Derived Data Types in MPI

Dilip Angom

angom@prl.ernet.in

Theoretical Physics and Complex Systems
Physical Research Laboratory

Point-Point Communications

Point to point communication functions.

```
int MPI_Send(
        void*          buffer        /* in */,
        int            count         /* in */,
        MPI_Datatype   datatype      /* in */,
        int            destination   /* in */,
        int            tag           /* in */,
        MPI_Comm       Communicator  /* in */ )

int MPI_Recv(
        void*          buffer        /* in */,
        int            count         /* in */,
        MPI_Datatype   datatype      /* in */,
        int            source        /* in */,
        int            tag           /* in */,
        MPI_Comm       Communicator  /* in */,
        MPI_Status*    status        /* in */ )
```

One-All Communications

One-all collective communication functions.

```
int MPI_Bcast(
    void*         message   /* in/out */,
    int           count     /* in     */,
    MPI_Datatype  datatype  /* in     */,
    int           root      /* in     */,
    MPI_Comm      Comm      /* in     */ )

int MPI_Scatter(
    void*         send_data   /* in  */,
    int           send_count  /* in  */,
    MPI_Datatype  send_type   /* in  */,
    void*         recv_data   /* out */,
    int           recv_count  /* in  */,
    MPI_Datatype  recv_type   /* in  */,
    int           root        /* in  */,
    MPI_Comm      Comm        /* in  */ )
```

All-One Communication Functions

All-one collective communication functions.

```
int MPI_Reduce(
    void*          operand      /* in  */,
    void*          result       /* out */,
    int            count        /* in  */,
    MPI_Datatype datatype     /* in  */,
    MPI_Op         operator     /* in  */,
    int            source       /* in  */,
    int            root         /* in  */,
    MPI_Comm       Communicator /* in  */)

int MPI_Gather(
    void*          send_data  /* in  */,
    int            send_count /* in  */,
    MPI_Datatype send_type  /* in  */,
    void*          recv_data  /* out */,
    int            recv_count /* in  */,
    MPI_Datatype recv_type  /* in  */,
    int            root       /* in  */,
    MPI_Comm       Comm       /* in  */ )
```

All-All Communication Functions

All-all collective communication functions.

```
int MPI_Allgather(
        void*           send_data   /* in  */,
        int             send_count  /* in  */,
        MPI_Datatype    send_type   /* in  */,
        void*           recv_data   /* out */,
        int             recv_count  /* in  */,
        MPI_Datatype    recv_type   /* in  */,
        int             root        /* in  */,
        MPI_Comm        Comm        /* in  */ )
```

What next?

Grouping Data

MPI Datatypes

MPI datatype	*C datatype*
`MPI_CHAR`	signed char
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double

MPI Derived Datatypes

Other than the predefined `MPI datatypes`, it is possible to define new datatypes by grouping. This class of data is the derived datatype.

Derived datatypes in MPI can be used in

- Grouping data of different datatypes for communication.
- Grouping non contiguous data for communication.

MPI has the following functions to group data

`MPI_Type_contiguous`	`MPI_Type_struct`
`MPI_Type_vector`	`MPI_Pack`
`MPI_Type_indexed`	`MPI_Unpack`

Why Group Data

In general, each element of a system of interest has attributes of different datatypes. It is desirable to group these attributes to streamline manipulation and access.

Why Group Data

In general, each element of a system of interest has attributes of different datatypes. It is desirable to group these attributes to streamline manipulation and access.

1. Classical many-body system
   Each particle has the following attributes

Mass (m)	`MPI_DOUBLE` (1)
Position ($\vec{r}$)	`MPI_DOUBLE` (3)
Momentum ($\vec{p}$)	`MPI_DOUBLE` (3)
ID tag	`MPI_INT` (1)

2. Atomic systems
   Each electronic states has the following attributes

Energy (ϵ)	`MPI_DOUBLE` (1)
Principal quantum no. (n)	`MPI_INT` (1)
Orbital ang mom quantum no.(l)	`MPI_INT` (2)
Spin ang mom quantum no.(s)	`MPI_INT` (2)

Why Group Data

In general, each element of a system of interest has attributes of different datatypes. It is desirable to group these attributes to streamline manipulation and access.

