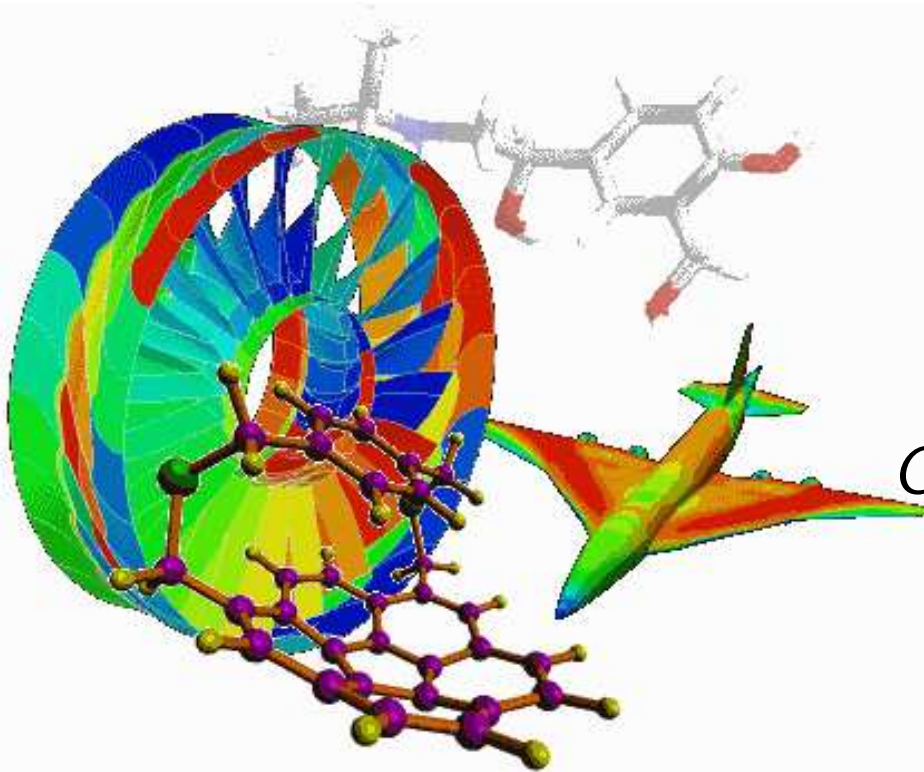


Communication in MPI. Continued



CME342 / AA220 / CS238
Lecture 6-7
April 11-13, 2005

Announcements

- If you have not registered for the `cs238-class` list, please do so.
- On HW1, do not use implied Gauss-Seidel iteration in the code excerpt. Simply use a Jacobi update by operating on a temporary array `a(i,j)` doing the Jacobi sweep, and then copying the result to the `b(i,j)` array. It is much easier this way.

Online MPI Documentation

Notice that on the course web page you have a series of links to web pages with useful information on MPI.

In the past I have found:

`http://www-unix.mcs.anl.gov/mpi`

particularly useful as a source of explanations of all the components of MPI and as a reference manual to use while developing code. Click on MPI Standard 1.1 to get to the information. Go to the bottom and click on Index for a full MPI Reference.

On the same web page you can click on MPI Standard 2.0 if you want to find out more about advanced features of the extension to the original MPI standard (particularly parallel I/O and one-sided communication).

MPI_RECEIVE Operations

Notice that it is not necessary to specify either a source or a tag for a message that is being received. In fact, it often may be advantageous to use wildcards for these arguments. These wildcards are:

- `MPI_ANY_SOURCE`
- `MPI_ANY_TAG`

and can be used both separately and in combination. Once you receive a message from an arbitrary source or with an arbitrary tag, you may want to know where it came from or what tag it carried, or other pieces of information about the message. This information is encoded in the status array/structure and can be accessed either directly or through auxiliary functions (`MPI_GET_COUNT` etc.)

Buffered / Nonbuffered Comm.

- No-buffering (*phone calls*)
 - ▷ Proc 0 initiates the send request and rings Proc 1. It **waits** until Proc 1 is ready to receive. The transmission starts.
 - ▷ Synchronous comm. – completed only when the message was received by the receiving proc.
- Buffering (*beeper*)
 - ▷ The message to be sent (by Proc 0) is copied to a system-controlled block of memory (buffer).
 - ▷ Proc 0 can continue executing the rest of its program.
 - ▷ When Proc 1 is ready to receive the message, the system copies the buffered message to Proc 1.
 - ▷ Asynchronous comm. – may be completed even though the receiving proc has not received the message.

