Parallel Programming with MPI

Lars Koesterke
04/01/2014: HPC Workshop at ASU
04/04/2014: HPC Workshop at CSU
Outline

• Message Passing Overview

• Compiling and running MPI programs

• Point-to-Point Communication

• Collective Communication
OVERVIEW
Message Passing Overview

• What is message passing?

Simple Answer: The sending and receiving of messages between diverse computational resources.
Message Passing Overview

• Messages may be used for
  – sending data
  – performing operations on data
  – synchronization between tasks

• Why do we need to send/receive messages?
  – On clusters, each node has its own address space, and no way to get at another’s, except over the network
Message Passing Model

- Tasks send and receive messages to exchange data.
- Data transfer requires a cooperative operation to be performed by each process.
- The programmer is responsible for determining all parallelism.
- Message Passing Interface (MPI) was first released in 1994. (MPI-2 in 1996.)
- MPI is the de facto standard for message passing.

http://www-unix.mcs.anl.gov/mpi/
What is MPI?

• MPI is
  – An acronym for Message Passing Interface
  – A standard Application Programming Interface (API)

• MPI is not
  – A language
  – An implementation
  – Specific to a particular machine
MPI Fundamentals

- Subsets of functionality
  - basic (about 6 functions)
  - intermediate
  - advanced (up to 125 functions)

- One goal of MPI is to provide access to advanced parallel hardware for application scientists (not just programmers and computer scientists)

- Many high-level application libraries are based on MPI
  - PETSc
  - SAMRAI
  - Cactus
  - FFTW
  - PLAPACK
Why learn MPI?

• MPI is a standard
  – Public domain version easy to install
  – Vendor-optimized version available on most communication hardware

• MPI applications are portable.

• MPI is expressive: MPI can be used for many different models of computation, therefore can be used with many different applications.

• MPI is a good way to learn the theory of parallel computing.
Compiling MPI Programs

• Building simple MPI programs, using MPICH
  % mpicc  -o first  first.c
  % mpif90  -o firstf firstf.f  (also mpif77)

• These are simply shell script wrappers for system compilers.

• Some MPI specific compiler options
  – -mpilog  : Generate log files of MPI calls
  – -mpitrace  : Trace execution of MPI calls
Compiling MPI Programs

• The names of the mpiXXX compiler scripts are not specified by the MPI standard.
• Examples:
  – IBM: mpcc_r, mpxlf_r
  – Kraken (A Cray system in the Teragrid): cc, and ftn
Running MPI Programs

• To run a simple MPI program using MPICH
  % mpirun -np 2 <programe>

• Some MPI specific running options
  – -t : shows the commands that mpirun would execute
  – -help : shows all options for mpirun

• The name “mpirun” is not part of the standard, other names include
  – IBM SP: poe
  – Lonestar/Stampede: ibrun
  – Mpich2: mpiexec
MPI BASICS
Outline

• Basic MPI code structure

• Point-to-point communication

• Collective communication
MPI Initialization & Termination

• All processes must initialize and finalize MPI (each is a collective call).
  - **MPI_Init**: starts up the MPI runtime environment
  - **MPI_Finalize**: shuts down the MPI runtime environment

• Must include header files – provides basic MPI definitions and types.
  - Header File

<table>
<thead>
<tr>
<th>Fortran 77</th>
<th>Fortran 90</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>include ‘mpif.h’</td>
<td>use mpi</td>
<td>#include “mpi.h”</td>
</tr>
</tbody>
</table>

• Format of MPI calls

<table>
<thead>
<tr>
<th>Fortran 77/90 binding</th>
<th>C/C++ binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MPI_XYYY(parameters..., ierr)</td>
<td>ierr = MPI_Xyy(parameters...)</td>
</tr>
</tbody>
</table>
Communicators

- MPI uses **MPI_Comm** objects to define subsets of processors which may communicate with one another.

- Most MPI routines require you to specify a communicator as an argument.

- Default communicator: **MPI_COMM_WORLD** – a predefined communicator that includes all of your processes.

- In MPI terminology, a processor’s “Rank” is:
  - A unique ID within a communicator
  - Assigned by the system when the communicator is created
  - Part of a contiguous integer range which begins at zero
  - The way one specifies the source and destination of messages
Communicators

- Two common functions for interacting with an MPI_Comm object are:
  - `MPI_Comm_size(MPI_Comm_World, int *np)`
    - Gets the number of processes in a run, \( NP \)
  - `MPI_Comm_rank(MPI_Comm_World, int *rank)`
    - Gets the rank of the current process
    - returns a value between 0 and \( NP-1 \) inclusive

