

MPI Parallel Programming

Part II

P.Y. Wang
Department of Computer Science 4A5
George Mason University
Fairfax VA 22030-4444 U.S.A.

Group Communications

MPI datatypes have two main purposes:

- to address heterogeneity in parallel programs – different processors
- to facilitate communications involving
 - noncontiguous data,
 - structures,
 - vectors with strides

MPI has several data types

- Elementary: `MPI_INT`, `MPI_FLOAT`, etc
- Struct: general mixed types (for C structs, etc)
- Contiguous: vector with stride of 1
- Vector: separated by a constant stride
- Indexed: array of indices (for scatter/gather)
- Hvector: vector whose stride is in bytes
- Hindexed: indexed with indices in bytes

Structs

- In a previous example, we used three communications to send three values from one process to another:

```
MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD):  
MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD):  
MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD):
```

- Note that three different memory address are passed to the “Send” routine
- This can be inefficient, especially if it is executed many times since the amount of data being send is very small
- One solution: create a struct containing the desired data
- **How do we get MPI to recognize this struct?**

Building a Struct Datatype

- Specify the layout of the data in the struct
 - specify the member types
 - specify their relative locations in memory

- That is use:

```
int MPI_Type_Struct ( int count,
                    int *array_of_block_lengths,
                    MPI_Aint *array_of_displacements,
                    MPI_Datatype *array_of_types,
                    MPI_Datatype *newtype);
```

Example

```
#include <stdio.h>
#include "mpi.h"
/* SPMD program sending struct between processes */
int main(argc, argv)
int argc; char *argv[];
{
    int me;          /* Rank of process */
    int size;       /* Number of processes */
    struct {        /* For local data */
        float a,b;
        int n;
    } x,y;

    MPI_Status status; /* Return status for receive */
    MPI_Datatype my_type; /* Creating a new data type */

    int block_size[3]; /* Size of each block in the struct */
    MPI_Aint addr[4]; /* Declare array of type MPI Address int */
    MPI_Aint offset[3]; /* Declare array of type MPI Address int */
    MPI_Datatype type[3];

    MPI_Init(&argc, &argv); /* usual startup info */
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &size);

x.a = 10.0 * (float) me; /* local data */
x.b = 20.0 * (float) me;
x.n = me;
y.a = y.b = -1.0; y.n = -1;

printf("Me: %d has a x struct %f %f %d\n",me, x.a, x.b, x.n);
printf("          y struct %f %f %d\n", y.a, y.b, y.n);
```

```
type[0] = MPI_FLOAT; /* specify the types */
type[1] = MPI_FLOAT;
type[2] = MPI_INT;

block_size[0] = 1; /* specify the number of items of each type */
block_size[1] = 1;
block_size[2] = 1;

MPI_Address(&x, &addr[0]); /* find the offsets */
MPI_Address(&x.a, &addr[1]);
MPI_Address(&x.b, &addr[2]);
MPI_Address(&x.n, &addr[3]);

offset[0] = addr[1] - addr[0];
offset[1] = addr[2] - addr[0];
offset[2] = addr[3] - addr[0];

/* Create and commit the derived data type */

MPI_Type_struct(3,block_size,offset,type,&my_type);
MPI_Type_commit(&my_type);
```

```
if (me == 0) {  
  
    /* Process 0 looking for struct message from process 1 */  
  
    MPI_Recv(&y, 1, my_type, 1, 10, MPI_COMM_WORLD, &status);  
    printf("    Data received from process 1:\n");  
    printf("    y.a: %f y.b: %f y.n: %d\n", y.a, y.b, y.n);  
  
} else {  
  
    /* Process 1 sending its x struct to process 0 */  
    if (me == 1) {  
        MPI_Send(&x, 1, my_type, 0, 10, MPI_COMM_WORLD);  
    }  
}  
  
MPI_Finalize();  
}
```

Other Derived Datatypes

Create a derived datatype consisting of count elements of type oldtype:

```
int MPI_Type_Contiguous ( int count,  
                          MPI_Datatype oldtype,  
                          MPI_Datatype *newtype);
```

```
int MPI_Type_vector ( int count,
                    int block_length,
                    int stride,
                    MPI_Datatype element_type,
                    MPI_Datatype *newtype);
```

- Create a derived datatype of `count` elements;
- Each element contains `block_length` entries of type `element_type`;
- `stride` is the number of elements of type `element_type` between successive elements of `new_type`

