MPI Data Types
Options for sending data

Sending blocks of data

Derived MPI_Datatype
  contiguous, vector, struct

Variable-content
  Pack and unpack
Sending blocks of data

All MPI calls can send blocks of data:

```c
MPI_Send(&data, count, MPI_INT, 1, tag, comm);
```

Requirements

All of the data must be of the same data type

The data must be contiguous in memory

*Type consistency* between sender and receiver

Pointer arithmetic works too!

You can use “address + count” to index into data

Useful for sending part of an array
MPI Datatypes

```c
MPI_Send(&data, count, MPI_INT, 1, tag, comm);
```
MPI Datatypes

\texttt{MPI\_Send}(&data, count, MPI\_INT, 1, tag, comm);

What do we need to know?
MPI Datatypes

**MPI_Send**(&data, count, MPI_INT, 1, tag, comm);

What do we need to know?

address, datatype, number of elements
MPI Datatypes

\texttt{MPI\_Send}(&data, count, MPI\_INT, 1, tag, comm);

What do we need to know?

address, datatype, number of elements

What does MPI do?
MPI Datatypes

```c
MPI_Send(&data, count, MPI_INT, 1, tag, comm);
```

What do we need to know?

address, datatype, number of elements

What does MPI do?

parses memory based on the starting address, size, and count
MPI Datatypes

What do we need to know?
address, datatype, number of elements

What does MPI do?
parses memory based on the starting address, size, and count
based on *contiguous* data

```c
MPI_Send(&data, count, MPI_INT, 1, tag, comm);
```
MPI Derived Data Types
Creating Custom Datatypes

What is a custom MPI datatype?

Construction

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_struct`

Commit

- `MPI_Type_commit`

Send and receive

Free

- `MPI_Type_free`
MPI Custom Datatypes
Where to the types start?

How many of each type?
What type of data?
MPI Custom Datatypes

Where to the types start?

disp[0]
disp[1]
disp[2]

How many of each type?
What type of data?
Where to the types start?

disp[0]  
disp[1]  
disp[2]

How many of each type?
What type of data?

count[0] = 1  
count[1] = 1  
count[2] = 2  
types[0] = MPI_INT  
types[1] = MPI_FLOAT  
types[2] = MPI_DOUBLE
MPI Datatype is a map

```c
struct {
    int num;
    float x;
    double data[2];
} obj;
```

```c
MPI_Datatype obj_type;
```

this is what we are learning today

```c
MPI_Bcast(&obj, 1, obj_type, 0, comm);
```
MPI Datatype is a map

```c
struct {
    int num;
    float x;
    double data[2];
} obj;
```

```c
MPI_Datatype obj_type;
```

```c
MPI_Bcast(&obj, 1, obj_type, 0, comm);
```
MPI Datatype is a map

```c
struct {
    int num;
    float x;
    double data[2];
} obj;

MPI_Datatype obj_type;

MPI_Bcast(&obj, 1, obj_type, 0, comm);
```

Memory address

This is what we are learning today.
MPI Datatype is a map

```c
struct {
    int num;
    float x;
    double data[2];
} obj;

MPI_Datatype obj_type;

MPI_Bcast(&obj, 1, obj_type, 0, comm);
```

<table>
<thead>
<tr>
<th></th>
<th>disp</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INT</td>
<td>address</td>
<td>1</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>address</td>
<td>1</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>address</td>
<td>2</td>
</tr>
</tbody>
</table>

this is what we are learning today

memory address

this is how you interpret the data
Contiguous

```c
int MPI_Type_contiguous(int count,
                        MPI_Datatype old_type,
                        MPI_Datatype new_type);
```

Example

```c
int B[2][3];
MPI_Datatype matrix;
MPI_Type_contiguous(6, MPI_INT, &matrix);
```

<table>
<thead>
<tr>
<th>datatype</th>
<th>displacement</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INT</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
Contiguous

```c
int MPI_Type_contiguous(int count,
                        MPI_Datatype old_type,
                        MPI_Datatype new_type);
```

Example

```c
int B[2][3];
MPI_Datatype matrix;
MPI_Type_contiguous(6, MPI_INT, &matrix);
```

<table>
<thead>
<tr>
<th>datatype</th>
<th>displacement</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INT</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

What’s the advantage over this?

```c
MPI_Send(B, 6, MPI_INT, 1, tag, comm);
```
const int N = 10;

double A[N][N];
double B[N][N];

MPI_Datatype matrix;

MPI_Type_contiguous(N*N, MPI_DOUBLE, &matrix);
MPI_Type_commit(&matrix);

if (rank == master_rank)
    MPI_Send(A, 1, matrix, 1, 10, comm);
else if (rank == 1)
    MPI_Recv(B, 1, matrix, 0, 10, comm, &status);
Vector

User specifies memory locations

```c
int MPI_Type_vector(int count,
                    int blocklength,
                    int stride,
                    MPI_Datatype old_type,
                    MPI_Datatype *newtype);
```

`newtype` has

- `count` blocks each consisting of `blocklength` copies of `oldtype`.

