Research Computing

UNIVERSITY OF COLORADO

MPI Datatypes

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`

Point to point communication

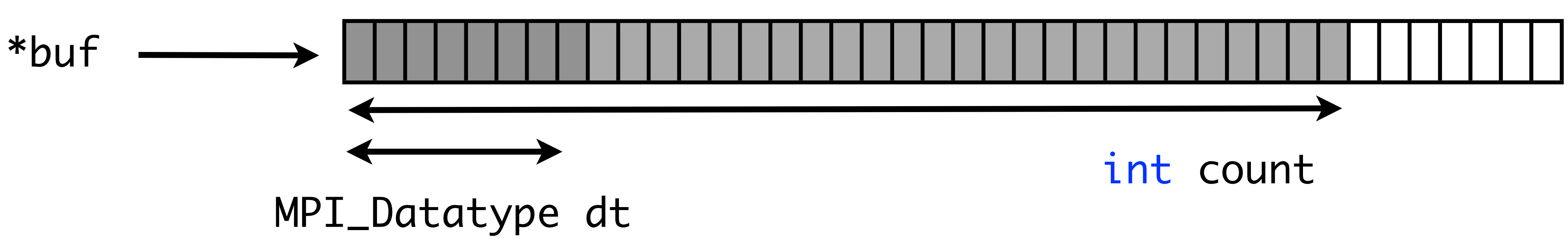
```
int MPI_Send( void *buf,
              int count,
              MPI_Datatype dt,
              int dest,
              int tag,
              MPI_Comm comm );
```

```
int MPI_Recv( void *buf,
              int count,
              MPI_Datatype dt,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Status *status );
```

```
int source

              int dest
              int tag
              MPI_Comm comm
```

envelope

message body



Communication review

Consider the following code:

```
int data = 10;
if (rank == 0)
    MPI_Send(&data,1,MPI_INT,1,tag,comm);
else
{
    MPI_Recv(&data,1,MPI_INT,0,tag,comm,&status);
    std::cout << "Message received!" << std::endl;
}
```

envelope

message body

If this code is run on 1 processor, what do you expect to happen?

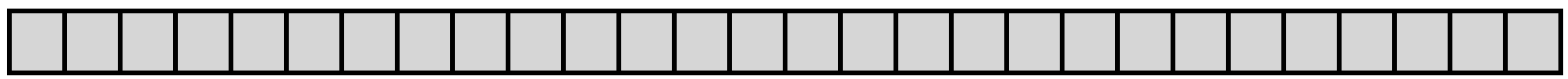
error condition

If the code is run on 3 processors, what do you expect?

code will hang

MPI Datatypes

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



What do we need to know?

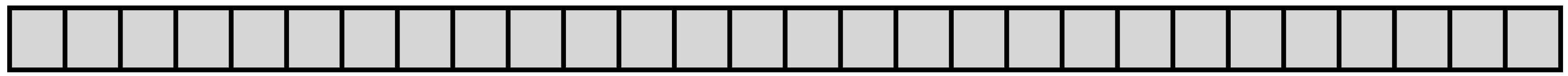
address, datatype, number of elements

What does MPI do?

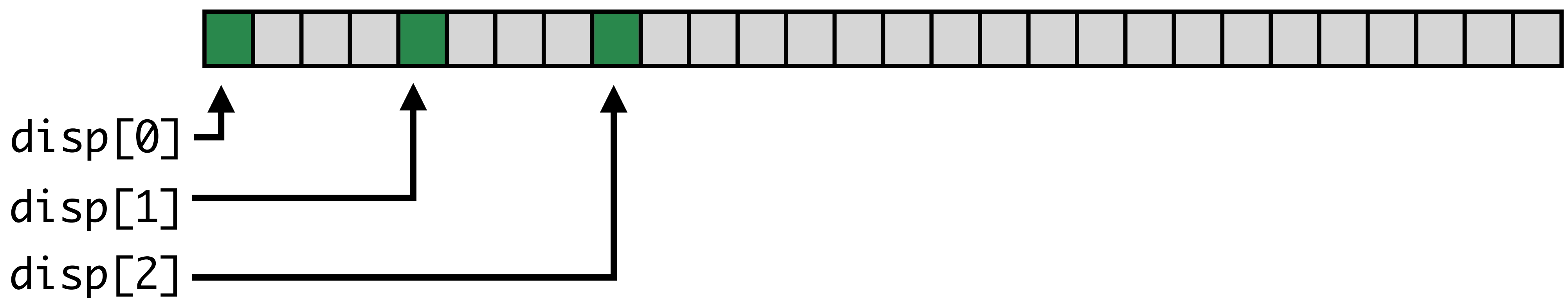
parses memory based on the starting address, size, and count

based on *contiguous* data

MPI Custom Datatypes

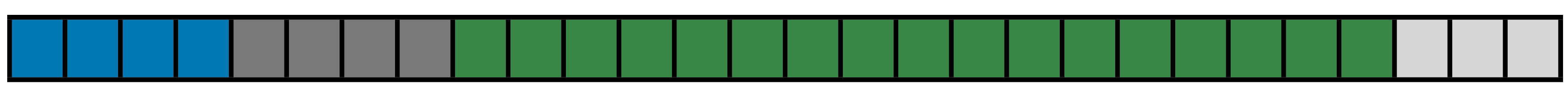


Where to the types start?