Vector Derived Datatype

- For a two-dimensional matrix of `int` values with `R` rows and `C` columns declared within a **C** program (i.e. **row-major** storage)
- To set up communication for one entire column:

```
int R, C;
MPI_Datatype newtype;

MPI_Type_vector (R, 1, C, MPI_INT, &newtype);
MPI_Type_commit (&newtype);

;
```

- Note: the count is `R`, the `block_length` is 1, and the stride is `C`, so we create a **column** datatype

Another Example

To send the **third row** of a two-dimensional C array (row-major) from process 0 to process 1

```
float a[10][10];
if (my_rank == 0)
    MPI_Send(&a[2][0], 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else /* assuming my_rank == 1 here */
    MPI_Recv(&a[2][0], 10, MPI_FLOAT, 0, 0, &status);
```

But what if we want to send the **third column**?

```
/* Define my_column_type which is of type MPI_Datatype */
MPI_Type_vector(10, 1, 10, MPI_FLOAT, &my_column_type);
MPI_Type_commit(&my_column_type);

if (my_rank == 0)
    MPI_Send(&a[0][2], 1, my_column_type, 1, 0, MPI_COMM_WORLD);
else /* assuming my_rank == 1 here */
    MPI_Recv(&a[0][2], 1, my_column_type, 0, 0, MPI_COMM_WORLD,
            &status);
```

```
int MPI_Type_indexed ( int count,
                      int *array_of_block_length,
                      int *array_of_displacements,
                      MPI_Datatype element_type,
                      MPI_Datatype *newtype);
```

- Creates a derived type consisting of `count` elements
- The i^{th} element ($i = 0, \dots, \text{count} - 1$) consists of `array_of_block_lengths[i]` entries of type `element_type`
- and is displaced `array_of_displacements[i]` units of type `element_type` from the beginning of `new_type`.

MPI_Type_Indexed Example

To send the **upper triangular** portion of a square matrix stored on process 0 to process 1

```
float A[n][n];          /* Complete Matrix */
float T[n][n];          /* Upper Triangular */
int displacements[n];
int block_lengths[n];
MPI_Datatype my_indexed_type;

for (i = 0; i < n; i++) {
    block_lengths[i] = n - i; displacements[i] = n * i; }

MPI_Type_indexed(n, block_lengths, displacements,
                 MPI_FLOAT, &my_indexed_type);
MPI_Type_commit(&my_indexed_type);

if (my_rank == 0)
    MPI_Send(A, 1, my_indexed_type, 1, 0, MPI_COMM_WORLD);
else /* assuming my_rank == 1 */
    MPI_Recv(T, 1, my_indexed_type, 0, 0, MPI_COMM_WORLD, &status);
```

More on Structs: Setting up Extents

- The **extent** of a datatype is normally the distance between the first and last member
- You can set up an artificial extent by using **MPI_UB** and **MPI_LB** in **MPI_Type_struct**
- When sending an array of a structure, it is important to ensure that MPI and the C compiler have the same value for the size of each structure; we can use **MPI_UB** for this

```
struct {                                /* For a C struct */
    char d[50];
    int n;
    double xmin, ymin;
    double xmax, ymax;
    int width;
    int height
} my_stats;

int block_size[5] = {50,1,4,2,1};      /* For MPI datatype */
MPI_Datatype type[5];
MPI_Aint offset[5];
MPI_Datatype stats_type;
```

```
MPI_Address (&my_stats.d, &offset[0]);
MPI_Address (&my_stats.n, &offset[1]);
MPI_Address (&my_stats.xmin, &offset[2]);
MPI_Address (&my_stats.width, &offset[3]);
MPI_Address (&my_stats+1, &offset[4]);

type[0] = MPI_CHAR;      type[1] = MPI_INT;
type[2] = MPI_DOUBLE;   type[3] = MPI_INT;
type[4] = MPI_UB;

offset[4] -= offset[0];  offset[3] -= offset[0];
offset[2] -= offset[0];  offset[1] -= offset[0];
offset[0] -= offset[0];

MPI_Type_struct (5, block_size, offset, type, &stats_type);
MPI_Type_commit (&stats_type);
```

Interleaving Data

- Suppose you have matrix stored in column-major format and you want to send different parts to different processes:

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

- Send 0–3, 8–11, 16–19, 24–27 to process 0
- Send 4–7, 12–15, 20–23, 28–31 to process 1
- Send 32–35, etc to process 2; send 36–39, etc to process 3

A Solution?