Data grouping allows transfer of different datatypes in one MPI communication function call. Otherwise, one call per one datatype is required.

Data grouping allows transfer of non contiguous data in one MPI communication function call.

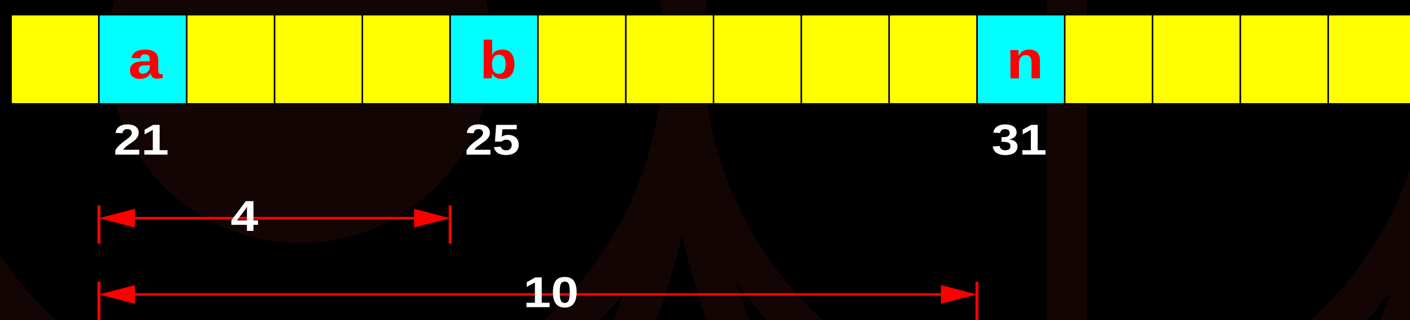
Each MPI function call is expensive as it involves several steps to initiate and ensure data communication is completed successfully. Data grouping can reduce the number of communication calls.

Building Derived Datatype

Suppose the following three data elements are defined in the main program.

```
float a, b;
int       n;
```

Schematically, the locations of these data in the memory can be represented as (each cell represents one memory location)



To send a, b and n in a single message, the following information is required

1. Number of elements.
2. List of the datatypes.
3. Relative memory locations.
4. Message beginning address.

Building Derived Datatype (contd)

The MPI function `MPI_Address` returns the address of a pointer. This can be used to find out the memory address of the message beginning and the relative locations of the data elements.

MPI derived datatype having n elements is a sequence of pairs

$$\{(t_0, d_0), (t_1, d_1), (t_2, d_2), \ldots (t_{n-1}, d_{n-1})\}$$

where t_i is the MPI datatype and d_i is the displacement in bytes.

The derived datatype to send a, b and n in a single message is

{(MPI_FLOAT, 0), (MPI_FLOAT, 4), (MPI_INT, 10)}

The final step of constructing a derived datatype is to commit it using the MPI function `MPI_Type_commit`. This is the mechanism to make internal changes that may improve communication performance.

MPI_Type_struct

The data elements a, b and n discussed earlier can be grouped using `MPI_Type_struct` in the following steps

1. Length of each element (`int block_lengths[3]`)

   ```
   block_lengths[0]=1;
   block_lengths[1]=1;
   block_lengths[2]=1;
   ```

2. Type of each element (`MPI_Datatype typelist[3]`)

   ```
   typelist[0]=MPI_FLOAT;
   typelist[1]=MPI_FLOAT;
   typelist[2]=MPI_INT;
   ```

3. Address of first element (`MPI_Aint start_add`)

   ```
   MPI_Adress(&a, &start_add);
   ```

MPI_Type_struct (contd)

4. Relative locations (`MPI_Aint relloc[3]`)

```
relloc[0] = 0;
MPI_Adress(&b, &address);
relloc[1] = address - start_add;
MPI_Adress(&n, &address);
relloc[2] = address - start_add;
```

5. Build the derived datatype (`MPI_Datatype* mesg_mpi_strct`)

```
MPI_Type_struct(3, block_lengths,
        relloc, typelist, mesg_mpi_strt);
```

6. Commit it

```
MPI_Type_commit(mesg_mpi_strct);
```

A Few Observations

The calling sequence of `MPI_Type_struct` is

```
int MPI_Type_struct (
            int               count,
            int               block_lengths[],
            MPI_Aint          relloc[],
            MPI_Datatype      typelist[],
            MPI_Datatype*     mesg_mpi)
```

Count is the number of elements in the derived type. In the example we considered it is three, two `MPI_FLOAT` (a and b) and one `MPI_INT` (n).

