High Performance Computing
Course Notes 2009-2010

Message Passing Programming II
Communications

Point-to-point communications: involving exact two processes, one sender and one receiver

For example, MPI_Send() and MPI_Recv()

Collective communications: involving a group of processes
Collective operations

- coordinated communication operations involving multiple processes

- Programmer could do this by hand (tedious), MPI provides a specialized collective communications
  - *barrier* – synchronize all processes
  - *reduction* operations – sums, multiplies etc. distributed data
  - *broadcast* – sends data from one to all processes
  - *gather* – gathers data from all processes to one process
  - *scatter* – scatters data from one process to all processes

- all executed collectively (by all processes in the group, at the same time, with the same parameters)
Collective operations

MPI_Barrier (comm)

*Global synchronization*

*comm* is the communicator handle

No processes return from function until all processes have called it

Good way of separating one phase from another
Barrier synchronizations

Your program is only as quick as your slowest process

Barrier sync.  Barrier sync.
Collective operations

MPI_Bcast (buf, count, type, root, comm)

**Broadcast data from root to all processes**

- **buf**: address of receiver’s buffer or sender’s buffer (root)
- **count**: no. of entries in buffer (>=0)
- **type**: datatype of buffer elements
- **root**: process id of root process
- **comm**: communicator

![Diagram showing one-to-all broadcast]

**Computer Science, University of Warwick**
Example of MPI_Bcast

Broadcast 100 ints from process 0 to every process in the group

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

**MPI_Gather** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

*Collective data movement function*

- **sendbuf**: address of input buffer
- **sendcount**: no. of elements sent from each (>=0)
- **sendtype**: datatype of input buffer elements
- **recvbuf**: address of output buffer (var param)
- **recvcount**: no. of elements received from each
- **recvtype**: datatype of output buffer elements
- **root**: process id of root process
- **comm**: communicator

Diagram:

<table>
<thead>
<tr>
<th>proc.</th>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All to one gather

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
</tr>
</thead>
</table>

MPI_GATHER
Collective operations

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

Collective data movement function

- sendbuf: address of input buffer
- sendcount: no. of elements sent from each (>=0)
- sendtype: datatype of input buffer elements
- recvbuf: address of output buffer (var param)
- recvcount: no. of elements received from each
- recvtype: datatype of output buffer elements
- root: process id of root process
- comm: communicator

Data flow diagram:
- proc. A₀, A₁, A₂, A₃
- All to one gather
- MPI_GATHER

Computer Science, University of Warwick
Collective operations

**MPI_Gather** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

*Collective data movement function*

- **sendbuf**: address of send buffer
- **sendcount**: no. of elements sent from each (>=0)
- **sendtype**: datatype of send buffer elements
- **recvbuf**: address of recv buffer (var param)
- **recvcount**: no. of elements received from each
- **recvtype**: datatype of recv buffer elements
- **root**: process id of root process
- **comm**: communicator

---

Data flow diagram:

- **proc.**:
  - A₀
  - A₁
  - A₂
  - A₃

- **data** →

- **All to one gather**

- **MPI_GATHER**

- **recvbuf**: A₀, A₁, A₂, A₃
Collective operations

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

Collective data movement function

- **sendbuf**: address of send buffer
- **sendcount**: no. of elements sent from each (>=0)
- **sendtype**: datatype of send buffer elements
- **recvbuf**: address of recv buffer (var param)
- **recvcount**: no. of elements received from each
- **recvtype**: datatype of recv buffer elements
- **root**: process id of root process
- **comm**: communicator

All to one gather

```
<table>
<thead>
<tr>
<th>proc.</th>
<th>data →</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td></td>
</tr>
<tr>
<td>A₁</td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td></td>
</tr>
<tr>
<td>A₃</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Computer Science, University of Warwick
MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

- **sendbuf**: address of send buffer
- **sendcount**: no. of elements sent from each (>=0)
- **sendtype**: datatype of send buffer elements
- **recvbuf**: address of recv buffer (var param) //note the size
- **recvcount**: no. of elements received from each
- **recvtype**: datatype of recv buffer elements
- **root**: process id of root process
- **comm**: communicator

Collective operations

- All to one gather
  ```
  proc.  | data →
  A₀    | A₁    | A₂    | A₃
  A₁    |       |       |     
  A₂    |       |       |     
  A₃    |       |       |     