- Define block to be a MPI struct datatype whose extent is just 1 entry:

```

/* define a matrix block in stages */
MPI_Type_vector (4, 4, 8, MPI_DOUBLE, &my_vector);

/* define a derived struct */
block_size[0] = 1;   block_size[1] = 1;
offset[0] = 0;      offset[1] = sizeof(double);
type[0] = my_vector; type[1] = MPI_UB;

MPI_Type_struct (2, block_size, offset, type, &block);

```

- Now set the displacements for each block as the location of the first element in the block
- Then use `MPI_Scatterv` (i.e. `MPI_Scatter` with different size chunks)

```
scatter_offset[0] = 0;      scatter_offset[1] = 4;  
scatter_offset[2] = 32;   scatter_offset[3] = 36;  
sendcounts[0] = 1;       sendcounts[1] = 1;  
sendcounts[2] = 1;       sendcounts[3] = 1;
```

```
MPI_Scatterv ( sendbuf, sendcounts, scatter_offset, block,  
              recvbuf, nx * ny, MPI_DOUBLE, 0, MPI_WORLD_COMM);
```

- Note: $nx * ny = 16$ in this example
- This works because `MPI_Scatterv` uses the extents to determine the start of each piece to send

Please see the Pachero textbook (Chapter 6) for more examples of derived datatypes and matrix message passing!

An Alternative to Derived Datatypes

- You can send noncontiguous data by explicitly selecting the data and *packing* it into a locally declared memory space
- After receiving such data, you will need to *unpack* it into a locally declared memory space
- This may be easier to understand, but is often less efficient (high run-time overhead)
- This approach is better **if** you use it infrequently or you need to repeatedly send different amounts of the same data each time

- To determine the space needed for a local memory space:

```
int MPI_Pack_size ( int in_count, MPI_Datatype dtype,  
                  MPI_Comm comm, int *size);
```
- To pack and unpack data:

```
int MPI_Pack ( void *inbuf, int in_count, MPI_Datatype dtype,  
              void *outbuf, int out_size, int *position,  
              MPI_Comm comm);  
  
int MPI_Unpack ( void *inbuf, int in_size, int *position,  
                void *outbuf, int out_count, MPI_Datatype dtype,  
                MPI_Comm comm);
```

An Example

```

float x[HUGE][4];      /* a matrix of some kind */
int   y[HUGE];        /* a vector of some kind */
/* For receiving */
int     src;
int     message_size;
MPI_Status status;
/* For sending */
unsigned int  membersize, maxsize;
int          position;
int          nhosts;
int          dest, tag;
char         *buffer;
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize = membersize;
MPI_Pack_size(HUGE*4, MPI_FLOAT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;
MPI_Pack_size(HUGE, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;

```

```

/* Pack up the desired message */

nhosts = 30;      /* pick something less than HUGE */
position = 0;

MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

MPI_Pack(x, nhosts*4, MPI_FLOAT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

MPI_Pack(y, nhosts, MPI_INT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

dest = 1; tag = 100;
MPI_Send (buffer, position, MPI_PACKED, dest, tag, MPI_COMM_WORLD);

. . .

```

```
nhosts = 4000;      /* pick something less than HUGE */
position = 0;

MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

MPI_Pack(x, nhosts*4, MPI_FLOAT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

MPI_Pack(y, nhosts, MPI_INT, buffer, maxsize, &position,
        MPI_COMM_WORLD);

dest = 1 ; tag = 200;
MPI_Send (buffer, position, MPI_PACKED, dest, tag, MPI_COMM_WORLD);

. . .
```

```
/* Receiving the packed message */

MPI_Recv (buffer, maxsize, MPI_PACKED, src, tag,
        MPI_WORLD_COMM, &status);

/* Unpack the message into local variables */

position = 0;
MPI_Get_count (&status, MPI_PACKED, &message_size);

MPI_Unpack(buffer, message_size, &position, &nhosts, 1, MPI_INT,
        MPI_COMM_WORLD);
MPI_Unpack(buffer, message_size, &position, &x, nhosts*4, MPI_FLOAT,
        MPI_COMM_WORLD);
MPI_Unpack(buffer, message_size, &position, &y, nhosts, MPI_INT,
        MPI_COMM_WORLD);
```

More MPI to come, but first a little something on matrix operations . . .

In the meantime, try completing the example that performs a block decomposition of an input matrix into p parts where each process gets a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix of an $n \times n$ array.