Buffered Comm. (cont.)

- Buffering requires system resources, e.g. memory, and can be slower if the receiving proc is ready at the time of requesting the send.
- Application buffer: address space that holds the data.
- System buffer: system space for storing messages. In buffered comm., data in application buffer is copied to/from system buffer.
- MPI allows comm. in buffered mode:
MPI_Bsend, MPI_Ibsend.
- User allocates the buffer by:
MPI_Buffer_attach(buffer, buffer_size)
- Free the buffer by MPI_Buffer_detach.

Blocking / Nonblocking Comm.

- Blocking Comm. (*McDonald's*)
 - ▷ The receiving proc has to wait if the message is not ready.
 - ▷ Different from synchronous comm.
 - ▷ Proc 0 may have already buffered the message to system and Proc 1 is ready, but the interconnection network is busy.
- Nonblocking Comm. (*In & Out*)
 - ▷ Proc 1 checks with the system if the message has arrived yet. If not, it continues doing other stuff. Otherwise, get the message from the system.
- Useful when computation and comm. can be performed at the same time.
- MPI allows both nonblocking send & receive:

`MPI_Isend`, `MPI_Irecv`.

- In nonblocking send, program identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for message to be copied out from application buffer.
- The program **should not** modify the application buffer until the nonblocking send has completed.
- Nonblocking comm. can combined with nonbuffering: `MPI_Issend`, or buffering: `MPI_Ibsend`.
- Use `MPI_Wait` or `MPI_Test` to determine if the nonblocking send or receive has completed. Also available `MPI_WAITANY`, `MPI_WAITALL`, `MPI_TESTANY`, `MPI_TESTALL`.

Non-Blocking Send Syntax

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
[ IN buf] initial address of send buffer (choice)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] datatype of each send buffer element (handle)
[ IN dest] rank of destination (integer)
[ IN tag] message tag (integer)
[ IN comm] communicator (handle)
[ OUT request] communication request (handle)
```

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Non-Blocking Receive Syntax

MPI_IRecv (buf, count, datatype, source, tag, comm, request)
[OUT buf] initial address of receive buffer (choice)
[IN count] number of elements in receive buffer (integer)
[IN datatype] datatype of each receive buffer element (handle)
[IN source] rank of source (integer)
[IN tag] message tag (integer)
[IN comm] communicator (handle)
[OUT request] communication request (handle)

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,  
int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)  
<type> BUF(*)  
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

Communication Completion

Wait until the communication operation associated with the specified request is completed. Note that for a send operation, this simply means that the message has been sent, and the send buffer is ready for reuse. This does **NOT** mean that the corresponding receive operation has also completed.

```
MPI_WAIT(request, status)
[ INOUT request] request (handle)
[ OUT status] status object (Status)
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Communication Completion

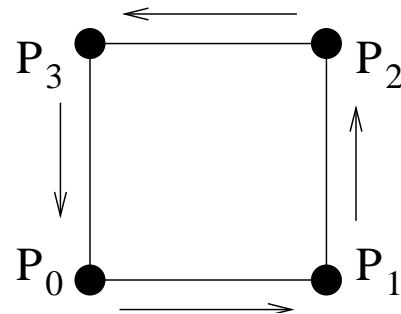
Test if the communication operation specified by `REQUEST` has completed or not. Status is returned in `FLAG`. If the communication request has not completed, you can go do something else and test again later.

```
MPI_TEST(request, flag, status)
[ INOUT request] communication request (handle)
[ OUT flag] true if operation completed (logical)
[ OUT status] status object (Status)
```

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Example: Data Exchange in a Ring Topology



- Blocking version:

```
for (i=0; i<p; i++) {  
    send_offset = ((my_id-i+p) % p)*blksize;  
    recv_offset = ((my_id-i-1+p) % p)*blksize;  
    MPI_Send(y+send_offset, blksize, MPI_FLOAT,  
            my_id+1, 0, ring_comm);  
    MPI_Recv(y+recv_offset, blksize, MPI_FLOAT,  
            my_id-1, 0, ring_comm, &status);  
}
```

