High Performance Computing
Course Notes 2008-2009

Message Passing Programming II
Modularity

- MPI supports modular programming via *communicators*
- Provides information hiding by encapsulating local communications and having local namespaces for processes
- All MPI communication operations specify a communicator (process group that is engaged in the communication)
Forming new communicators – one approach

MPI_Comm world, workers;
MPI_Group world_group, worker_group;
int ranks[1];
MPI_Init(&argc, &argv);
world=MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server=numprocs-1;
MPI_Comm_group(world, &world_group);
ranks[0]=server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&world_group);
MPI_Comm_free(&workers);
Forming new communicators - functions

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

Int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Group_free(MPI_Group *group)

int MPI_Comm_free(MPI_Comm *comm)
Forming new communicators – another approach (1)

MPI_Comm_split (comm, colour, key, newcomm)

*Creates one or more new communicators from the original comm*

- **comm**: communicator (handle)
- **colour**: control of subset assignment (processes with same colour are in same new communicator)
- **key**: control of rank assignment
- **newcomm**: new communicator

Is a *collective* communication operation (must be executed by all processes in the *comm*)

Is used to (re-) allocate processes to communicator (groups)
Forming new communicators – another approach (2)

MPI_Comm_split (comm, colour, key, newcomm)

MPI_Comm comm, newcomm; int myid, color;
MPI_Comm_rank(comm, &myid); // id of current process
color = myid%3;
MPI_Comm_split(comm, colour, myid, *newcomm);
Forming new communicators – another approach (3)

MPI_Comm_split (comm, colour, key, newcomm)

- New communicator created for each new value of colour
- Each new communicator (sub-group) comprises those processes that specify its value in colour
- These processes are assigned new identifiers (ranks, starting at zero) with the order determined by the value of key (or by their ranks in the old communicator in event of ties)
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

- i.e. coordinated communication operations involving multiple processes

- Programmer could do this by hand (tedious), MPI provides a specialized collective communications
  - barrier – synchronize all processes
  - 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
  - reduction operations – sums, multiplies etc. distributed data

- 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

You are only as quick as your slowest process
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

```
<table>
<thead>
<tr>
<th>proc</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One to all broadcast

MPI_BCAST

```

```A0 A0 A0 A0
```
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**:

- Data flow from processes A0, A1, A2, A3
- All to one gather
- MPI_GATHER function

[Diagram showing data movement from processes to a root process]

Computer Science, University of Warwick

14
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:
- **All to one gather**
- Data from processes A0, A1, A2, A3 is gathered to process A0 using MPI_GATHER

Diagram:
- Process grid from A0 to A3
- Arrows indicating data flow
- Process A0 collects data from A1, A2, A3

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

All to one gather

```plaintext
A_0  A_1  A_2  A_3
```

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

**Diagram:**
- Data flow from processes A₀, A₁, A₂, A₃ to a single process with MPI_GATHER function.
**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( 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
- **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

In the diagram:
- Data moves from process `A0`, `A1`, `A2`, `A3` to their respective receiving processes.
- `MPI_SCATTER` function is used for the collective scatter operation.
Example of MPI_Scatter

MPI_Scatter is reverse of MPI_Gather

It is as if the root sends using

\[
\text{MPI\_Send}(\text{sendbuf}+i*\text{sendcount}*\text{sizeof}(\text{intype}), \text{sendcount}, \text{sendtype}, \text{pid}_i, \ldots)
\]

\text{pid}_i \text{ is the process id of the i-th process}

\begin{verbatim}
MPI_Comm comm;
int gsize, *sendbuf;
int root, rbuff[100];
...
MPI_Comm_size (comm, &gsize);
sendbuf = (int *) malloc (gsize*100*sizeof(int));
...
MPI_Scatter (sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
\end{verbatim}
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

0 2

MPI_REDUCE
Collective operations

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

Collective reduction function

- `sendbuf`: address of sender’s buffer
- `recvbuf`: address of receiver’s 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

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

Using MPI_REDUCE

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Computer Science, University of Warwick
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

Data:

- Using `MPI_MIN`:
  - 0 2
  - 0 2
  - 0 2

- `MPI_ALLREDUCE`:
  - 0 2
  - 0 2
  - 0 2

Computer Science, University of Warwick
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

```c
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), but 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.