



## MPI: Data

David McCaughan, *HPC Analyst*  
 SHARCNET, University of Guelph  
 dbm@sharcnet.ca

## Derived Data Types

- Recall we always provide a datatype argument to MPI functions
  - how can we define our own (derived) data types?
  - note that the built-in MPI data types are actually variables, not types
- MPI provides functions to create new instances of MPI\_Datatype variables for data types we define ourselves
  - MPI\_Datatype structures define the structure of data to be sent, it does not encapsulate that data (see example)

HPC Resources

## Derived Data Types (cont.)

- Building derived data types is relatively expensive
  - if we are going to bother with the overhead there must be some expected benefit
  - we should be expecting to use a derived data type extensively to warrant creating one
- Derived types in MPI are internally represented as sequences of (*datatype*, *offset*) pairs
  - *datatype* is a known data type
  - *offset* is the displacement in bytes from the start of the message where the value is located
  - the sequence of data types is the *type signature*

HPC Resources

## Derived Data Types (cont.)

- This is an important concept in understanding how we can define our own derived types
  - e.g. an array of 5 doubles might have the following type signature:
 

```
{ (MPI_DOUBLE, 0), (MPI_DOUBLE, 8),
      (MPI_DOUBLE, 16), (MPI_DOUBLE, 24),
      (MPI_DOUBLE, 32) }
```
  - MPI allows us to define our own types by constructing type signatures (internally) matching the definitions we provide

HPC Resources

## MPI Type Equivalence

- Types are considered compatible if the complete sequence of types (i.e. the type signature) in the derived data type in the receiver matches the initial sequence of types from the sender
  - i.e. sender's type signature can be longer than the receivers as long as the initial sequences matches
  - displacements are not considered in type matching
- Example: `compatibility.c`
  - modification of indexed type example (to follow)
  - the main diagonal of a matrix is sent, and received into the reverse diagonal of a matrix
  - can play a similar game with columns sent and received into rows, etc.

HPC Resources

## Committing Data Types: `MPI_Type_commit`

- Before we can use a derived data type we must "commit" the type to MPI
  - allows the library to optimize representation of the type for communication (where possible)
  - recall: type signature

```
int MPI_Type_commit
(
    MPI_Datatype *new
);
```

- `new`
  - an initialized derived data type to be committed

HPC Resources

## Contiguous Type: `MPI_Type_contiguous`

```
int MPI_Type_contiguous
(
    int count,
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- `count`
    - number of contiguous elements in the derived type
  - `type`
    - an existing data type identifying the elements in the derived type
  - `new`
    - a data type object to be initialized by this call
- Sending contiguous data
    - note that in C this is typically unnecessary
      - arrays are allocated contiguously anyway
    - type is provided for portability

HPC Resources

## Contiguous Type Example

```
...
double vals[SIZE];          /* contiguous array of doubles */
MPI_Datatype dbl_array;
...

MPI_Type_contiguous(SIZE, MPI_DOUBLE, &dbl_array);
MPI_Type_commit(&dbl_array);

if (rank == 0)
{
    /*
     * send 1 instance of the dbl_array type (vs. 10 of MPI_DOUBLE)
     */
    MPI_Send(vals, 1, dbl_array, 1, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Recv(vals, 1, dbl_array, 0, 0, MPI_COMM_WORLD, &status);
}
...
```

## Vector Type: MPI\_Type\_vector

```
int MPI_Type_vector
(
    int count,
    int length,
    int stride,
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- Sending non-adjacent data stored in contiguous memory with constant stride

HPC Resources

- length
  - number of entries in each element (i.e. can be an array)
- stride
  - number of elements of *type* between elements of *new*
- e.g. C stores matrices in row-major order --- the values in a column of the matrix are row-length elements apart

## Vector Type Example

```
...
double vals[SIZE][SIZE]; /* contiguous array of doubles */
MPI_Datatype dbl_column;
...
MPI_Type_vector(SIZE, 1, SIZE, MPI_DOUBLE, &dbl_column);
MPI_Type_commit(&dbl_column);

if (rank == 0)
{
    /* note we can send any column of a SIZExSIZE array of
     * doubles in this same way */
    MPI_Send(&vals[0][0], 1, dbl_column, 1, 0, MPI_COMM_WORLD);
}
else
{
    /* we'll receive the column sent into a different
     * column of the array in this process */
    MPI_Recv(&vals[0][3], 1, dbl_column, 0, 0, MPI_COMM_WORLD, &status);
}
...