- Both are typically called just after `MPI_Init`. 
Sample MPI code (C)

```c
#include <mpi.h>
[other includes]

int main(int argc, char **argv){
    int ierr, np, rank;
    [other declarations]

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    :
    [actual work goes here]
    :
    MPI_Finalize();
}
```
Sample MPI code (C++)

```cpp
#include <mpi.h>
[other includes]
int main(int argc, char **argv){
    int np, rank;
    [other declarations]
    :
    MPI::Init(argc, argv);
    np  = MPI::COMM_WORLD.Get_size();
    rank= MPI::COMM_WORLD.Get_rank();
    :
    [actual work goes here]
    :
    MPI::Finalize();
}
```
Sample MPI code (F90)

```fortran
program samplempi
    use mpi
    [other includes]

    integer :: ierr, np, rank
    [other declarations]

    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, np, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, rank, ierr)
    :
    [actual work goes here]
    :
    call mpi_finalize(ierr)
end program
```
MPI Execution

• Every process gets a copy of the executable: *Single Program, Multiple Data* (SPMD).

• They all start executing it.

• Each looks at its own rank to determine which part of the problem to work on.

• Each process works *completely independently* of the other processes, except when communicating.
POINT-TO-POINT COMMUNICATION
Point-to-Point Communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives
P2P Communication: Send

MPI_Send(void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>initial address of send/receive buffer</td>
</tr>
<tr>
<td>count</td>
<td>number of items to send</td>
</tr>
<tr>
<td>datatype</td>
<td>MPI data type of items to send/receive</td>
</tr>
<tr>
<td>dest</td>
<td>MPI rank or task receiving the data</td>
</tr>
<tr>
<td>tag</td>
<td>message ID</td>
</tr>
<tr>
<td>comm</td>
<td>MPI communicator where the exchange occurs</td>
</tr>
</tbody>
</table>
P2P Communication: Receive

MPI_Recv(void *buf,
           int count,
           MPI_Datatype datatype,
           int source,
           int tag,
           MPI_Comm comm,
           MPI_Status *status)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>initial address of send/receive buffer</td>
</tr>
<tr>
<td>count</td>
<td>number of items to send</td>
</tr>
<tr>
<td>datatype</td>
<td>MPI data type of items to send/receive</td>
</tr>
<tr>
<td>source</td>
<td>MPI rank of task sending the data</td>
</tr>
<tr>
<td>tag</td>
<td>message ID</td>
</tr>
<tr>
<td>comm</td>
<td>MPI communicator where the exchange occurs</td>
</tr>
<tr>
<td>status</td>
<td>returns information on the message received</td>
</tr>
</tbody>
</table>
Summary: MPI_Send & MPI_Recv

MPI_Send(buf, count, datatype, dest, tag, comm);
MPI_Recv(buf, count, datatype, source, tag, comm, status);

• In the status object, the system can return details about the message received. Can pass default MPI_STATUS_IGNORE object instead.

• These calls are “blocking”

• This means that program flow does not return to the calling function until the send/recv pair is completed.

• Can lead to a condition known as “deadlock” in case a Send is not paired with a matching Receive.
MPI_SendRecv

MPI_SendRecv(/*send arguments*/
    sendbuf, sendcount, sendtype, 
    dest, sendtag,
/*receive arguments*/
    recvbuf, recvcount, recvtype, source, 
    recvtag, comm, status);

• Union of MPI_Send and MPI_Recv commands
• Executes a **blocking** send & receive operation
• Send and Receive stages use the same communicator, but have distinct tags
• Useful for communications patterns where each node both sends and receives messages (two-way communication)
A quick overview of MPI's send modes

MPI has a number of different "send modes." These represent different choices of **buffering** (where is the data kept until it is received) and **synchronization** (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

- **MPI_Send**: will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- **MPI_Bsend**: May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- **MPI_Ssend**: will not return until matching receive posted
- **MPI_Rsend**: May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- **MPI_Isend**: Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to *immediate*, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.

- And then there is: **MPI_Ibsend**, **MPI_Issend**, and **MPI_Irsend**
A quick overview of MPI's send modes

Recommendations

- The best performance is likely if you can write your program so that you could use just MPI_Ssend; in that case, an MPI implementation can completely avoid buffering data. Use MPI_Send instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data. (Unfortunately, one vendor has chosen to have MPI_Send emphasize buffering over performance; on that system, MPI_Ssend may perform better.) If nonblocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvenient to use MPI_Isend. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

Synchronous Communication

- Process 0 waits until process 1 is ready
- *Handshaking* occurs between send & receive tasks to confirm a safe send.
- Blocking send/receive
- This is rarely the case in real world.
- Need to be able to deal with storing data when multiple tasks are out of sync.
- MPI_Ssend & MPI_Srecv
Buffered Communication

- The contents of the message is copied into a system-controlled block of memory (system buffer).
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the buffered message into the appropriate memory location controlled by process 1.
- MPI_Bsend & MPI_Brecv
Blocking vs. Non-blocking