How many of each type?

What type of data?



`count[0] = 1`
`types[0] = MPI_INT`

`count[1] = 1`
`types[1] = MPI_FLOAT`

`count[2] = 2`
`types[2] = MPI_DOUBLE`

MPI Datatype is a map

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

int	float	double	double

```c
MPI_Datatype obj_type;
```

type	disp	count
MPI_INT	address	1
MPI_FLOAT	address	1
MPI_DOUBLE	address	2

this is what we are learning today

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

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);
```

datatype	displacement	count
MPI_INT	0	6

What’s the advantage over this?

```c
MPI_Send(B, 6, MPI_INT, 1, tag, comm);
```

Contiguous Example

```
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

```
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

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

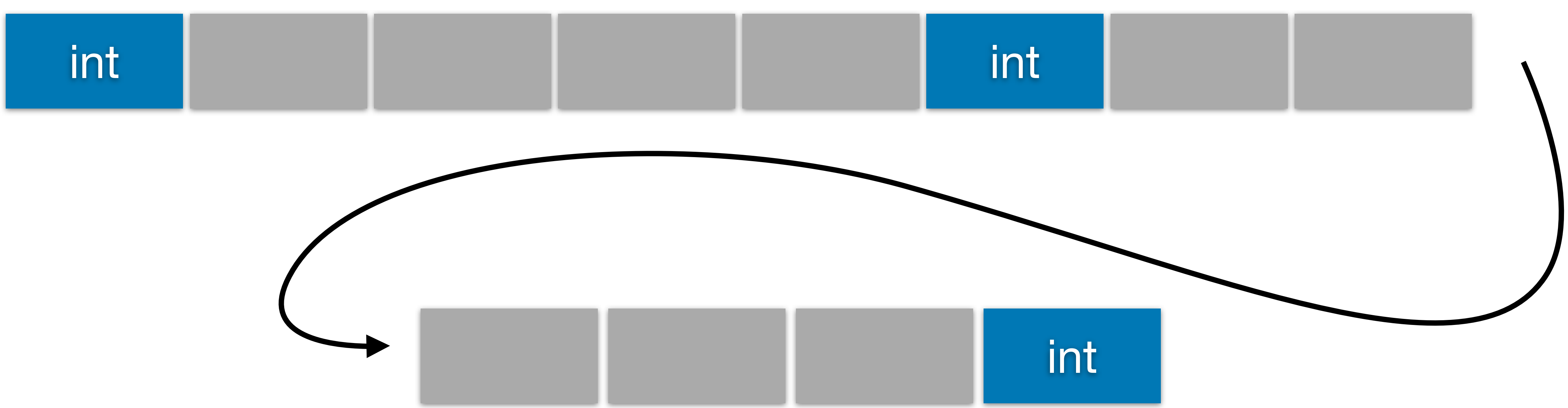
Variables

count = 5

blocklength = 1

stride = 5

How is this useful?



Matrix Column Example

```
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

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

Usage: think “size of”

```
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

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

`newtype` consists of

- `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]`

Example

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

Variables

count =

blocks =

types =

displacements =

`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]`

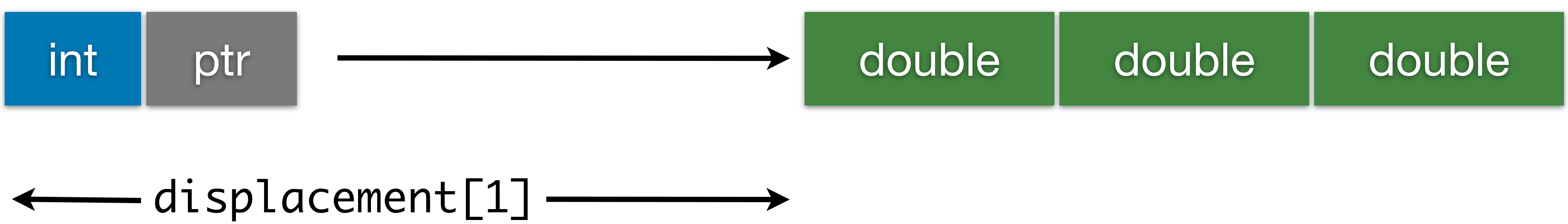
Example

```
   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++

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


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



MPI_Address

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

Gets the address location in memory

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

Commit and Free

After construction, you must commit the datatype

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

When you are done, you need to free the datatype

```
int MPI_Type_free(MPI_Datatype *datatype);
```

Send are Receive

```
  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);

}
```

Broadcast

```
   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);
```