Block lengths are the number of the entries in each element and `relloc` refer to the relative location of each element from the beginning of the message.

A Few Observations (contd)

Datatype of relative location `relloc` is `MPI_Aint` and not `int`.

1. Addresses in C are integer longer than `int`.
2. Displacements, which are differences of two addresses can be longer than `int`. Datatype `MPI_Aint` takes care of this possibility.
3. FORTRAN has `integer` which is four bytes long, hence it is not necessary to use `MPI_Aint`.

Datatype of the entries in the `typelist` and `mesg_mpi` are the same, so `MPI_Type_struct` can be called recursively to construct complex datatypes.

Among all the MPI derived datatype constructors, the `MPI_Type_struct` is the most general. It allows grouping of different datatypes.

Grouping Data of Same Datatype

MPI has three functions to construct derived datatype consisting of elements of same datatype.

```
MPI_Type_vector
MPI_Type_contiguous
MPI_Type_indexed
```

`MPI_Type_vector` group data which are equally separated entries in an array.

`MPI_Type_contiguous` group data located in contiguous memory locations, for example sequence of entries in an array.

`MPI_Type_indexed` group data of same type located at specified locations, for example the diagonal elements of a square matrix.

MPI_Type_vector

The calling sequence of `MPI_Type_vector` is

```
int MPI_Type_vector (
        int               count,
        int               block_length,
        int               stride,
        MPI_Datatype      type,
        MPI_Datatype*     mesg_mpi)
```

Arguments of the function are scalars unlike in `MPI_Type_struct`, where other than `count` and `mesg_mpi` were arrays. This is a consequence of grouping data of same datatype.

`Stride` is the separation of each elements as entries in an array.

`MPI_Type_vector` an Example

Consider the 4×4 matrix, schematically represented as

3	7	21	1
8	6	2	0
12	9	1	3
8	4	2	12

ROW

Columns

In C matrices are stored in row major format (it is column major in FORTRAN), the elements of a column are not contiguous.

In the present example, the first column elements `(3, 8, 12, 8)` have the nearest neighbors separated by four memory locations (in FORTRAN the equivalent would be elements of rows).

MPI_Type_vector an Example (contd)

A derived datatype can be constructed to access the columns of the matrix using MPI_Type_vector.

The calling sequence is

```
MPI_Type_vector (
 /* int           */ 4,  /* count        */
 /* int           */ 1,  /* block length */
 /* int           */ 3,  /* stride       */
 /* MPI_Datatype*/ MPI_INT,
 /* MPI_Datatype*/ &column_mpi);
```

Commit it to use for future communications

```
MPI_Type_commit(&column_mpi);
```

What would be calling sequence of MPI_Type_vector to group two columns?

MPI_Type_vector an Example (contd)

The calling sequence to group two columns is

```
MPI_Type_vector (
 /* int            */ 4,  /* count        */
 /* int            */ 2,  /* block length */
 /* int            */ 2,  /* stride       */
 /* MPI_Datatype*/ MPI_INT,
 /* MPI_Datatype*/ &column_mpi);
```

and

```
MPI_Type_vector (
 /* int            */ 4,  /* count        */
 /* int            */ 4,  /* block length */
 /* int            */ 0,  /* stride       */
 /* MPI_Datatype*/ MPI_INT,
 /* MPI_Datatype*/ &column_mpi);
```

would group the whole matrix.

Commit it for use in future communications.

Sending and Receiving Grouped Data

We have constructed two derived datatypes so far

1. mesg_mpi_strct
   constructed using `MPI_Type_struct` and consist of a, b and n.
2. column_mpi
   constructed using `MPI_Type_vector` column elements of a 4×4 matrix.