```

## Indexed Type: MPI\_Type\_index

```
int MPI_Type_index
(
    int count,
    int lengths[],
    int offsets[],
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- Sending non-adjacent data stored in contiguous memory with variable stride

HPC Resources

- lengths
  - lengths[i] gives number of elements in the  $i^{\text{th}}$  entry of the new type
- offsets
  - offsets[i] gives number elements of *type* offset from the beginning of type for the  $i^{\text{th}}$  entry of the new type
- e.g. main diagonal of a matrix (since lengths can be variable we can send sub-matrices this way as well)

## Indexed Type Example

```
...
double vals[SIZE][SIZE]; /* contiguous array of doubles */
MPI_Datatype dbl_diag; /* main diagonal doubles from 2-D array */
int lengths[SIZE]; /* entry lengths */
int offsets[SIZE]; /* entry offsets */
...

for (i = 0; i < SIZE; i++)
{
    lengths[i] = 1;
    offsets[i] = (SIZE+1) * i;
}

MPI_Type_indexed(SIZE, lengths, offsets, MPI_DOUBLE, &dbl_diag);
MPI_Type_commit(&dbl_diag);

if (rank == 0)
    MPI_Send(vals[0], 1, dbl_diag, 1, 0, MPI_COMM_WORLD);
else
    MPI_Recv(vals[0], 1, dbl_diag, 0, 0, MPI_COMM_WORLD, &status);
...

```

## Structure Type: MPI\_Type\_struct

```
int MPI_Type_struct
(
    int count,
    int lengths[],
    MPI_Aint offsets[],
    MPI_Datatype types[],
    MPI_Datatype *new
);
```

- MPI allows us to combine arbitrary types into a structure type
  - i.e. create arbitrary type signatures

HPC Resources

- offsets
  - offsets[i] gives offset from the beginning of the type for the  $i^{\text{th}}$  component of the new type
    - note that these can be completely arbitrarily located in memory, all offsets are simply relative to the beginning of the type (i.e. the buffer provided to send/recv)
  - note: MPI\_Aint is a MPI defined type for addresses (allows for addresses too large to represent with an integer)

## Structure Type (cont.): MPI\_Address

```
int MPI_Address
(
    void *ref,
    MPI_Aint *address
);
```

- Offset values are expressed as addresses rather than counts
  - consecutive values are not necessarily the same type, and may not occur contiguously in storage (structure types are completely arbitrary)
  - in C we could just use the address-of operator (&); this function ensures portability

- ref
  - reference to storage containing value
- address
  - initialized with address of provided reference upon return

HPC Resources

## Structure Type Example

```
...
int val1 = 0; double val2 = 0.0; char val3[STR_LEN];

MPI_Datatype int_dbl_str; /* type containing int/double/string */
int lengths[3]; /* entry lengths */
MPI_Aint offsets[3]; /* entry offsets */
MPI_Datatype types[3]; /* entry types */
MPI_Aint base, off;
...

lengths[0] = lengths[1] = 1;
lengths[2] = STR_LEN;

/* all addresses relative to val1 (&val1 is the "buffer" on send) */
MPI_Address(&val1, &base);
offsets[0] = 0;
MPI_Address(&val2, &off);
offsets[1] = off - base;
MPI_Address(&val3, &off);
offsets[2] = off - base;
```

## Structure Type Example (cont.)