**Blocking**
- A blocking send routine will only return after it is *safe* to modify the buffer.
- *Safe* means that modification will not affect the data to be sent.
- *Safe* does not imply that the data was actually received.
- MPI_Send, MPI_Recv

**Non-blocking**
- Send/receive routines return immediately.
- Non-blocking operations request that the MPI library perform the operation “when possible”.
- It is *unsafe* to modify the buffer until the requested operation has been performed. There are *wait* routines used to do this (MPI_Wait).
- Primarily used to overlap computation with communication.
- MPI_Isend, MPI_Irecv
# Blocking/Non-Blocking Routines

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax for C bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking send</td>
<td><code>MPI_Send(buf, count, datatype, dest, tag, comm)</code></td>
</tr>
<tr>
<td>Blocking receive</td>
<td><code>MPI_Recv(buf, count, datatype, source, tag, comm, status)</code></td>
</tr>
<tr>
<td>Non-blocking send</td>
<td><code>MPI_Isend(buf, count, datatype, dest, tag, comm, request)</code></td>
</tr>
<tr>
<td>Non-blocking receive</td>
<td><code>MPI_Irecv(buf, count, datatype, source, tag, comm, request)</code></td>
</tr>
</tbody>
</table>

- **MPI_Request** objects are used by non-blocking send and receive calls.
- **MPI_Wait()/MPI_Waitall()**
  - Blocking functions
  - Pause program execution until outstanding Isend/Irecv calls have completed.
Send/Recv Pairs in Code

• **Blocking Send & Blocking Recv**

```plaintext
IF (rank==0) THEN
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
```

• **Non-blocking Send & Blocking Recv**

```plaintext
IF (rank==0) THEN
   CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ierr)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
CALL MPI_WAIT(req, wait_status)
```
Deadlock Example

! The following code contains a deadlock... can you spot it?
IF (rank==0) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
  CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
  CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF

! Solution
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
  CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
  CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF
Alternative Deadlock Solutions

! Solution using sendrecv
IF (rank==0) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, sendtag, 
                   recvbuf, count, MPI_REAL, 1, recvtag, 
                   MPI_COMM_WORLD, status, ierr)
ELSEIF (rank==1) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, sendtag, 
                   recvbuf, count, MPI_REAL, 0, recvtag, 
                   MPI_COMM_WORLD, status, ierr)
ENDIF

! Another possible solution (using all non-blocking calls)
IF (rank==0) THEN
  CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req1,ierr)
  CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req2,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_ISEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
  CALL MPI_IRECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req2,ierr)
ENDIF
CALL MPI_WAIT(req1, wait_status, ierr)
CALL MPI_WAIT(req2, wait_status, ierr)
COLLECTIVE COMMUNICATION
Collective Communication

- Defined as communication between > 2 processors
  - One-to-many
  - Many-to-one
  - Many-to-many

- A collective operation requires that all processes within the communicator group call the same collective communication function with matching arguments.
Naïve collective communication

- With MPI_Send/Recv, you have all the tools...

```c
if (rank == 0) {
    for (int id=1; id<np; id++) {
        MPI_Send( ..., /* dest= */ id, ... );
    }
} else {
    MPI_Recv( ..., /* source= */ 0, ... );
}
```

- This code “broadcasts” information from proc. 0 to all other processors, but:
  - It’s too primitive: no room for the OS/hardware to optimize
  - It uses blocking communications (deadlocks!)
Broadcast Implementation

Provided by MPI

Naïve Implementation
The Basics of Collective Communications

• Involve **ALL** processes within a communicator.
• It is the programmer’s responsibility to ensure that all processors call the **same** collective communication at the same time.
• Type of collective operations
  – **Synchronization** (MPI_Barrier)
  – **Data Movement** (MPI_Bcast/Scatter/Gather/Allgather/AlltoAll)
  – **Computation** (MPI_Reduce/Allreduce/Scan)
• Programming considerations & restrictions
  – Collective Communications are **blocking** operations
  – Collective operations on subsets of processes require separate grouping/new communicators
  – The size of data sent must exactly match the size of data received, not the case for P2P communications
Collective Operation Visualization

- **Broadcast**: Data from a root task is distributed to all tasks.
- **Gather/Scatter**: Elements from all tasks are combined into a root task or distributed from a root task to all tasks.
- **Allgather**: Elements from all tasks are combined into all tasks.
- **Alltoall**: Elements from all tasks are distributed to all tasks.