These derived datatypes can be use in any MPI communication function call. For example

1. Broadcast a, b and n using `mesg_mpi_strct`

```
MPI_Bcast(&a, 1, mesg_mpi_strct,
                 0, MPI_COMM_WORLD);
```

2. Send the second column from root to the first ranked processor

```
MPI_Send(&A[0][2], 1, column_mpi,
                1, MPI_COMM_WORLD);
```

Type Matching

Suppose we are using two processors and we construct `mesg_mpi_strct`. Since the memory usage need not be identical on the two processors, the addresses of a, b and n are different on the two processors.

The derived data type `mesg_mpi_strct` are constructed separately by the two processors. In the most general case, the name of the derived datatype can be different on the two processors.

How to ensure that `mesg_mpi_strct` received is the same as the local one. This is achieved by type matching

1. `mesg_mpi_strct` is a sequence of datatype and location pairs. On each processor, the `mesg_mpi_strct` is `{(MPI_FLOAT, &a), (MPI_FLOAT, &b), (MPI_INT, &n)}`

Type Matching (contd)

2. type signature is the sequence of the MPI datatypes, for `mesg_mpi_strct` is `{MPI_FLOAT, MPI_FLOAT, MPI_INT}`

3. type signature must be compatible in a send and receive pair of function calls.

Compatibility of the type signature does not mean exact match between the sender and receiver. Suppose

```
{t0, t1, ......., tn}
```

is the type signature passed to `MPI_Send` and

```
{u0, u1, ......., um}
```

is the type signature specified in `MPI_Recv`.

By compatibility, it means n must be less than or equal to m and ti must be equal to ui.

Type Matching (contd)

This means that a `MPI_Recv` function call with type signature

`{MPI_FLOAT, MPI_FLOAT, MPI_INT, MPI_FLOA`

can receive data sent using the type signature of `mesg_mpi_strct`

`{MPI_FLOAT, MPI_FLOAT, MPI_INT}`

In the more general case, it is not necessary for the processors to share the same sequence of derived datatype constructions.

MPI_Type_contiguous

The calling sequence of `MPI_Type_contiguous` is

```
int MPI_Type_contiguous (
    int              count,
    MPI_Datatype     old_type,
    MPI_Datatype*    new_mpi_type)
```

This derived datatype groups `count` number of consecutive entries of type `old_type`.

This derived datatype can be used to define a new datatype `row_mpi` representing one row of the 4×4 matrix discussed earlier. The calling sequence is

```
MPI_Type_contiguous (
    /* int */                4,
    /* MPI_Datatype  */   MPI_INT,
    /* MPI_Datatype* */   &row_mpi);
```

MPI_Type_indexed

The calling sequence of `MPI_Type_indexed` is

```
int MPI_Type_contiguous (
        int               count,
        int               block_lengths[],
        int               displacements[],
        MPI_Datatype      old_type,
        MPI_Datatype*     new_mpi_type)
```

The calling sequence is very similar to that of `MPI_Type_struct`, except that there is only one entry for the datatype.

`MPI_Type_indexed` (contd)

The derived datatype function `MPI_Type_indexed` can be used to define a new datatype `diag_mpi` representing the diagonal elements of 4×4 matrix discussed earlier. The calling sequence is

```
MPI_Type_contiguous (
    /* int */                4,
    /* int */                block_lengths,
    /* int */                relloc,
    /* MPI_Datatype  */      MPI_INT,
    /* MPI_Datatype* */      &row_mpi);
```

where `block_lengths` and `relloc` are the arrays `{1, 1, 1}` and `{4, 4, 4, 4}` respectively.

Bibliography

1. Books

   (a) Peter S. Pacheco
   Parallel Programming with MPI

   (b) William Gropp, Ewing Lusk, Anthony Skjellum
   Using MPI: Portable Parallel Programming with the Message-Passing Interface

2. URLs

   (a) http://www.cs.usfca.edu/mpi/

   (b) http://www-unix.mcs.anl.gov/mpi/mpich/

   (c) http://www.lam-mpi.org/

   (d) http://www.scali.com/