COS786: Parallel and Distributed Computing

MPI: Introduction to Communication
MPI Standard

- http://www.mpi-forum.org/
- Starting with MPI 1.3
- Finishing with MPI 2.2
OpenMPI

- http://www.open-mpi.org/
- Open MPI combines MPI 1 and MPI 2.
- Version 1.8.7 is the latest, stable release.
Message Passing Interface (MPI)

• Write a program.
• Run the program as multiple processes.
• The processes communicate by passing messages.
Point-to-Point Communication

- Communication:
  - The send operation sends a message to a specified destination.
  - The message consists of data, its length and its type.
  - Information about the source and destination of the message is transmitted as the message's envelope.
  - A matching receive operation receives the message at the destination process.
Point-to-Point Communication

• Message envelope
  – source and destination processes
  – tag
  – communicator
Point-to-Point Communication

• Message tags
  – Integer values
  – Range 0,...,UB where UB >= 32767
  – Query MPI_TAG_UB attribute to determine implementation maximum.
Point-to-Point Communication

- Communicators
  - Specify the context in which messages are sent.
  - Use MPI_COMM_WORLD to communicate with all processes.
Point-to-Point Communication

• Receive:
  – Messages must fit the receive buffer.
  – An overflow error occurs if they do not.
  – Use MPI_PROBE to receive messages of an unknown length.
Point-to-Point Communication

• MPI_PROBE allows one to check incoming messages without receiving them.
• This is useful for allocating memory or deciding how to handle different types of messages.
Point-to-Point Communication

• Blocking semantics:
  – A send operation blocks the calling thread until the message and its envelope are stored in a buffer.
  – The buffer may be provided by a corresponding read operation or it may be a temporary location.
  – The communication mode determines the blocking semantics.
Point-to-Point Communication

• Communication modes:
  - Standard – MPI decides whether or not to use a buffer.
  - Buffered – Messages are always buffered and the user controls the buffer size.
  - Synchronized – Send does not complete until a receive is posted and starts receiving.
  - Ready – Send can only be called if a matching receive has already been posted.
Point-to-Point Communication

• Communication semantics:
  – Certain properties of point-to-point communications are guaranteed by MPI:
    • Order: Messages do not overtake each other.
    • Progress: A matching send and receive will always complete.
    • Resources: MPI is required to do no worse than what is implied by the user-controlled buffering model.
  – Properties **not** guaranteed by MPI:
    • Fairness: Multiple processes cause messages to overtake one another; Multiple threads cause receives to overtake one another.
Point-to-Point Communication

- **Deadlock:**
  - Buffering may cause deadlock when neither process may continue due to resources being unavailable.
  - This may occur when standard mode or synchronous mode messaging is used.
Point-to-Point Communication

if (my_rank == 0)
{
    mpi_send(sbuff, count, MPI_REAL, 1, ...);
    mpi_recv(rbuff, count, MPI_REAL, 1, ...);
}
else  // my_rank == 1
{
    mpi_recv(rbuff, count, MPI_REAL, 0, ...);
    mpi_send(sbuff, count, MPI_REAL, 0, ...);
}
Point-to-Point Communication

```c
if (my_rank == 0)
{
    mpi_recv(rbuff, count, MPI_REAL, 1, ...);
    mpi_send(sbuff, count, MPI_REAL, 1, ...);
}
else  // my_rank == 1
{
    mpi_recv(rbuff, count, MPI_REAL, 0, ...);
    mpi_send(sbuff, count, MPI_REAL, 0, ...);
}
```
Point-to-Point Communication

```c
if (my_rank == 0) {
    mpi_send(sbuff, count, MPI_REAL, 1, ...);
    mpi_recv(rbuff, count, MPI_REAL, 1, ...);
}
else  // my_rank == 1
{
    mpi_send(sbuff, count, MPI_REAL, 0, ...);
    mpi_recv(rbuff, count, MPI_REAL, 0, ...);
}
```
Point-to-Point Communication

• Non-blocking communication:
  – Two mechanisms may be used to increase parallelism:
    • Light-weight threads;
    • Non-blocking send and receive operations.
Point-to-Point Communication

- Non-blocking send and receive:
  - A **send start** initiates but does not complete a send.
  - A **send complete** must be called to verify transmission.
  - A **receive start** initiates but does not complete a receive.
  - A **receive complete** must be called to verify reception.
Point-to-Point Communication

- Communication modes:
  - Send start:
    - Standard, buffered and synchronous modes allow a send start to begin immediately.
    - Ready requires that a matching receive was posted.
  - Send complete:
    - Synchronous: completes only if a matching receive has started.
    - Buffered: completes even if no matching receive is posted.
    - Standard: May or may not complete.
Point-to-Point Communication

• Implications:
  – In standard and synchronous modes, the sending process may proceed ahead of the receiver.
  – In buffered and ready modes, non-blocking operations are advantageous only if computation may proceed while data is copied.
Point-to-Point Communication

- Non-blocking operations may be matched with blocking operations.

