type1 can be used anew for communication**

- 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.

## Further Examples of MPI Derived Datatypes and Virtual Topologies

- **So far, we have 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)    Note this new MPI  
                                                    routine
```

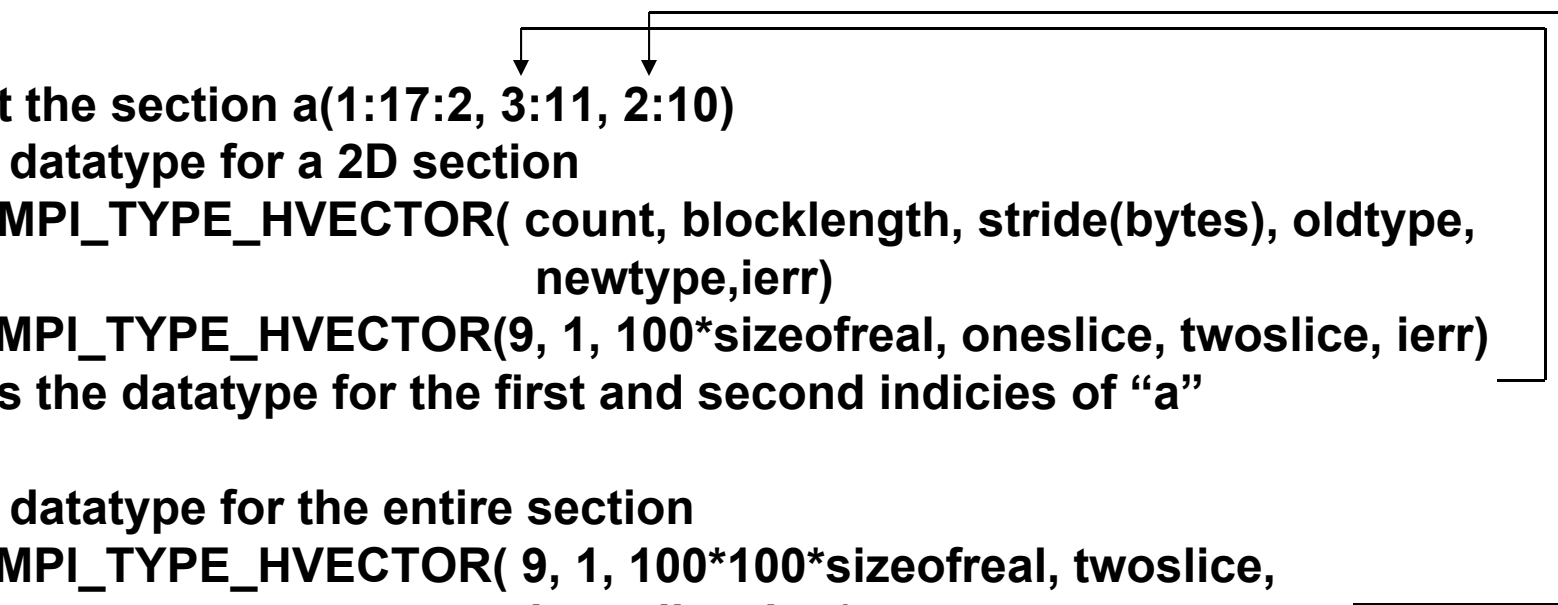
```
! create datatype for a 1D section
```

```
! 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"
```

## 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,ierr)
! CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)
! creates the datatype for the first and second indices of "a"

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

To be described later

send/recv to myself

## Example: Transpose of a Matrix

```
REAL a(100,100), b(100,100)
```

```
INTEGER row, xpose, sizeofreal, myrank, ierr
```

```
INTEGER status(MPI_STATUS_SIZE)
```

column-1  
↓  
row-1 → a(1,1) a(1,2).....a(1,100)  
a(2,1) a(2,2).....a(2,100)  
.  
a(100,1) a(100,2).....a(100,100)

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

```
! 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)
```

## 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 on 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 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 partitioning 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 the processor topology.**
- **Note that this creates the topology but does not map the computational grid to the topology.**
- **Note that we are not doing this for our projects-2 and 3 in order to make our codes more general.**

## **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.
- 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 – MPI\_SENDRECV

- 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 approximately the same convergence as the 501x501 Dirichlet Project 1 solution with the 10x10 decompositions that you generated in Project-2**

## Project-3

- **Due Thursday, November 5<sup>th</sup>**
  - 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 (10x10 block) solution for the 501x501 point sheet metal problem
  - A direct comparison of your convergence rates between the single-block and multi-block (10x10 block) solvers for the 501x501 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 (10x10 block) solvers for the 501x501 plate problem. Use HPC1 for both! You are allowed to make improvements to your project-1 solver).