Blocking synchronous send

→ the sender doesn’t return until it receives the acknowledgement from the receiver that the message has been received

Format:

MPI_Ssend (buf, count, datatype, dest, tag, comm)
The sender doesn’t return until the application buffer can be reused

Format:

MPI_Bsend(&buf, count, datatype, dest, tag, comm)

Must attach buffer space using:

MPI_Buffer_attach(buffer, size)

Buffer space is detached using:

MPI_Buffer_detach(buffer, size)
Blocking buffered send (2)

Determining the size of the buffer

\[ \text{MPI\_Buffer\_attach}( \text{buffer, size} ); \]
\[ \text{MPI\_Bsend}( ..., \text{count}=20, \text{datatype}=\text{type1}, ... ); \]
\[ \text{MPI\_Bsend}( ..., \text{count}=40, \text{datatype}=\text{type2}, ... ); \]

the value of \text{size} should be greater than the value computed by

\[ \text{MPI\_Pack\_size}( 20, \text{type1}, \text{comm}, &s1 ); \]
\[ \text{MPI\_Pack\_size}( 40, \text{type2}, \text{comm}, &s2 ); \]
\[ \text{size} = s1 + s2 + 2 \times \text{MPI\_BSEND\_OVERHEAD}; \]

\text{MPI\_BSEND\_OVERHEAD} can be found in \text{mpi.h} (for C) and \text{mpif.h} (for Fortran)
Blocking ready send

The sender returns when the application buffer can be reused

Format:

\\texttt{MPI\_Rsend (buf, count, datatype, dest, tag, comm)}
Blocking standard send

Format:

MPI_Send(buf, count, datatype, dest, tag, comm)
Non-blocking synchronous send

Format:

\[ \text{MPI} \_ \text{Issend (buf, count, datatype, dest, tag, comm, request)} \]

- The routine returns immediately, but the communication is considered complete when the sender receives the acknowledgement from the receiver

- The system issues a unique “request handle”

- The request can be used later to determine the completion of the communication

- Other non-blocking send functions are similar, all have one more parameter, \textit{request}, in addition to the parameter list of the corresponding blocking send functions
Two Receive routines

Blocking receive routine: MPI_Recv()

Non-blocking receive routine: MPI_Irecv()
Virtual topology

Is a mechanism for naming the processes in a communicator in a way that fits the communication pattern better

Useful when the structure of the virtual topology matches the communication pattern

Add convenience to MPI (can make coding easier)
**Cartesian topology**

naming the processes in a communicator using Cartesian coordinates
Cartesian topology

Create a Cartesian topology

```c
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
```

- `[IN comm_old]` input communicator
- `[IN ndims]` number of dimensions of cartesian grid
- `[IN dims]` integer array of size ndims specifying the number of processes in each dimension
- `[IN periods]` logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
- `[IN reorder]` ranking may be reordered (true) or not (false)
- `[OUT comm_cart]` communicator with new cartesian topology (handle)

The topology is only accessible through the new communicator returned in `comm_cart`
Converting between ranks and coordinates

MPI_Cart_rank (comm, coords, rank)

converts process grid coordinates to process rank.

It might be used to determine the rank of a particular process whose grid coordinates are known, in order to send a message to it or receive a message from it.

MPI_Cart_coords (comm, rank, ndims, coords)

converts process rank to coordinates.

It might be used to determine the grid coordinates of a particular process from which a message has just been received.
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Comm comm;
    int dim[2], period[2], reorder;
    int coord[2], id;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 12){
        printf("Please run with 12 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    dim[0]=4; dim[1]=3;
    period[0]=1; period[1]=0;
    reorder=0;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);
    if (rank == 5){
        MPI_Cart_coords(comm, rank, 2, coord);
        printf("Rank %d coordinates are %d %d\n", rank, coord[0], coord[1]);
    }
    if(rank==0){
        coord[0]=3; coord[1]=1;
        MPI_Cart_rank(comm, coord, &id);
        printf("The processor at position (%d, %d) has rank %d\n", coord[0], coord[1], id);  
    }
    MPI_Finalize();
    return 0;
}
#include<mpi.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
    int rank, size, rank2;
    MPI_Comm comm;
    int dim[2], period[2], reorder;
    int coord[2], id;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 12)
    {
        printf("Please run with 12 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    dim[0]=4; dim[1]=3;
    period[0]=1; period[1]=0;
    reorder=1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);
    MPI_Comm_rank(comm, &rank2);
    if (rank == 5)
    {
        MPI_Cart_coords(comm, rank, 2, coord);
        printf("Rank %d coordinates are %d %d\n", rank, coord[0], coord[1]);
    }
    if(rank==0)
    {
        coord[0]=3; coord[1]=1;
        MPI_Cart_rank(comm, coord, &id);
        printf("The processor at position (%d, %d) has rank %d\n", coord[0], coord[1], id);
    }
    MPI_Finalize();
    return 0;
}
Derived datatype

Users can construct (derive) their own data types

The memory layout of a datatype in MPI is expressed as

\{(type_0, offset_0), (type_1, offset_1), \ldots, (type_n, offset_n)\}
MPI_Type_contiguous

int MPI_Type_contiguous( int count, MPI_Datatype old_type, MPI_Datatype *new_type_p );

MPI_Type_contiguous(3, MPI_REAL, newtype)
// returns a new datatype that represents the concatenation of 3
// instances of MPI_REAL.

