Derived Data Types in MPI

Dilip Angom
angom@prl.ernet.in

Theoretical Physics and Complex Systems
Physical Research Laboratory
Point to point communication functions.

```c
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 collective communication functions.

```c
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.

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

int MPI_Gather(
    void* send_data /* in */, /* in */,
    int send_count /* in */, /* in */,
    MPI_Datatype send_type /* in */, /* in */,
    void* recv_data /* out */, /* out */,
    int recv_count /* in */, /* in */,
    MPI_Datatype recv_type /* in */, /* in */,
    int root /* in */, /* in */,
    MPI_Comm Comm /* in */)
```
All-all collective communication functions.

```c
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

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>
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_Type_indexed
- MPI_Pack
- MPI_Unpack
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 \((\epsilon)\) 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)
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.
Suppose the following three data elements are defined in the main program.

```c
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.
The MPI function \texttt{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.

\textbf{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

$\{(\texttt{MPI\_FLOAT}, 0), (\texttt{MPI\_FLOAT}, 4), (\texttt{MPI\_INT}, 10)\}$

The final step of constructing a derived datatype is to commit it using the MPI function \texttt{MPI\_Type\_commit}. This is the mechanism to make internal changes that may improve communication performance.
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**
   ```c
   int block_lengths[3]
   block_lengths[0]=1;
   block_lengths[1]=1;
   block_lengths[2]=1;
   ```

2. **Type of each element**
   ```c
   MPI_Datatype typelist[3]
   typelist[0]=MPI_FLOAT;
   typelist[1]=MPI_FLOAT;
   typelist[2]=MPI_INT;
   ```

3. **Address of first element**
   ```c
   MPI_Aint start_add
   MPI_Address(&a, &start_add);
   ```
MPI_Type_struct (contd)

4. Relative locations (MPI_Aint relloc[3])
   
   `reloc[0] = 0;
   MPI_Address(&b, &address);
   reloc[1] = address - start_add;
   MPI_Address(&n, &address);
   reloc[2] = address - start_add;`

5. Build the derived datatype (MPI_Datatype* mesg_mpi_strct)
   
   `MPI_Type_struct(3, block_lengths,
   reloc, typelist, mesg_mpi_strct);`

6. Commit it
   
   `MPI_Type_commit(mesg_mpi_strct);`
The calling sequence of `MPI_Type_struct` is

```c
int MPI_Type_struct ( int count, int block_lengths[], MPI_Aint relocl[], 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 `reloc` refer to the relative location of each element from the beginning of the message.
Datatype of relative location `reloc` 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.
MPI has three functions to construct derived datatype consisting of elements of same datatype.

- **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.
The calling sequence of `MPI_Type_vector` is

```c
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.
Consider the $4 \times 4$ matrix, schematically represented as

$$
\begin{array}{cccc}
3 & 7 & 21 & 1 \\
8 & 6 & 2 & 0 \\
12 & 9 & 1 & 3 \\
8 & 4 & 2 & 12 \\
\end{array}
$$

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

The calling sequence is

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

Commit it to use for future communications

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

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

\[
\text{MPI\_Type\_vector (}
\begin{array}{l}
/* \text{int} */ 4, /* \text{count} */ \\
/* \text{int} */ 2, /* \text{block length} */ \\
/* \text{int} */ 2, /* \text{stride} */ \\
/* \text{MPI\_Datatype} */ \text{MPI\_INT,} \\
/* \text{MPI\_Datatype} */ &\text{column\_mpi)};
\end{array}
\]

and

\[
\text{MPI\_Type\_vector (}
\begin{array}{l}
/* \text{int} */ 4, /* \text{count} */ \\
/* \text{int} */ 4, /* \text{block length} */ \\
/* \text{int} */ 0, /* \text{stride} */ \\
/* \text{MPI\_Datatype} */ \text{MPI\_INT,} \\
/* \text{MPI\_Datatype} */ &\text{column\_mpi)};
\end{array}
\]

would group the whole matrix.

Commit it for use in future communications.
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 $4 \times 4$ 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`
   
   ```c
   MPI_Bcast(&a, 1, mesg_mpi_strct, 0, MPI_COMM_WORLD);
   ```

2. **Send** the second column from root to the first ranked processor
   
   ```c
   MPI_Send(&A[0][2], 1, column_mpi, 1, MPI_COMM_WORLD);
   ```
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) \}
2. **type signature** is the sequence of the MPI datatypes, for `msg_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

\[ \{t_0, t_1, \ldots, t_n\} \]

is the type signature passed to `MPI_Send` and

\[ \{u_0, u_1, \ldots, u_m\} \]

is the type signature specified in `MPI_Recv`.

By compatibility, it means \(n\) must be less than or equal to \(m\) and \(t_i\) must be equal to \(u_i\).
This means that a `MPI_Recv` function call with type signature

```
{MPI_FLOAT, MPI_FLOAT, MPI_INT, MPI_FLOAT}
```

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.
The calling sequence of `MPI_Type_contiguous` is

```c
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 $4 \times 4$ matrix discussed earlier. The calling sequence is

```c
MPI_Type_contiguous (   
    /* int */ 4,           
    /* MPI_Datatype */ MPI_INT,  
    /* MPI_Datatype* */ &row_mpi);  
```
The calling sequence of `MPI_Type_indexed` is

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

```c
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/