  MPI_GATHER
  ```
Collective operations

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

Collective data movement function

- **sendbuf**: address of send buffer
- **sendcount**: no. of elements sent from each (>=0)
- **sendtype**: datatype of send buffer elements
- **recvbuf**: address of recv buffer (var param)
- **recvcount**: no. of elements received from each
- **recvtype**: datatype of recv buffer elements
- **root**: process id of root process
- **comm**: communicator

proc.  
\[ \begin{array}{c}
A_0 \\
A_1 \\
A_2 \\
A_3 \\
\end{array} \]  

All to one gather

\[ \begin{array}{c}
A_0 \quad A_1 \quad A_2 \quad A_3 \\
\end{array} \]  

MPI_GATHER

Computer Science, University of Warwick
**MPI_Gather example**

Gather 100 ints from every process in group to root

```c
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank(comm, myrank); // find proc. id
If (myrank == root) {
    MPI_Comm_size(comm, &gsize); // find group size
    rbuf = (int *) malloc(gsize*100*sizeof(int)); // calc. receive buffer
}
// MPI_Gather is run by all processes at the same time
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```
Collective operations

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

Collective data movement function

- **sendbuf**: address of send buffer //note buff size
- **sendcount**: no. of elements sent to each (>=0)
- **sendtype**: datatype of send buffer elements
- **recvbuf**: address of recv buffer
- **recvcount**: no. of elements received by each
- **recvtype**: datatype of recv buffer elements
- **root**: process id of root process
- **comm**: communicator

![Diagram](image)
Example of MPI_Scatter

MPI_Scatter is reverse of MPI_Gather

MPI_Comm comm;
    int gsize, *sendbuf;
    int root, rbuf[100];
...
    MPI_Comm_rank( comm, myrank); // find proc. id
If (myrank == root) {
    MPI_Comm_size (comm, &gsize);
    sendbuf = (int *) malloc (gsize*100*sizeof(int));
}
...
    MPI_Scatter (sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

It is as if the root sends the data in sendbuf using

MPI_Send(sendbuf+i*sendcount*sizeof(MPI_INT), sendcount, sendtype, pid_i, ...)

pid_i is the process id of the i-th process
Collective operations

MPI_Reduce (sendbuf, recvbuf, count, type, op, root, comm)

Collective reduction function

- sendbuf: address of send buffer
- recvbuf: address of recv buffer
- count: no. of elements in input buffer (>=0)
- type: datatype of send buffer elements
- op: operation
- root: process id of root process
- comm: communicator

Using MPI_MIN
Root = 0

MPI_REDUCE
Collective operations

MPI_Reduce (sendbuf, recvbuf, count, type, op, root, comm)

Collective reduction function

- **sendbuf**: address of send buffer
- **recvbuf**: address of recv buffer
- **count**: no. of elements in input buffer (>=0)
- **type**: datatype of input buffer elements
- **op**: operation
- **root**: process id of root process
- **comm**: communicator

Using MPI_SUM
Root = 1

**Using MPI_REDUCE**

<table>
<thead>
<tr>
<th>proc.</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

13 16
Collective operations

MPI_Allreduce (sendbuf, recvbuf, count, type, op, comm)