- Nonblocking version:

```
send_offset = my_id*blksize;
recv_offset = (my_id-1+p)*blksize;
for (i=0; i<p-1; i++) {
    MPI_Isend(y+send_offset, blksize, MPI_FLOAT,
             my_id+1, 0, ring_comm, &send_request);
    MPI_Irecv(y+recv_offset, blksize, MPI_FLOAT,
             my_id-1, 0, ring_comm, &recv_request);

    send_offset = ((my_id-i-1+p) % p)*blksize;
    recv_offset = ((my_id-i-2+p) % p)*blksize;

    MPI_Wait(&send_request, &status);
    MPI_Wait(&recv_request, &status);
}
```

- The comm. & computations of next offsets are **overlapped**.

Summary: Comm. Modes

- 4 comm. modes in MPI: standard, buffered, synchronous, ready. They can be either blocking or nonblocking.
- In standard modes (`MPI_Send`, `MPI_Recv`, ...), it is up to the system to decide whether messages should be buffered.
- In synch. mode, a send won't complete until a matching receive has been posted which has begun reception of the data.
 - ▷ `MPI_Ssend`, `MPI_Issend`.
 - ▷ No system buffering.
- In buffered mode, the completion of a send does not depend on the existence of a matching receive.

- ▷ MPI_Bsend, MPI_Ibsend.
 - ▷ System buffering by MPI_Buffer_attach & MPI_Buffer_detach.
- Ready mode not discussed.
 - Make sure to look through the MPI reference to fully understand all the subtleties of the various communication modes.
 - Homework #1 will allow you to gain deeper understanding in practice.

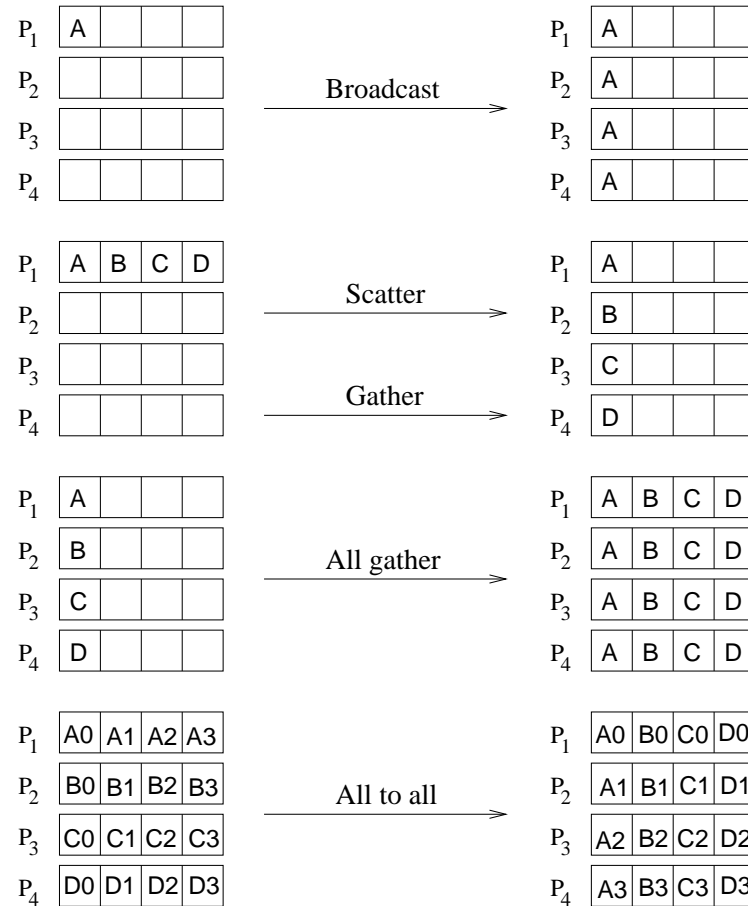
Collective Communication

- Comm. pattern involving a group of procs; usually more than 2.
- MPI_Barrier: synchronize all procs.
- Broadcast (MPI_Bcast)
 - ▷ A single proc sends the same data to every proc.
- Reduction (MPI_Reduce)
 - ▷ All the procs contribute data that is combined using a binary operation.
 - ▷ Example: max, min, sum, etc.
 - ▷ One proc obtains the final answer.
- Allreduce (MPI_Allreduce)
 - ▷ Same as MPI_Reduce but every proc contains the final answer.
 - ▷ Effectively as MPI_Reduce + MPI_Bcast, but more efficient.