**Tasks**: 
- **A0**, **A1**, **A2**, **A3**
- **B0**, **B1**, **B2**, **B3**
- **C0**, **C1**, **C2**, **C3**
- **D0**, **D1**, **D2**, **D3**

**Data**: 
- **Root to all tasks**
- **All tasks to root**
- **All tasks to all tasks**

TACC
Texas Advanced Computing Center
Barrier

**MPI_BARRIER**(comm, ierr)

| IN  | comm | Communicator |

- Each task waits in the Barrier until all tasks in the communicator have reached it.
- Can be used when measuring communication/computation time and for debugging.
- Caution must be exercised to avoid over-synchronization: This will unnecessarily slow down program execution.
Broadcast

MPI_BCAST(buf, count, datatype, root, comm, ierr)

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INOUT</td>
<td>buf</td>
<td>starting address of buffer</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of entries in buffer</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>data type of buffer</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
<td>rank of broadcast root</td>
</tr>
<tr>
<td>IN</td>
<td>Comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>

- Broadcast a message from the process with rank “root” to all the processes in the group (in the communicator).
**MPI_Scatter**

\[
MPI_{\text{SCATTER}}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, \text{comm})
\]

<table>
<thead>
<tr>
<th>IN</th>
<th>sendbuf</th>
<th>address of send buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements sent to each process</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>IN</td>
<td>Recvcount</td>
<td>number of elements in receive buffer</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
<td>rank of sending process</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>

- The root process divides its send buffer into \( n \) equal segments and sends.
- Each process receives a segment from the root and places it in its receive buffer.
- The reverse of MPI_Gather
MPI_Gather

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

- Each process sends the contents of its send buffer to the root process.
- Root stores them in its receive buffer according to the ranks of the senders.
- The reverse of MPI_Scatter
MPI_Allgather

**MPI_ALLGATHER**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sendbuf</code></td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td><code>sendcount</code></td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td><code>sendtype</code></td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td><code>recvbuf</code></td>
<td>address of receive buffer</td>
</tr>
<tr>
<td><code>recvcount</code></td>
<td>number of elements received from any process</td>
</tr>
<tr>
<td><code>recvtype</code></td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td><code>comm</code></td>
<td>communicator</td>
</tr>
</tbody>
</table>

- An MPI_Gather whose result ends up on all processors.
- Each task in the group, in effect, performs a one-to-one broadcasting operation within the group.
## MPI_Alltoall

**MPI_ALLTOALL**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements sent to each process</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>IN</td>
<td>recvcount</td>
<td>number of elements received from any process</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>IN</td>
<td>Comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>

- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order, by index.
MPI_Reduce

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

- sendbuf: address of send buffer
- recvbuf: address of receive buffer on root process
- count: number of elements in send buffer
- datatype: data type of elements of send buffer
- op: reduce operation
- root: rank of root process
- comm: communicator

- Applies a reduction operation on all tasks in the group and places the result in the receive buffer on the root process.
- Possible operations are MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, ...
MPI_Allreduce

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

- Applies a reduction operation and places the result in all tasks in the group.
- Equivalent to an MPI_Reduce followed by MPI_Bcast.
# Full List of Reduction Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical And</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise And</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical Or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise Or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical Xor</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bit-wise Xor</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Max value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Min value and location</td>
</tr>
</tbody>
</table>
Collective Communication: Summary

- **Broadcast**: Data is sent from one process to all other processes.
  - Process 0 sends to P1, P2, P3.
  - Process 1 sends to P0, P2, P3.
  - Process 2 sends to P0, P1, P3.
  - Process 3 sends to P0, P1, P2.

- **Reduce**: Data is combined across all processes.
  - Each process sends its data to a central process (reducer).
  - Central process combines data from all processes.

- **Scatter**: Data is distributed to processes.
  - Central process sends data to all processes.
  - Each process receives data from the central process.

- **Gather**: Data is collected from all processes.
  - Each process sends its data to a central process.
  - Central process collects data from all processes.

- **Allgather**: Data is distributed to all processes.
  - Central process sends data to all processes.
  - Each process receives data from the central process.

- **Alltoall**: Data is distributed to all processes.
  - Each process sends data to all other processes.
  - Each process receives data from all other processes.

- **Scan**: Data is accumulated in all processes.
  - Each process accumulates data from previous processes.

- **Reduce-scatter**: Data is reduced and then scattered.
  - Data is combined across all processes.
  - Combined data is scattered to all processes.
Final Thoughts

• Before jumping headfirst into MPI, pause and consider
  – Are you implementing something others have already implemented and put into libraries
  – Should you re-use code or develop new code from the ground up
  – Map the development cycle and requirements, make sure MPI is the right answer for you

• If MPI is the right answer
  – Survey existing tools, libraries and functions, code re-use saves time, effort and frustration
  – Map application requirements, procedures and algorithm to existing parallel/distributed algorithms, avoid reinventing the wheel
  – Include “smart” debugging statements in your code, ideally some form of tracing to be able to track how things go wrong (because chances are, they will go wrong)
Thank you very much

lars@tacc.utexas.edu

Please participate in our survey