Lecture 13b – 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.**
- **Used with MPI_Isend and MPI_Irecv non-synchronous sends and receives**

MPI_WAIT(request, status)
[INOUT request] request (handle)
[OUT status] status object (Status)
C:
int MPI_Wait(MPI_Request *request, MPI_Status *status)
Fortran 90/95:
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.**
- **Again, used with MPI_Isend and MPI_Irecv non-synchronous communications**

MPI_TEST(request, flag, status)
[INOUT request] communication request (handle)
[OUT flag] true if operation completed (logical)
[OUT status] status object (Status)
C:
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
Fortran 90/95:
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

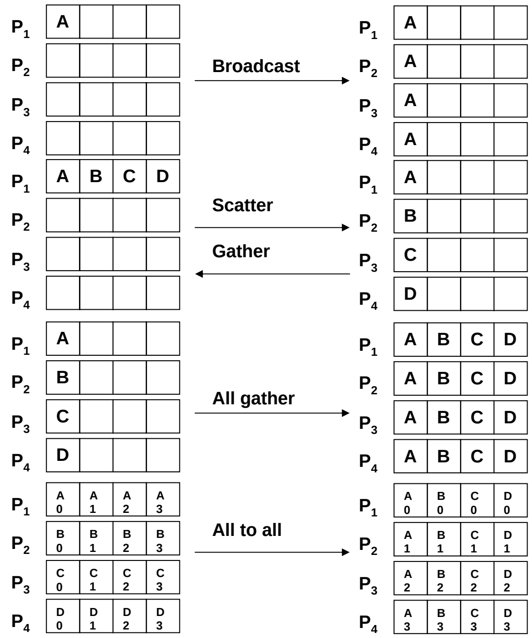
Review: Collective Communication

- **Communication 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 of 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

Review: Other Collective Communicators

- **Scatter (MPI_Scatter)**
  - Split the data on the root processor 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 processor and store the data on root processor
  - Similar to but more general than MPI_Reduce
- **Can collect and store the data on all procs using MPI_Allgather**

Review: Comparison of Collective Communicators



Barrier Synchronization

MPI_Barrier(comm)

[IN comm] communicator (handle)

C:

Int MPI_Barrier(MPI_Comm comm)

Fortran 90/95:

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**
- **Use this only where needed since there is high overhead associated with synching processors**

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)

C:

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

Fortran 90/95:

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

Broadcast

- **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 integers 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. Broadcasting is costly, so use Bcast only when needed.**

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)

C:

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

Fortran 90/95:

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


Gather

- 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)

C:

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

Fortran 90/95:

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


Scatter

- 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)

C:

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

Fortran 90/95:

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
    IERROR
```


All-to-All

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

C:

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

Fortran 90/95:

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


Reduce

- 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.

Reduce

- **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.**

Homework 4 (Reading Only)

- **Read Chapters 4-6 in <u>Using MPI</u> by Gropp et al. and review the more advanced MPI routines**
  - The OpenMPI manual can also be used

- **Read Chapter 6 of <u>Introduction to Parallel Computing</u> by Grama et al. if you purchased the book.**

- **Exercise (not for credit)**
  - Write a routine that uses MPI_Send and MPI_Receive to find the minimum and maximum of a set of 100 random numbers
  - Write another routine that uses MPI_Reduce to find the minimum and maximum instead of MPI_Send and MPI_Receive
  - Compare the wall clock time for the two routines using MPI_Wtime.