MPI_Type_commit(newtype)
// commits the datatype, must be done before communication

MPI_Send(buff, 1, newtype, dest, tag, MPI_COMM_WORLD)
// sends the data at location data to dest

MPI_Type_free(newtype)
// frees the datatype

This is equivalent to the following single call

MPI_SEND(buff, 3, MPI_REAL, dest, tag, MPI_COMM_WORLD)

Where the elements to be sent are already contiguous
Memory layout in last example (assume MPI_REAL occupies 32 bits)

- Old datatype = {(MPI_REAL, 0)}
- New datatype = {(MPI_REAL, 0), (MPI_REAL, 32), (MPI_REAL, 64)}
**MPI_Type_vector**

**MPI_TYPE_VECTOR (count, blocklen, stride, oldtype, newtype)**

Defines a derived type *newtype* comprising *count* consecutive blocks of data elements with datatype *oldtype*, with each block containing *blocklen* data elements, and the start of successive blocks separated by *stride* data elements. E.g. (Assume MPI_FLOAT occupies 16 bits)

```c
float data [1024];
MPI_Datatype floattype;
MPI_TYPE_vector (3, 2, 4, MPI_FLOAT, &floattype);
MPI_Type_commit (&floattype);
MPI_Send (data, 1, floattype, dest, tag, MPI_COMM_WORLD);
MPI_Type_free (&floattype)
```
The code in last slide is equivalent to the following code

```c
float data[1024], buff[6];
for (i=0; i<3; i++) {
    buff[j] = data[i*4*16];
    buff[j+1]=data[i*4*16+16];
}
MPI_Send (buff, 6, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
```

Both send 6 FP numbers from locations data[0], data[64], data[128]

→ The memory layout in the example (Assume MPI_FLOAT occupies 16 bits)

- Old datatype = `{(MPI_FLOAT, 0)}`
- New datatype = `{(MPI_FLOAT, 0), (MPI_FLOAT, 16), (MPI_FLOAT, 64), (MPI_FLOAT, 80), (MPI_FLOAT, 128), (MPI_FLOAT, 144)}`
MPI_Type_create_indexed_block

int MPI_Type_create_indexed_block( int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype );

Parameters:

count
  - [in] length of array of displacements (integer)

blocklength
  - [in] size of block (integer)

array_of_displacements
  - [in] array of displacements (array of integer)

oldtype
  - [in] old datatype (handle)

newtype
  - [out] new datatype (handle)
### MPI_Type_create_indexed_block

```c
int MPI_Type_create_indexed_block( int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype );
```

Let `oldtype = {(double, 0), (char, 32)} with extent 40 bits. Let D = (4, 1). After calling MPI_Type_indexed(2, 2, D, oldtype, newtype)

```text
newtype=
{
(double, 160), (char, 192), (double, 200), (char, 232)
(double, 40), (char, 72), (double, 80), (char, 112)
}
```

![Diagram](image)
MPI_Type_Indexed

MPI_Type_Indexed (count, lengths[], offsets[], oldtype, newtype)

→ Used in the case where the elements in the datatype to be constructed have the same type, but each block has different number of elements and the distance between two consecutive blocks are different

→ Used to define a type comprising one or more blocks of a primitive or previously defined datatype, where block lengths and the displacement between blocks are specified in arrays

→ The above call defines a type newtype comprising count consecutive blocks of data elements with type oldtype, with block i having a displacement of offsets data elements and containing lengths data elements
MPI_Type_Indexed

MPI_Type_Indexed (count, lengths[], offsets[], oldtype, newtype)

Let oldtype = {(double, 0), (char, 32)} with extent 40 bits. Let B = (1, 3) and let D = (1, 4).
After calling MPI_Type_indexed(2, B, D, oldtype, newtype)

newtype =
{
(ddouble, 40), (char, 72),
(ddouble, 160), (char, 192), (double, 200), (char, 232), (double, 240), (char, 272),
}
int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_int *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

The derived datatype includes different datatypes, each block with different displacements and different data elements
Let oldtype1 = {(double, 0), (char, 32)} with extent 40 bits, and oldtype2={((float, 0), (char, 16))} with extent 24 bits
Let B = (3, 1) and D = (160, 16) and C=(oldtype1, oldtype2)
After calling MPI_Type_struct(2, B, D, C, newtype)

newtype=
{(double, 160), (char, 192), (double, 200), (char, 232), (double, 240), (char, 272), (float, 16), (char, 32)}
## Derived datatype summary

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Same Datatype in each block</th>
<th>Same number of elements in each block</th>
<th>Same Block distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_vector</td>
<td>yes</td>
<td>yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MPI_Type_create_indexed_block</td>
<td>yes</td>
<td>yes</td>
<td>No</td>
</tr>
<tr>
<td>MPI_Type_indexed</td>
<td>yes</td>
<td>no</td>
<td>No</td>
</tr>
<tr>
<td>MPI_Type_struct</td>
<td>no</td>
<td>no</td>
<td>No</td>
</tr>
</tbody>
</table>
Summary of MPI

Point-to-point communication

Collective communication

Communication modes

Virtual topology

Derived datatype