Other Collective Comm.

- **Scatter** (MPI_Scatter)
 - ▷ Split the data on proc *root* into p segments.
 - ▷ The 1st segment is sent to proc 0, the 2nd to proc 1, etc.
 - ▷ Similar to but more general than MPI_Bcast.
- **Gather** (MPI_Gather)
 - ▷ Collect the data from each proc and store the data on proc *root*.
 - ▷ Similar to but more general than MPI_Reduce.
- Can collect and store the data on *all* procs using MPI_Allgather.

Comparisons of Collective Comms.



Barrier Synchronization

```
MPI_BARRIER( comm )  
[ IN comm] communicator (handle)
```

```
int MPI_Barrier(MPI_Comm comm )
```

```
MPI_BARRIER(COMM, IERROR)  
INTEGER COMM, IERROR
```

`MPI_BARRIER` blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

Broadcast

```
MPI_BCAST( buffer, count, datatype, root, comm )  
[ INOUT buffer] starting address of buffer (choice)  
[ IN count] number of entries in buffer (integer)  
[ IN datatype] data type of buffer (handle)  
[ IN root] rank of broadcast root (integer)  
[ IN comm] communicator (handle)
```

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,  
             int root, MPI_Comm comm )
```

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)  
<type> BUFFER(*)  
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of

group using the same arguments for `comm`, `root`. On return, the contents of `root`'s communication buffer has been copied to all processes.

For example: Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;  
    int array[100];  
    int root=0;  
    ...  
    MPI_Bcast( array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as `comm` in the above) have been assigned appropriate values.

Gather

```
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf,  
            recvcount, recvtype, root, comm)
```

[IN sendbuf] starting address of send buffer (choice)

[IN sendcount] number of elements in send buffer (integer)

[IN sendtype] data type of send buffer elements (handle)

[OUT recvbuf] address of receive buffer (choice, significant only at root)

[IN recvcount] number of elements for any single receive
(integer, significant only at root)

[IN recvtype] data type of recv buffer elements
(significant only at root) (handle)

[IN root] rank of receiving process (integer)

[IN comm] communicator (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
           REVCOUNT, RECVTYPE, ROOT, COMM, IERROR)  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to `MPI_SEND` and the root had executed n calls to `MPI_RECV`.

Scatter

```
MPI_SCATTER( sendbuf, sendcount, sendtype, recvbuf,  
             recvcount, recvtype, root, comm)
```

[IN sendbuf] address of send buffer (choice, significant only at root)

[IN sendcount] number of elements sent to each process
(integer, significant only at root)

[IN sendtype] data type of send buffer elements
(significant only at root) (handle)

[OUT recvbuf] address of receive buffer (choice)

[IN recvcount] number of elements in receive buffer (integer)

[IN recvtype] data type of receive buffer elements (handle)

[IN root] rank of sending process (integer)

[IN comm] communicator (handle)

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
            RECVCOUNT, RECVMODE, ROOT, COMM, IERROR)  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMODE, ROOT, COMM, IERROR
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

The outcome is as if the root executed n send operations, and each process executed a receive.

All-to-All

`MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`
[IN sendbuf] starting address of send buffer (choice)
[IN sendcount] number of elements sent to each process (integer)
[IN sendtype] data type of send buffer elements (handle)
[OUT recvbuf] address of receive buffer (choice)
[IN recvcount] number of elements received from any process (integer)
[IN recvtype] data type of receive buffer elements (handle)
[IN comm] communicator (handle)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
                void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,  
             RECVTYPE, COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
```

```
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

`MPI_ALLTOALL` is an extension of `MPI_ALLGATHER` to the case where each process sends distinct data to each of the receivers. The j th block sent from process i is received by process j and is placed in the i th block of `recvbuf`.

Reduce

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
[ IN sendbuf] address of send buffer (choice)
[ OUT recvbuf] address of receive buffer (choice, significant only at root)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] data type of elements of send buffer (handle)
[ IN op] reduce operation (handle)
[ IN root] rank of root process (integer)
[ IN comm] communicator (handle)
```

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

`MPI_REDUCE` combines the elements provided in the input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer of the process with rank `root`. The input buffer is defined by the arguments `sendbuf`, `count` and `datatype`; the output buffer is defined by the arguments `recvbuf`, `count` and `datatype`; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for `count`, `datatype`, `op`, `root` and `comm`. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence.