Collective reduction function

- **sendbuf**: address of send buffer
- **recvbuf**: address of recv buffer
- **count**: no. of elements in input buffer (>=0)
- **type**: datatype of input buffer elements
- **op**: operation
- **comm**: communicator

```
<table>
<thead>
<tr>
<th>proc.</th>
<th>data</th>
<th>0</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Using MPI_MIN

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

Using MPI_ALLREDUCE

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```
Buffering in MPI communications

→ Application buffer: specified by the first parameter in MPI_Send/Recv functions

→ System buffer:
  - Hidden from the programmer and managed by the MPI library
  - Is limited and can be easy to exhaust

MPI_Send (buf, count, datatype, dest, tag, comm)
Blocking and non-blocking communications

→ Blocking send
  - The sender doesn’t return until the application buffer can be re-used (which often means that the data have been copied from application buffer to system buffer)  
    //note: it doesn’t mean that the data will be received

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

→ Blocking receive
  - The receiver doesn’t return until the data have been ready to use by the receiver (which often means that the data have been copied from system buffer to application buffer)

→ Non-blocking send/receive
  - The calling process returns immediately
  - Just request the MPI library to perform the operation, the user cannot predict when that will happen
  - Unsafe to modify the application buffer until you can make sure the requested operation has been performed (MPI provides routines to test this)
  - Can be used to overlap computation with communication and have possible performance gains

  \[\text{MPI\_Isend (buf, count, datatype, dest, tag, comm, request)}\]
Testing non-blocking communications

→ Completion tests come in two types:
  - WAIT type
  - TEST type

→ WAIT type: the WAIT type testing routines block until the communication has been completed.
  - A non-blocking communication immediately followed by a WAIT-type test is equivalent to the corresponding blocking communication

→ TEST type: these testing routines return immediately with a TRUE or FALSE value
  - The process can perform some other tasks if the communication has not completed
Testing non-blocking communications for completion

The WAIT-type test is:

MPI_Wait (request, status)

This routine blocks until the communication specified by the request handle has completed. The request handle will have been returned by an earlier call to a non-blocking communication routine.

The TEST-type test is:

MPI_Test (request, flag, status)

In this case the communication specified by the handle request is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned into flag.
Testing multiple non-blocking communications for completion

Wait for all communications to complete

**MPI_Waitall (count, array_of_requests, array_of_statuses)**

This routine blocks until all the communications specified by the request handles, array_of_requests, have completed. The statuses of the communications are returned in the array array_of_statuses and each can be queried in the usual way for the source and tag if required.

Test if all communications have completed

**MPI_Testall (count, array_of_requests, flag, array_of_statuses)**

If all the communications have completed, flag is set to TRUE, and information about each of the communications is returned in array_of_statuses. Otherwise flag is set to FALSE and array_of_statuses is undefined.
Testing multiple non-blocking communications for completion

Query a number of communications at a time to find out if any of them have completed.

Wait: `MPI_Waitany (count, array_of_requests, index, status)`

- `MPI_Waitany` blocks until one or more of the communications associated with the array of request handles, `array_of_requests`, has completed.

- The index of the completed communication in the `array_of_requests` handles is returned in `index`, and its status is returned in `status`.

- Should more than one communication have completed, the choice of which is returned is arbitrary.

Test: `MPI_Testany (count, array_of_requests, index, flag, status)`

- The result of the test (TRUE or FALSE) is returned immediately in `flag`.
Communication modes

→ Synchronous mode

- The communication is considered complete when the sender receives the acknowledgement from the receiver that the data have been received.

→ Buffered mode

- The sender uses the explicitly defined buffer instead of system buffer (the system buffer is limited).
- Communication is considered complete when the application buffer can be re-used, which means that the data has been copied from the application buffer to the user-defined buffer.

→ Ready mode

- This mode can be used only when the programmer can make sure that the receive routine will be posted before the corresponding send routine, otherwise, the outcome is undefined.

→ Standard mode

- The system may or may not buffer the messages depending on the implementations.
Blocking and non-blocking forms for the communication modes

All these four communication modes have both blocking and non-blocking forms

The communication modes refers to the send routines

Standard send: MPI_Send (blocking), MPI_Isend (non-blocking)

Synchronous send: MPI_Ssend (blocking), MPI_Issend (non-blocking)

Buffered send: MPI_Bsend (blocking), MPI_Ibsend (non-blocking)

Ready send: MPI_Rsend (blocking), MPI_Irsend (non-blocking)