- If communication is initiated by the sender,
  - Lower overheads will be achieved if a receive is posted by the time the send is called.
  - However, a receive may complete only after the corresponding send call completes. A non-blocking receive maintains lower overheads without blocking the receiver.
Point-to-Point Communication

• Communication semantics:

  – The existing semantics are extended for non-blocking communication:
    ● Order: Messages are ordered according to the sequence of send initiations.
    ● Progress:
      – A receive will complete if a matching send has been started. If the matching send is non-blocking, then the receive will complete even if the send is not completed.
      – A send will complete if a matching receive has been started, even if no call is issued to complete the receive.
Point-to-Point Communication

• Cancel a pending, non-blocking communication using MPICANCEL.
  – MPICANCEL returns immediately.
  – Cancelling a buffered send removes the message from the buffer and frees the associated memory.
  – An operation marked for cancellation may either complete or be cancelled.
Collective Communication

• All processes call the communication routine with matching arguments

• Semantics are consistent with point-to-point communication routines (i.e. general data types are allowed and must match at sender and receiver).

• Some routines have a single point of origination or reception. The process wherein this point occurs is called the root process. These routines may call for an argument that is “significant at root only” and is ignored in all other processes.
Collective Communication

- Type matching is stronger than for point-to-point. For collective operations, the size of data sent must match the size of data specified by the receiver.

- Collective routines may return as soon as their participation in the communication is complete. This implies that the process may access the communication buffer but **not** that the operation is started or complete.
Collective Communication

- One should not rely on synchronization side-effects of broadcast routines.
- One should allow for the fact that these side-effects may exist.
- Collective routines use the same communicators as point-to-point routines. MPI guarantees that messages are not mixed.
Collective Communication

- The collective routines involve groups of participating processes. However, the routines do not take a process group identifier, as might be expected, but a communicator alone.
Barrier

MPI_BARRIER(comm)

• Blocks the calling processes until all participating processes have called the routine.
Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

- Broadcasts a message from the process with rank root to all processes in the group, including the root process.
- All members call the routine, specifying the same arguments.
- On return, the buffer has been copied to all other processes.
Gather

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

- Each process in the group (root included) sends the contents of its send buffer to the root process.
- The root process stores the received messages in rank order.
- The recvcount argument specifies the number of items received from each process.
Gather (Variable Data Counts)

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

- Data from process $j$ is placed at the $j$th portion of the receive buffer in the root process.
- The $j$th portion begins at $\text{displs}[j]$. 
Scatter

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

- Is the inverse of MPI_GATHER.

- Sends the ith segment of the message to the ith process.

- Note that the specification of counts and types should not cause any location in the send buffer to be read more than once. This is done to ensure symmetry with MPI_GATHER.
Scatter (Variable Data Counts)

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

- Data from sendbuf[displs[j]] to sendbuf[displs[j] + sendcounts[j]] is sent to process j.
Gather to All

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- Like MPI_GATHER but instead of just the root process receiving the results, all processes do.
- The data from the jth process is placed in the jth block of the receive buffer of every process.
Gather to All (Variable Data Counts)

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

- The data received from the jth process is placed in the jth block of every process' receive buffer.

- The blocks need not be the same size.
All to All Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- Extends MPI_ALLGATHER so that each process sends distinct data to each of the receivers.
- The jth block sent from process i is received by process j and stored in the ith block.
Global Reduction Operations

- Perform some operation over a group of processes.
- This operation may be one of the pre-defined operations or a user-specified operation.
- A number of variants are provided to meet communication needs amongst processes.
Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

- Combines the elements provided in the send buffer of each process and returns the result to the root process.
User-defined Operations

MPI_OP_CREATE(function, commute, op)

- Create a handle to a function that will be executed by a global reduction operation.
All Reduce

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

- Same as MPI_REDUCE except that the result appears in all of the processes involved.
Reduce-Scatter

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

- Firstly, conducts a reduce operation.
- Secondly, scatters the result vector to the processes in the specified group.
Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

- Performs a prefix reduction on data distributed across the group.
- Process i receives, in its buffer, the reduction of the inputs provided by processes 0 to i (inclusive).
Dangerous Uses

- Collective operations must be called in the same order in each process lest a deadlock occur.
  - Do not reverse the operations.
  - Beware of cyclic dependencies.
  - Do not interleave blocking, point-to-point operations with collective operations.
  - Do not use a communicator concurrently in the same process that invokes collective operations in multiple threads.