There are a series of pre-defined operations

```
[ MPI_MAX] maximum  
[ MPI_MIN] minimum  
[ MPI_SUM] sum
```

[MPI_PROD] product
[MPI_LAND] logical and
[MPI_BAND] bit-wise and
[MPI_LOR] logical or
[MPI_BOR] bit-wise or
[MPI_LXOR] logical xor
[MPI_BXOR] bit-wise xor
[MPI_MAXLOC] max value and location
[MPI_MINLOC] min value and location

The user can also define global operations to perform on distributed data with MPI_OP_CREATE.

Limitations of Point-To-Point Communication

So far, all point-to-point communication has involved contiguous buffers containing elements of the same fundamental types provided by MPI such as `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`, etc. This is rather limiting since one often wants to:

- Send messages with different kinds of datatypes (say a few integers followed by a few reals.)
- Send non-contiguous data (such as a sub-block of a matrix or an unstructured subset of a one-dimensional array.)

These two goals could easily be accomplished by either sending a number of messages with all data of one type and by copying the non-contiguous data to a contiguous array that is later sent in a single message.

There are several disadvantages

- A larger number of messages than necessary may need to be sent.
- Additional inefficient memory-to-memory copying may be required to fill contiguous buffers.
- Resulting code is needlessly complicated.

MPI *derived datatypes* allow us to carry out these operations: we can send/receive messages with non-contiguous data of different types without the need for multiple messages or additional memory-to-memory copying.

Definition of a Derived Datatype

A *general datatype* is an object that specifies two things:

1. A sequence of basic datatypes.
2. A sequence of integer (byte) displacements.

The displacements do not need to be positive, distinct, or in any particular order. The sequence of pairs of datatype-displacement is called a *type map*, while the sequence of datatypes alone is called the *type signature*.

For example,

Typemap = (type_0, disp_0), (type_1, disp_1), ..., (type_n, disp_n)

can be such a map, whose signature would be

`Typesig = type_0, type_1, ..., type_n`

Derived datatypes can be used in all communication routines (`MPI_SEND`, `MPI_RECV`, ...) in place of the basic datatypes. In addition, by using MPI's derived datatype constructors, you can automatically select data of interest such as subportions of a block of data, both in structured and unstructured fashion.

Datatype Constructors. MPI_TYPE_CONTIGUOUS

The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
[ IN count] replication count (nonnegative integer)
[ IN oldtype] old datatype (handle)
[ OUT newtype] new datatype (handle)
```

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

`newtype` is the datatype obtained by concatenating `count` copies of `oldtype`.

Let the old type have a map

`(double,0), (char,8)`

and let `count = 3`. The type map of the newtype datatype is

`(double,0), (char,8), (double,16), (char,24), (double,32), (char,40)`

MPI_TYPE_VECTOR

The function `MPI_TYPE_VECTOR` is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

```
MPI_TYPE_VECTOR( count, blocklength, stride, oldtype, newtype)
[ IN count] number of blocks (nonnegative integer)
[ IN blocklength] number of elements in each block (nonnegative integer)
[ IN stride] number of elements between start of each block (integer)
[ IN oldtype] old datatype (handle)
[ OUT newtype] new datatype (handle)
```

```
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,  
IERROR)
```

```
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

Assume again that the oldtype has map

```
(double,0), (char,8)
```

A call to `MPI_TYPE_VECTOR(2,3,4,oldtype,newtype)`

will create the datatype with type map

```
(double,0), (char,8), (double,16), (char,24), (double,32), (char,40),
```

```
(double,64), (char,72), (double,80), (char,88), (double,96), (char,104)
```

A call to `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)` is equivalent to a call to `MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype)`, or to a call to `MPI_TYPE_VECTOR(1, count, n, oldtype, newtype)`, `n` arbitrary.

The function `MPI_TYPE_HVECTOR` is identical to `MPI_TYPE_VECTOR`, except that `stride` is given in bytes, rather than in elements.

