1. **MPI Datatypes**
   - Datatypes in MPI
   - Derived Datatypes in MPI

2. **Extended example of vector type**
   - Cartesian Grids
   - Cartesian communicator example
Outline

1. **MPI Datatypes**
   - Datatypes in MPI
   - Derived Datatypes in MPI

2. Extended example of vector type
   - Cartesian Grids
   - Cartesian communicator example
The MPI standard defines many built in datatypes, mostly mirroring standard C/C++ or FORTRAN datatypes. These are sufficient when sending single instances of each type. They are also usually sufficient when sending contiguous blocks of a single type. Sometimes, however, we want to send non-contiguous data or data that is comprised of multiple types. MPI provides a mechanism to create derived datatypes that are built from simple datatypes.
Typemaps

- Derived datatypes in MPI are described by a **typemap**.
- Typemaps consist of an order pair or sequence of ordered pairs each containing:
  - a basic datatype
  - a displacement (integer offset)
- For example, a typemap might consist of \{(double,0),(char,8)\}
  indicating the type has two elements:
  - a double precision floating point value starting at displacement 0, and
  - a single character starting at displacement 8.
Types also have **extent**, which indicates how much space is required for the type.

The extent of a type may be more than the sum of the bytes required for each component.

For example, on a machine that requires double-precision numbers to start on an 16-byte boundary the type {((double,0),(char,8))} will have an extent of 16 even though it only requires 9 bytes.
1. **MPI Datatypes**
   - Datatypes in MPI
   - Derived Datatypes in MPI

2. **Extended example of vector type**
   - Cartesian Grids
   - Cartesian communicator example
MPI provides for user-constructed datatypes to handle a wide variety of situations. Constructors exist for the following types of derived datatypes:

- Contiguous
- Vector
- Hvector
- Indexed
- Hindexed
- Indexed_block
- Struct

The “H” routines are the same as the similarly named types except that strides and block displacements are specified in bytes.
Creating and using a new datatype

Two steps are necessary to create and use a new datatype in MPI:

1. *Create* the type using one of MPI’s type construction routines (explained next),
2. *Commit* the type using `MPI_Type_commit()`.

Once a type has been committed it may be used in send, receive, and other buffer operations.

A committed type can be released with `MPI_Type_free()`.
The contiguous datatype allows for a single type to refer to multiple contiguous elements of an existing datatype.

```c
int MPI_Type_contiguous(
    int count, // replication count
    MPI_Datatype oldtype, // old datatype
    MPI_Datatype* newtype) // new datatype
```
The contiguous datatype allows for a single type to refer to multiple contiguous elements of an existing datatype.

```c
int MPI_Type_contiguous(
    int count, // replication count
    MPI_Datatype oldtype, // old datatype
    MPI_Datatype* newtype) // new datatype
```

The new datatype is essentially an array of `count` elements having type `oldtype`. For example, the following two code fragments are equivalent:

```c
MPI_Send(a, n, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

and

```c
MPI_Datatype rowtype;
MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
MPI_Send(a, 1, rowtype, dest, tag, MPI_COMM_WORLD);
```
The vector datatype is similar to the contiguous datatype but allows for a constant non-unit *stride* between elements.

```c
int MPI_Type_vector(
    int count,      // number of blocks
    int blocklength, // number of elements in each block
    int stride,     // number of elements between each block
    MPI_Datatype oldtype,   // old datatype
    MPI_Datatype* newtype)  // new datatype
```

For example, suppose an $nx \times ny$ Cartesian grid is stored so data in rows (constant $y$) is contiguous. The following two types can be used to communicate a single row and a single column of the grid:

```c
MPI_Datatype xSlice, ySlice;
MPI_Type_vector(nx, 1, ny, MPI_DOUBLE, &xSlice);
MPI_Type_vector(ny, 1, 1, MPI_DOUBLE, &ySlice);
MPI_Type_commit(&xSlice);
MPI_Type_commit(&ySlice);
```
Vector type

The vector datatype is similar to the contiguous datatype but allows for a constant non-unit \textit{stride} between elements.

\texttt{int MPI_Type_vector(}
\begin{verbatim}
    int count,     // number of blocks
    int blocklength, // number of elements in each block
    int stride,     // number of elements between each block
    MPI_Datatype oldtype,  // old datatype
    MPI_Datatype* newtype) // new datatype
\end{verbatim}
\texttt{)}

For example, suppose an $n_x \times n_y$ Cartesian grid is stored so data in rows (constant $y$) is contiguous. The following two types can be used to communicate a single row and a single column of the grid:

\begin{verbatim}
MPI_Datatype xSlice, ySlice;
MPI_Type_vector(nx, 1, ny, MPI_DOUBLE, &xSlice);
MPI_Type_vector(ny, 1, 1, MPI_DOUBLE, &ySlice);
MPI_Type_commit(&xSlice);
MPI_Type_commit(&ySlice);
\end{verbatim}
Note: In contrast to how we view matrices, in C/C++ elements in a row of a Cartesian grid have non-unit stride: $u[0][0]$ and $u[1][0]$ are not contiguous.
In general, the last index corresponds to the dimension with unit stride and the first index corresponds to the dimension with greatest stride.
double** u = new double* [nx];
u[0] = new double [nx * ny];

for ( i = 1; i < nx; i++ )
    u[i] = &u[0][i * ny];

- Note that by this construction u is a pointer to a pointer.
- u[0] is a pointer to the start of the first grid column.
- Consecutive locations in memory correspond to consecutive values of the last array index; in this case that is along the y axis.
- If the grid was 3-dimensional, consecutive memory locations would be along z, consecutive z-columns would be adjacent on the y axis, and finally yz-slices would be adjacent along the x axis.
Indexed type

The indexed datatype provides for varying strides between elements.

```c
int MPI_Type_indexed(
    int count,            // number of blocks
    int* blocklengths,    // number of elements per block
    int* displacements,   // displacement for each block
    MPI_Datatype oldtype, // old datatype
    MPI_Datatype* newtype) // new datatype
```

This generalizes the vector type; instead of a constant stride, blocks can be of varying length and displacements. For example, the code fragment

```c
int blocklen [] = {4, 2, 2, 6, 6};
int disp [] = {0, 8, 12, 16, 23};
MPI_Datatype mytype;
MPI_Type_indexed (5, blocklen, disp, MPI_DOUBLE, & mytype);
MPI_Type_commit (& mytype);
```

defines a type that corresponds to the shaded cells:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>
```

CPS343 (Parallel and HPC)  
MPI derived datatypes & Cartesian grids  
Spring 2020
Indexed type

The indexed datatype provides for varying strides between elements.

```c
int MPI_Type_indexed(
    int count,       // number of blocks
    int* blocklengths,   // number of elements per block
    int* displacements,   // displacement for each block
    MPI_Datatype oldtype,       // old datatype
    MPI_Datatype* newtype)       // new datatype
```

This generalizes the vector type; instead of a constant stride, blocks can be of varying length and displacements. For example, the code fragment

```c
int blocklen[] = {4, 2, 2, 6, 6};
int disp[] = {0, 8, 12, 16, 23};
MPI_Datatype mytype;
MPI_Type_indexed(5, blocklen, disp, MPI_DOUBLE, &mytype);
MPI_Type_commit(&mytype);
```

defines a type that corresponds to the shaded cells:

```
0  8  12  16  23
```
```c
do something ...
```
**Struct type**

The most general constructor allows for the creation of types representing general C/C++ structs/classes.

```c
int MPI_Type_create_struct(
    int count,               // number of blocks
    int* blocklengths,      // number of elements per block
    MPI_Aint* displacements, // byte displacement of each block
    MPI_Datatype* datatypes, // type of elements in each block
    MPI_Datatype* newtype)   // new datatype
```

The type `MPI_Aint` is an `address` type; variables of this type can hold valid addresses (byte offsets from the start of memory).
Outline

1. MPI Datatypes
   - Datatypes in MPI
   - Derived Datatypes in MPI

2. Extended example of vector type
   - Cartesian Grids
   - Cartesian communicator example
A sample grid

- Typical $10 \times 10$ grid
- Lines separate grid elements
Grid boundary

- Grid boundary in cyan
- Typically contains constant data
- Grid points on interior change
Grid boundary

- Grid boundary in cyan
- Typically contains constant data
- Grid points on interior change
- We will assume grid elements are updated using a 5-point finite-difference stencil
Suppose we want to partition the domain into four subdomains.

Could be done vertically or horizontally.

In this case, we’ll partition in both directions.
Partitioning the grid

- To update the grid point marked with the black dot information is needed from its four neighbors.
Partitioning the grid

- To update the grid point marked with the black dot information is needed from its four neighbors.
- The stencil shows the grid points needed for the update.
Partitioning the grid

- To update the grid point marked with the black dot information is needed from its four neighbors.
- The stencil shows the grid points needed for the update.
- All accessed grid points are inside the local subdomain.
Partitioning the grid

To update the grid point marked with the black dot information is needed from its four neighbors.