```
types[0] = MPI_INT;
types[1] = MPI_DOUBLE;
types[2] = MPI_CHAR;

MPI_Type_struct(3, lengths, offsets, types, &int_dbl_str);
MPI_Type_commit(&int_dbl_str);

if (rank == 0)
{
    val1 = 5;
    val2 = 3.14;
    strcpy(val3, "hello");

    MPI_Send(&val1, 1, int_dbl_str, 1, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Recv(&val1, 1, int_dbl_str, 0, 0, MPI_COMM_WORLD, &status);
}
...
```

## Notes on Derived Types

- Set-up is relatively expensive
  - only use when data being transmitted forms a regular part of communication
- Reasonably efficient once set-up is complete
  - permit more natural data representation in code
- CAUTION re: Structure types
  - offsets in structure types reflect actual data layout in memory
  - only reference addresses that will exist at the time of transmission

HPC Resources

2000	val1=5	← offsets[0] = 0
2004		
2008		
200C	val2=3.14	← offsets[1] = 12
2010		
2014		
2018		
201C		
2020	val3=2030	
2024		
2028		
202C		
2030	hell	← offsets[2] = 48
2034	o\0	

## Packing Data: MPI\_Pack

```
int MPI_Pack
(
    void *data,
    int count,
    MPI_Datatype datatype,
    void *buffer,
    int size,
    int *offset,
    MPI_Comm comm
);
```

- Packing allows us to explicitly store non-contiguous data in contiguous memory locations

HPC Resources

- data
  - reference to data to be packed
- buffer
  - buffer into which we are packing the data
- offset
  - offset into buffer where data should be packed
  - when the function returns *offset* has been changed to refer to the first location in *buffer* after the data that was just packed

## Unpacking Data : MPI\_Unpack

```
int MPI_Unpack
(
    void *buffer,
    int size,
    int *offset,
    void *data,
    int count,
    MPI_Datatype datatype,
    MPI_Comm comm
);
```

- Unpacking data is exactly the opposite process

HPC Resources

- buffer
  - buffer from which we are unpacking the data
- offset
  - references the starting position of the data to be unpacked within buffer
  - after return *offset* is changed to reference the first position in *buffer* after the data that was just unpacked
- data
  - reference to location into which data is to be unpacked

## Notes on Packing Data

- The type of contiguous data sent in this manner is MPI\_PACKED
- This is completely explicit data packing/unpacking
  - there is no part of this where you define a type that can be reused; you are simply jamming data into a contiguous representation for transmission
- There is significant ongoing overhead with packing data in this manner
  - if it must be done repetitively you are probably better off with a contiguous or structured derived type
  - if you are transmitting highly variable messages, it becomes an empirical question whether you'll see better performance from packing data as compared to numerous individual sends

HPC Resources

## Packing Data Example

```

...
int val1 = 0;
double val2 = 0.0;
char val3[STR_LEN];
int offset = 0;
char buf[BUF_SIZE];
...

if (rank == 0)
{
    val1 = 5; val2 = 3.14; strcpy(val3,"hello");
    /*
     * pack data to be sent into buffer
     */
    MPI_Pack(&val1,1,MPI_INT,buf,BUF_SIZE,&offset,MPI_COMM_WORLD);
    MPI_Pack(&val2, 1, MPI_DOUBLE,
            buf, BUF_SIZE, &offset, MPI_COMM_WORLD);
    MPI_Pack(&val3, STR_LEN, MPI_CHAR,
            buf, BUF_SIZE, &offset, MPI_COMM_WORLD)
}

```

## Packing Data Example (cont.)

```

/*
 * send the packed data buffer
 */
MPI_Send(buf, BUF_SIZE, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Recv(buf,BUF_SIZE,MPI_PACKED,0,0,MPI_COMM_WORLD,&status);

    /*
     * unpack received data from packed data buffer
     */
    MPI_Unpack(buf, BUF_SIZE, &offset,
              &val1, 1, MPI_INT, MPI_COMM_WORLD);
    MPI_Unpack(buf, BUF_SIZE, &offset,
              &val2, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buf, BUF_SIZE, &offset,
              &val3, STR_LEN, MPI_CHAR, MPI_COMM_WORLD);
}
...

```

## Data Summary

- If data to be sent is already stored in contiguous memory
  - use standard methods for transmission (or contiguous type depending on language)
  - use vector derived type if values are not adjacent, but are uniform in relative location
  - use indexed derived type if values are non adjacent and stride between values not a constant
- If data is not stored contiguously
  - use structure derived type if arbitrary storage locations are used repeatedly
  - use pack/unpack only where communication needs are highly non-uniform and there is measurable benefit from doing it

HPC Resources