MPI_TYPE_INDEXED

The function `MPI_TYPE_INDEXED` allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

```
MPI_TYPE_INDEXED(count, array_of_blocklengths,  
                 array_of_displacements, oldtype, newtype)  
[ IN count] number of blocks (nonnegative integer)  
[ IN array_of_blocklengths] number of elements per block  
                        (array of nonnegative integers)  
[ IN array_of_displacements] displacement for each block,  
                        in multiples of oldtype extent (array of integers)  
[ IN oldtype] old datatype (handle)  
[ OUT newtype] new datatype (handle)
```

```
int MPI_Type_indexed(int count, int *array_of_blocklengths, int
```

```
*array_of_displacements, MPI_Datatype oldtype, MPI_Datatype  
*newtype)
```

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,  
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)  
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),  
OLDTYPE, NEWTYPE, IERROR
```

MPI_COMMIT and MPI_FREE

A datatype object has to be committed before it can be used in a communication. A committed datatype can still be used as a argument in datatype constructors. There is no need to commit basic datatypes. They are “pre-committed.”

```
MPI_TYPE_COMMIT(datatype)
[ INOUT datatype] datatype that is committed (handle)
```

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

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR
```

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

```
MPI_TYPE_FREE(datatype)
[ INOUT datatype] datatype that is freed (handle)
```

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

```
MPI_TYPE_FREE(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR
```

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Derived datatypes that were defined from the freed datatype are not affected.

For example,

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
           ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
```

```

                ! now type1 can be used for communication
type2 = type1
                ! type2 can be used for communication
                ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
                ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
                ! now type1 can be used anew for communication

```

Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Sending a Section of a 3D Array

```
REAL a(100,100,100), e(9,9,9)
INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)
```

```
C    extract the section a(1:17:2, 3:11, 2:10)
C    and store it in e(:, :, :).
```

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
```

```
CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
```

```
C    create datatype for a 1D section
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)
```

```
C    create datatype for a 2D section
CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)
```

```
C      create datatype for the entire section
CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, 1,
                      threeslice, ierr)

CALL MPI_TYPE_COMMIT( threeslice, ierr)
CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Transpose of a Matrix

```
REAL a(100,100), b(100,100)
INTEGER row, xpose, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)
```

C transpose matrix a onto b

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
```

```
CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
```

C create datatype for one row

```
CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```

C create datatype for matrix in row-major order

```
CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
```

```
CALL MPI_TYPE_COMMIT( xpose, ierr)
```

```
C    send matrix in row-major order and receive in column major order  
CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,  
                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Processor Topologies

When writing a parallel program, you often have to arrange the processors into some sort of topology for communication purposes. For example, in HW#1, you are arranging a number of processors into a 2D grid. In more complicated solvers, you may have a 3D grid (single block Navier-Stokes solver, FLO107), or a 3D periodic grid (Direct Numerical Simulation of turbulence in a periodic box), or a more general graph connecting processors in an irregular fashion (TFLO, multiblock structured code with arbitrary mesh topologies, AIRPLANE, unstructured flow solver with domain decomposition).

As a programmer, you have to keep track of the processors (and their rank) that each processor needs to communicate with. Although this can be done by hand, MPI has a number of functions/subroutines that facilitate this procedure greatly, especially in the case of n-dimensional structured (periodic or non-periodic) topologies.

The more general case of a graph is not discussed in the notes, but can be found in the documentation in the web.