The stencil shows the grid points needed for the update.

All accessed grid points are inside the local subdomain.

To update an adjacent grid point we still only need to access points inside the subdomain.
Partitioning the grid

- To update the grid point marked with the black dot information is needed from its four neighbors.
- The stencil shows the grid points needed for the update.
- All accessed grid points are inside the local subdomain.
- To update an adjacent grid point we still only need to access points inside the subdomain.
- But moving over one more, we now need information from an adjacent subdomain.
Adding interior boundaries

Assuming a message passing environment, we want to minimize communication.

To facilitate this, we create additional grid locations to hold copies of data from adjacent subdomains; these are often called ghost points or the halo region.

Still need to transfer, but now blocks can be transferred...
Adding interior boundaries

- Assuming a message passing environment, we want to minimize communication.
- To facilitate this, we create additional grid locations to hold copies of data from adjacent subdomains; these are often called *ghost points* or the *halo region*.
Assuming a message passing environment, we want to minimize communication. To facilitate this, we create additional grid locations to hold copies of data from adjacent subdomains; these are often called *ghost points* or the *halo region*. Still need to transfer, but now blocks can be transferred.
Before each new set of updates...
Adding interior boundaries

- Before each new set of updates...
- Interior boundary data is sent to processes working on adjacent subdomains
Before each new set of updates...

- Interior boundary data is sent to processes working on adjacent subdomains
- Now accesses are once again limited to local subdomain
Adding interior boundaries

Before each new set of updates...

Interior boundary data is sent to processes working on adjacent subdomains

Now accesses are once again limited to local subdomain

True boundary data can be copied, but will not be used
An important application that uses this pattern of data access is the numerical solution of boundary value problems involving partial differential equations.

Programs are iterative, and repeatedly updates grid points with various “sweeps” through the domain.

Between each sweep interior boundary data must be communicated.

Need to avoid deadlock situations; suppose both processes exchanging data send before receiving? This may or may not be a problem, depending on how `MPI_Send()` is implemented.

We can avoid any problems, however, by using the `MPI_Sendrecv()` function.
The calling sequence for `MPI_Sendrecv()` is

```c
int MPI_Sendrecv(
    void* sendbuf,     // address of send buffer
    int sendcount,     // number of elements to send
    MPI_Datatype sendtype,  // type of elements to send
    int dest,          // rank of destination
    int sendtag,       // send tag
    void* recvbuf,     // address of receive buffer
    int recvcount,     // number of elements to receive
    MPI_Datatype recvtype,  // type of elements to receive
    int source,        // rank of source
    int recvtag,       // receive tag
    MPI_Comm comm,     // communicator
    MPI_Status* status) // status object
```
1. **MPI Datatypes**
   - Datatypes in MPI
   - Derived Datatypes in MPI

2. **Extended example of vector type**
   - Cartesian Grids
   - Cartesian communicator example
Steps necessary to set up a Cartesian communicator:

1. Construct new communicator to use rather than \texttt{MPI\_COMM\_WORLD}
2. Determine my portion of grid
3. Determine my neighbors
4. Adjust boundaries as needed
5. Create necessary MPI data types
6. Communicate!
// Create Cartesian communicator
MPI_Dims_create( numProcesses, 2, dims );
MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periodic, reorder, &comm2d );
MPI_Comm_rank( comm2d, &myRank );

- `dims[]` is a two-dimensional array that contains the number of blocks in the x and y dimensions we want the grid to have. If these values are 0 then MPI chooses them for us.
- Entries in `periodic[]` are non-zero to indicate the grid is periodic in the corresponding dimension. Zero entries (as in this example) mean grid is non-periodic.
- If `reorder` is nonzero then processes can be be reassigned ranks, possibility different than those they received during MPI initialization.
- Rank will be determined relative to `comm2d` communicator, not `MPI_COMM_WORLD`. 
// Figure out the size of my portion of the grid.  
// x0, y0, x1 and y1 are the starting and ending  
// indices of both dimensions of our portion of  
// the grid.

MPI_Cart_get( comm2d, 2, dims, periodic, coords );
decompose1d( NX, dims[0], coords[0], &x0, &x1 );
decompose1d( NY, dims[1], coords[1], &y0, &y1 );

- periodic[] gets filled with 0s or 1s to indicate if the grid is periodic along the corresponding dimension.
- coords[] gets filled with coordinates (indexed from 0) of the block associated with the this process.
- decompose1d() returns the start and ending values of subinterval for this process. (See *Using MPI* by Gropp et al.)
// Figure out who my neighbors are. left, right, 
// down, and up will be set to the rank of the 
// process responsible for the corresponding block 
// relative to the position of the block we are 
// responsible for. If there is no neighbor in a 
// particular direction the returned rank will be 
// MPI_PROC_NULL which will be ignored by subsequent 
// MPI_sendrecv() calls.