Displacement between blocks is set by `stride`. 
Example

```cpp
const int N = 5;
MPI_Datatype dt;
MPI_Type_vector(N, 1, N, MPI_INT, &dt);
```

Variables

- `count = 5`
- `blocklength = 1`
- `stride = 5`
Example

```c
const int N = 5;
MPI_Datatype dt;
MPI_Type_vector(N, 1, N, MPI_INT, &dt);
```

Variables

- count = 5
- blocklength = 1
- stride = 5
Example

```c
const int N = 5;
MPI_Datatype dt;
MPI_Type_vector(N, 1, N, MPI_INT, &dt);
```

Variables

- count  = 5
- blocklength = 1
- stride = 5

How is this useful?
Matrix Column Example

```c
const int N = 10;

double A[N][N];
MPI_Datatype column;

MPI_Type_vector(N, 1, N, MPI_DOUBLE, &column);
MPI_Type_commit(&column);

if (rank == master_rank)
    MPI_Send(&A[0][2], 1, column, 1, 10, comm);
else if (rank == 1)
    MPI_Recv(&A[0][2], 1, column, 0, 10, comm, &status);
```
Extent

Memory span of a datatype

```c
int MPI_Type_extent(MPI_Datatype datatype,
                     MPI_Aint *extent);
```

Usage: think “size of”

```c
MPI_Aint intex;

MPI_Type_extent(MPI_INT, &intex);

displacements[0] = static_cast<MPI_Aint>(0);
displacements[1] = intex;
```
Structure

Heterogeneous

Most general derived datatype

```c
int MPI_Type_struct(int count,
                            int blocks[],
                          MPI_Aint displacements[],
                          MPI_Datatype types[],
                          MPI_Datatype *newtype);
```

newtype consists of

- count blocks where the $i$th block is `$blocks[i]$` copies of the type `$types[i]$`

- The displacement of the $i$th block (in bytes) is given by `$displacements[i]$`
Example

```c
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

count =
blocks =
types =

displacements =

<table>
<thead>
<tr>
<th>int</th>
<th>float</th>
<th>double</th>
<th>double</th>
<th>double</th>
<th>double</th>
<th>double</th>
<th>double</th>
</tr>
</thead>
</table>

Thursday, November 8, 12
Example

```c
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

- `count` blocks where the *ith* block is `blocks[i]` copies of the type `types[i]`
- The displacement of the *ith* block (in bytes) is given by `displacements[i]`

Variables:

- `count =` `int`
- `blocks =` `float`
- `types =` `double`
- `displacements =` `double`
Example

```c
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

- **count** = 3
- **blocks** = where the \(i^{th}\) block is
  \(\text{blocks}[i]\) copies of the type \(\text{types}[i]\)
- **types** =
- **displacements** =

```
int float double double double double double
```
Example

```c
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

- **count = 3**
- **blocks = \{ 1, 1, 4 \}**
- **types = \{ int, float, double \}**
- **displacements = \{ 0, size(int), size(int) + size(float) \}**

- count blocks where the *ith* block is \( \text{blocks}[i] \) copies of the type \( \text{types}[i] \)
- The displacement of the *ith* block (in bytes) is given by \( \text{displacements}[i] \)

| int | float | double | double | double | double | double | double |
Example

```
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

- `count` blocks where the `ith` block is `blocks[i]` copies of the type `types[i]`
- The displacement of the `ith` block (in bytes) is given by `displacements[i]`

```
count = 3
blocks = { 1, 1, 4 }
types = { int, float, double }
displacements =
```

- `int`
- `float`
- `double`
- `double`
- `double`
- `double`
- `double`
Example