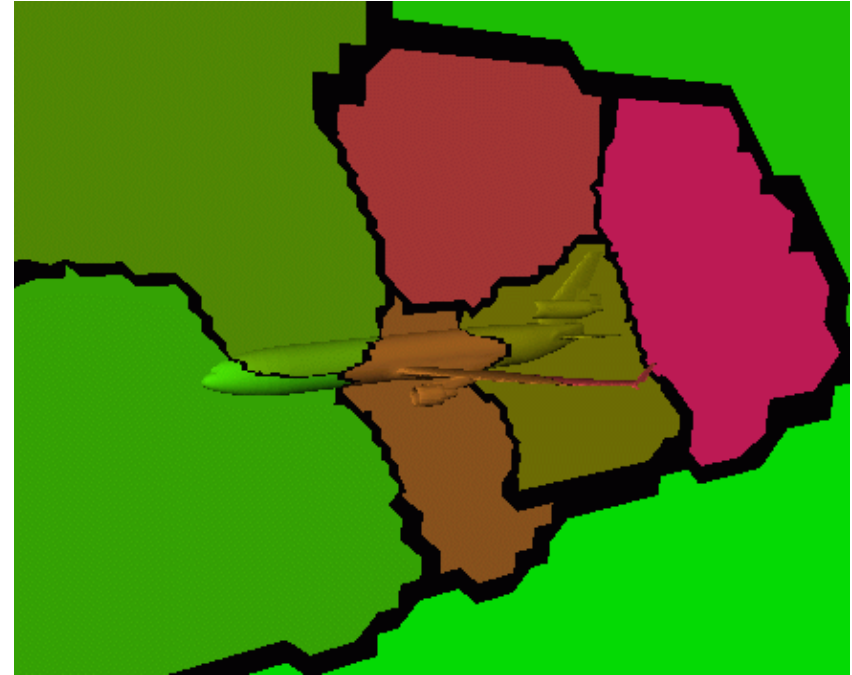
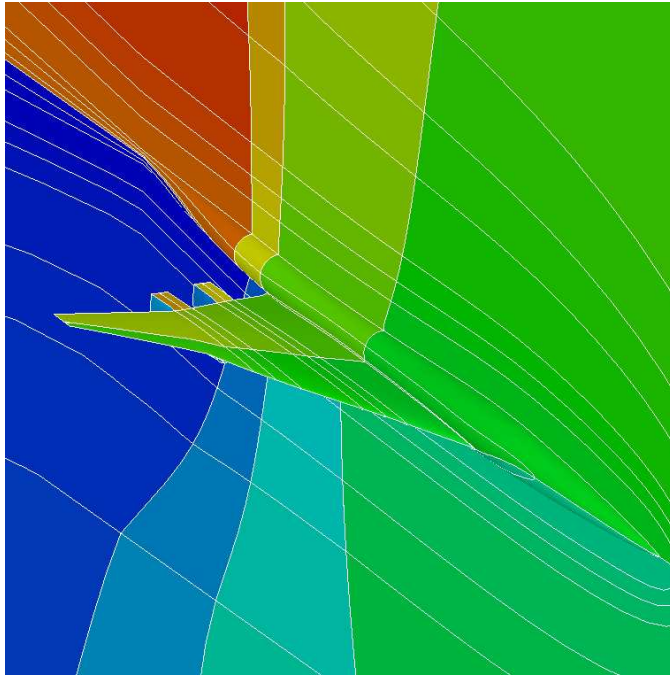


Figure 1: TFLO-type Multiblock Mesh (left) and Fully Unstructured AIRPLANE Mesh (right).

Virtual Topologies

Virtual topologies in MPI have the dual advantage that

- In the midst of communication, it may be easier to refer to the processors that we need to communicate with through MPI built-in functions than by calculating the processor ranks ourselves.
- *If* (and this is a BIG if) the MPI implementation is *smart* it can take advantage of the virtual topology constructors to infer the processor arrangement you would like to use and map processes more efficiently to the underlying hardware. I am not aware of any implementations that do this.

Note that despite the fact that you may have defined a virtual topology that only has connections to nearest neighbors, you can still communicate with all the processors in the communicator if you continue to use processor ranks as the arguments to the point-to-point functions.

The functions `MPI_GRAPH_CREATE` and `MPI_CART_CREATE` are used to create general (graph) virtual topologies and cartesian topologies, respectively. These topology creation

functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator `comm_old`, which defines the set of processes on which the topology is to be mapped. A new communicator `comm_topo1` is created that carries the topological structure as cached information. In analogy to function `MPI_COMM_CREATE` (which creates a new communicator out of a subgroup of processes), no cached information propagates from `comm_old` to `comm_topo1`.

MPI_CART_CREATE

MPI_CART_CREATE can be used to describe cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
[ IN comm_old] input communicator (handle)
[ IN ndims] number of dimensions of cartesian grid (integer)
[ 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) (logical)
[ OUT comm_cart] communicator with new cartesian topology (handle)
```

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

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)  
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR  
LOGICAL PERIODS(*), REORDER
```

`MPI_CART_CREATE` returns a handle to a new communicator to which the cartesian topology information is attached. If `reorder = false` then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the cartesian grid is smaller than the size of the group of `comm`, then some processes are returned `MPI_COMM_NULL`. The call is erroneous if it specifies a grid that is larger than the group size.