MPI_Cart_shift( comm2d, 0, 1, &left, &right );
MPI_Cart_shift( comm2d, 1, 1, &down, &up );

- Second argument is shift axis: 0, 1, 2... for x, y, z...
- Third argument for displacement to neighboring block:
  > 0 for “up” shift,
  < 0 for “down” shift.
// Adjust domain bounds to account for internal domain boundary data. If we have a neighbor in a given direction (rank of neighbor is non-negative) then we need to adjust the starting or ending index.

if ( left >= 0 ) x0--;  
if ( right >= 0 ) x1++;  
if ( down >= 0 ) y0--;  
if ( up >= 0 ) y1++;  

nx = x1 - x0 + 1;  // actual x size of our grid
ny = y1 - y0 + 1;  // actual y size of our grid
// Create my portion of the grid. For the exchange
// to work properly we must have a constant stride
// in each dimension. This is accomplished by
// allocating an array of pointers then allocating
// the full data array to the first pointer. The
// remaining pointers are set to point to the start
// of each "row" of contiguous data in the single
// linear array.

double** u = new double*[nx];
u[0] = new double[nx * ny];
for ( i = 1; i < nx; i++ ) u[i] = &u[0][i * ny];
// Create datatypes for exchanging x and y slices
MPI_Type_vector( nx, 1, ny, MPI_DOUBLE, &xSlice );
MPI_Type_commit( &xSlice );

MPI_Type_vector( ny, 1, 1, MPI_DOUBLE, &ySlice );
MPI_Type_commit( &ySlice );

Recall that

- first argument is number of data blocks
- second argument is number of data elements in a block
- third argument is the *stride*
- last argument is pointer to variable for new type
// Exchange x-slices with top and bottom neighbors
MPI_Sendrecv( &u[0][ny-2], 1, xSlice, up, TAG,
&u[0][0], 1, xSlice, down, TAG,
comm2d, MPI_STATUS_IGNORE );

MPI_Sendrecv( &u[0][0], 1, xSlice, down, TAG,
&u[0][ny-1], 1, xSlice, up, TAG,
comm2d, MPI_STATUS_IGNORE );

// Exchange y-slices with left and right neighbors
MPI_Sendrecv( &u[nx-2][0], 1, ySlice, right, TAG,
&u[0][0], 1, ySlice, left, TAG,
comm2d, MPI_STATUS_IGNORE );

MPI_Sendrecv( &u[1][0], 1, ySlice, left, TAG,
&u[nx-1][0], 1, ySlice, right, TAG,
comm2d, MPI_STATUS_IGNORE );

Note the format of the first argument to each send-receive call. Doing this ensures that the correct address is passed regardless of how the array u is allocated.
Exchange example

Grid subdomain and its four neighbors
Send $x$-slice starting at $u[0][ny-2]$; receive into $x$-slice starting at $u[0][0]$.
Send $x$-slice starting at $u[0][1]$; receive into $x$-slice starting at $u[0][ny-1]$
Exchange example

Send $y$-slice starting at $u[nx-2][0]$; receive into $y$-slice starting at $u[0][0]$
Send $y$-slice starting at $u[1][0]$; receive into $y$-slice starting at $u[nx-1][0]$. 

---

**CPS343 (Parallel and HPC)**

MPI derived datatypes & Cartesian grids

Spring 2020 35 / 36
Interior nodes on subdomain can now be updated; dark gray nodes not referenced.
You may wonder what happens with a process responsible for a subdomain at the top (or bottom) of the grid executes the command

```c
MPI_Sendrecv( &u[0][ny-2], 1, xSlice, up, TAG,
               &u[0][0], 1, xSlice, down, TAG,
               comm2d, MPI_STATUS_IGNORE );
```

There is no neighboring subdomain above (or below)...

Recall that `up`, `down`, `left`, and `right` were rank values returned by `MPI_Cart_shift()`. In the case of a subdomain at the top, `up` will be `MPI_PROC_NULL`. Any send or receive operations to or from a process with this rank will be ignored.

If, however, we’d indicated the grid was periodic, then `up` would be assigned the rank of the process responsible for the bottom subdomain in the same subdomain column; in other words it wraps around.