```c
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Variables

- `count` blocks where the *ith* block is
  - `blocks[i]` copies of the type `types[i]`
- The displacement of the *ith* block (in bytes) is given by `displacements[i]`

```plaintext
count = 3
blocks = { 1, 1, 4 }
types = { int, float, double }
displacements = { 0, size(int), size(int) + size(float) }
```

```
| int  | float | double | double | double | double | double |
```
Example

```c
int blocks[3]={1,1,4};
MPI_Datatype types[3]={MPI_INT, MPI_FLOAT, MPI_DOUBLE};
MPI_Aint displacements[3];

MPI_Datatype obj_type;
MPI_Aint intex, floatex;

MPI_Type_extent(MPI_INT, &intex);
MPI_Type_extent(MPI_FLOAT, &floatex);
displacements[0] = static_cast<MPI_Aint>(0);
displacements[1] = intex;
displacements[2] = intex+floatex;
MPI_Type_struct(3, blocks, displacements, types, &obj_type);
```
C/C++

```c
struct {
    int n;
    double x[3];
} obj;
```

<table>
<thead>
<tr>
<th>int</th>
<th>double</th>
<th>double</th>
<th>double</th>
</tr>
</thead>
</table>
C/C++

```c
struct {
    int n;
    double x[3];
} obj;
```

```c
struct {
    int n;
    double * x;
} obj;
```
C/C++

```c
struct {
    int n;
    double x[3];
} obj;
```

```c
struct {
    int n;
    double * x;
} obj;
```
C/C++

```c
struct {
    int n;
    double x[3];
} obj;
```

```c
struct {
    int n;
    double * x;
} obj;
```

```
int  double  double  double
```

```
int  ptr
     →  double  double  double
```

```
←  displacement[1]  →
```
int MPI_Address(void *location, MPI_Aint *address);

Gets the address location in memory

struct {
    int n;
    double * x;
} obj;
int MPI_Address(void *location, MPI_Aint *address);

Gets the address location in memory

struct {
    int n;
    double * x;
} obj;

obj.n = 10;
obj.x = new double[obj.n];
MPI_Address

    int MPI_Address(void *location, MPI_Aint *address);

Gets the address location in memory

struct {
    int n;
    double * x;
} obj;

obj.n = 10;
obj.x = new double[obj.n];

MPI_Aint address_n, address_x;
MPI_Address(&obj.n, &address_n);
MPI_Address(obj.x, &address_x);
MPI_Address

    int MPI_Address(void *location, MPI_Aint *address);

Gets the address location in memory

struct {
    int n;
    double * x;
} obj;

obj.n = 10;
obj.x = new double[obj.n];

MPI_Aint address_n, address_x;
MPI_Address(&(obj.n), &address_n);
MPI_Address(obj.x, &address_x);

MPI_Aint displacements[2];
displacements[0] = static_cast<MPI_Aint>(0);
```c
int MPI_Address(void *location, MPI_Aint *address);

Gets the address location in memory

struct {
    int n;
    double * x;
} obj;

obj.n = 10;
obj.x = new double[obj.n];

MPI_Aint address_n, address_x;
MPI_Address(&(obj.n), &address_n);
MPI_Address(obj.x, &address_x);

MPI_Aint displacements[2];
displacements[0] = static_cast<MPI_Aint>(0);
displacements[1] = address_x - address_n;
```
Commit and Free

After construction, you must commit the datatype

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

When you are done, you need to free the datatype

```c
int MPI_Type_free(MPI_Datatype *datatype);
```
Send are Receive

```c
if (rank==3) {
    obj.num=6;
    obj.x=3.14;
    for(int i=0;i<4;++i)
        obj.data[i]=(double) i;

    MPI_Send(&obj, 1, obj_type, 1, 52, comm);
}
else if(rank==1) {
    MPI_Recv(&obj, 1, obj_type, 3, 52, comm,&status);
}
```
if (rank == master_rank) {
    for(int i=0; i<obj.n; ++i)
        obj.x[i] = i+1;
}

MPI_Bcast(&obj, 1, obj_type, 0, comm);
Pack and Unpack
Alternative approach to grouping data for communication

Manually pack variables into a contiguous buffer

Transmit

Unpack into desired variables

Paradigm: just like a stream
Considerations

Good idea:

- May avoid the use of system buffering
- Ideal for sending variable length messages
- Sparse matrix data, client/server task dispatch

Bad idea:

- You have to pack and unpack data yourself
- Requires a lot of attention to detail

Do you really want variable length messages?

- You still need big pre-allocated fixed-size buffers