Cartesian Topology Utility Functions

`MPI_CARTDIM_GET(comm, ndims)`

[IN comm] communicator with cartesian structure (handle)

[OUT ndims] number of dimensions of the cartesian structure (integer)

`int MPI_Cartdim_get(MPI_Comm comm, int *ndims)`

`MPI_CARTDIM_GET(COMM, NDIMS, IERROR)`

`INTEGER COMM, NDIMS, IERROR`

The functions `MPI_CARTDIM_GET` and `MPI_CART_GET` return the cartesian topology information that was associated with a communicator by `MPI_CART_CREATE`.

`MPI_CART_GET(comm, maxdims, dims, periods, coords)`

[IN comm] communicator with cartesian structure (handle)

[IN maxdims] length of vectors `dims`, `periods`, and `coords` in the
calling program (integer)

[OUT dims] number of processes for each cartesian dimension (array of integer)
[OUT periods] periodicity (true/ false) for each cartesian
 dimension (array of logical)
[OUT coords] coordinates of calling process in cartesian
 structure (array of integer)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int
*coords)
```

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)
```

```
MPI_CART_RANK(comm, coords, rank)
[ IN comm] communicator with cartesian structure (handle)
[ IN coords] integer array (of size ndims) specifying the cartesian coordinates
[ OUT rank] rank of specified process (integer)
```

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
```

```
INTEGER COMM, COORDS(*), RANK, IERROR
```

For a process group with cartesian structure, the function `MPI_CART_RANK` translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension i with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i) < 0` or `coords(i) > dims(i)`, it is shifted back to the interval $0 \leq \text{coords}(i) < \text{dims}(i)$ automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

```
MPI_CART_COORDS(comm, rank, maxdims, coords)
```

```
[ IN comm] communicator with cartesian structure (handle)
```

```
[ IN rank] rank of a process within group of comm (integer)
```

```
[ IN maxdims] length of vector coord in the calling program (integer)
```

```
[ OUT coords] integer array (of size ndims) containing the cartesian coordinates  
(integer)
```

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)  
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

The inverse mapping, rank-to-coordinates translation is provided by MPI_CART_COORDS.

```
MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)  
[ IN comm] communicator with graph topology (handle)  
[ IN rank] rank of process in group of comm (integer)  
[ OUT nneighbors] number of neighbors of specified process (integer)
```

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)  
INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general, graph topology.

```
MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
[ IN comm] communicator with graph topology (handle)
[ IN rank] rank of process in group of comm (integer)
[ IN maxneighbors] size of array neighbors (integer)
[ OUT neighbors] ranks of processes that are neighbors to specified process (array)
```

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int
*neighbors)
```

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

Cartesian Shift Coordinates

If the process topology is a cartesian structure, a `MPI_SENDRECV` operation is likely to be used along a coordinate direction to perform a shift of data. As input, `MPI_SENDRECV` takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function `MPI_CART_SHIFT` is called for a cartesian process group, it provides the calling process with the above identifiers, which then can be passed to `MPI_SENDRECV`. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
[ IN comm] communicator with cartesian structure (handle)
[ IN direction] coordinate dimension of shift (integer)
[ IN disp] displacement (> 0: upwards shift, < 0: downwards shift) (integer)
[ OUT rank_source] rank of source process (integer)
[ OUT rank_dest] rank of destination process (integer)
```

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int
```

```
*rank_source, int *rank_dest)
```

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)  
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

The direction argument indicates the dimension of the shift, i.e., the coordinate which value is modified by the shift. The coordinates are numbered from 0 to ndims-1, when ndims is the number of dimensions.

Depending on the periodicity of the cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

Example

The communicator, `comm`, has a two-dimensional, periodic, cartesian topology associated with it. A two-dimensional array of REALs is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

....

```
C find process rank
```

```
    CALL MPI_COMM_RANK(comm, rank, ierr))
```

```
C find cartesian coordinates
```

```
    CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
```

```
C compute shift source and destination
```

```
    CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
```

```
C skew array
```

```
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,  
+                             status, ierr)
```