MPI Collective Communication I

Michaela Lechner, PDC Center for HPC

based on work by Michael Hanke

Summer School on High Performance Computing

Outline

What You Already Know

- How to start and stop MPI
- How to exchange messages between pairs of processes
- How to avoid deadlocks
- How to use communication routines efficiently

What Is Covered in This Module?

- Collective communication: many processes are exchanging messages with each other simultaneously.
- Different types of communication patterns:
  - Process synchronization
  - Data movement
  - Global computations
- Performance issues

Collective Communication

- An MPI call made identically by all processes in process group
- Supports easy manipulation of a "common" piece or set of information
- Uses an MPI communicator defined for the process group

MPI Collective Communication Routines

- Barrier synchronization
- Broadcast from one member to all other members
- Gather data from an array spread across processors into one array
- Scatter data from one member to all members
- All-to-all exchange of data
- Global reduction (e.g., SUM, MIN of "common" data elements)
- Scan across all members of a communicator

Characteristics

- Collective Communication performes on a group of processes, identified by a communicator.
- Works as a substitute for a sequence of point-to-point calls.
- Communications are locally blocking.
- Synchronization is not guaranteed (implementation dependent).
- Some routines use a root process to originate or receive all data.
- Data amounts must exactly match.
- Many variations to basic categories exist.
- No message tags are needed.

Barrier Synchronization

Blocks the calling process until all group members have called the routine.

C

```
MPI_Barrier(MPI_comm comm)
```

Fortran

```
MPI_BARRIER(comm, ierr)
```

Data Movement Routines

- Broadcast
- Gather and Gatherv
- Scatter and Scatterv
- Allgather and Allgatherv
- Alltoall and Alltoallv

Broadcast

Sends a message from the root process to all processes in the group, including the root process.

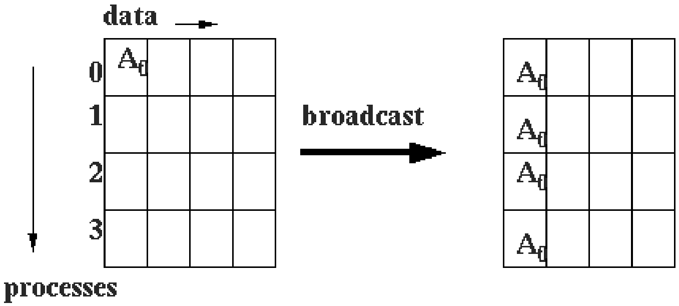
C

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

Fortran

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

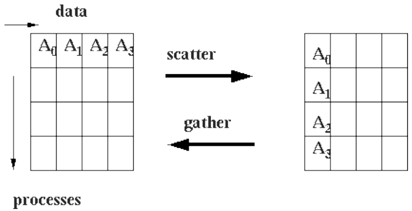
Broadcast: Effect



Example: Hello World

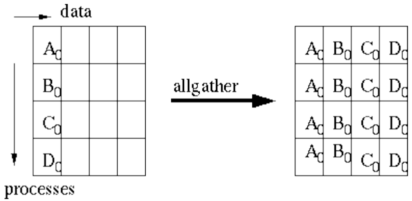
```
!  Prepare MPI ...
 if (rank .eq. 0) then
   message = 'Hello, world'
 endif

 call MPI_BCAST(message, 12, MPI_CHARACTER, root,
&               MPI_COMM_WORLD, ierr)




 print*, 'node', rank, ':', message
!  Finalize MPI ...
```

MPI_Gather And MPI_Scatter



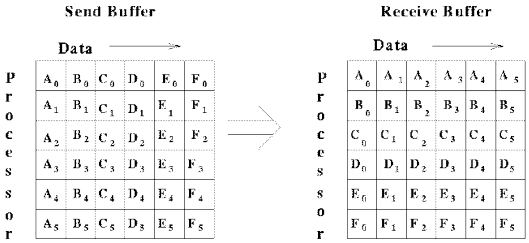
- Gather Purpose:
    - Each process sends the contents of its send buffer to the root process.
    - The root process receives the messages and stores them in rank order.
- Scatter Purpose:
    - The root process splits a buffer of data into chunks, then sends each process in the group 1 chunk.
    - Reverse of GATHER operation

`MPI_Allgather`



- Same as GATHER, except that all processes, not just root, receive the result.
- Data is stored in rank order.

MPI_Alltoall



Send Buffer:

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Receive Buffer:

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

- Extension to ALLGATHER
- Each process sends distinct data to each receiver.
- Process i sends its jth block of data to process j.
- Process j stores data from process i in its ith block.

Data Movement: Final Remarks

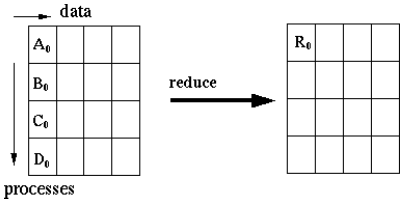
- All routines provided so far require equal chunk sizes.
- There are so-called varying data size versions (with the exception of `MPI_Bcast`, of course) available.
- The varying data size versions have a letter `v` appended to its names, e.g. `MPI_Gather` becomes `MPI_Gatherv`.

Global Computation Routines

- Communication routines that include a computation
- Computation function included in routine call may be either
    - An MPI predefined routine or
    - A user-supplied function
- Two types of global computation routines:
    - Reduce
    - Scan

MPI_Reduce

- Combines the elements of the input buffer of each process using a specified operation.
- Returns result to root process.



MPI_Reduce (cont)

C

```
int MPI_Reduce(void* sbuf, void* rbuf, int count,
               MPI_Datatype stype, MPI_Op op,
               int root, MPI_Comm comm)
```

Fortran

```
MPI_REDUCE(sbuf, rbuf, count, stype, op,
           root, comm, ierr)
```

Predefined Operators

Name	Meaning
`MPI_MAX`	maximum value
`MPI_MIN`	minimum value
`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

- Each of these operations makes sense for only certain datatypes.
- The MPI Standard lists the types accepted for each operation.

Example: MPI_Reduce

- Each process in a forest dynamics simulation calculates the maximum tree height for its region.
- Process 0, which is writing output, must know the global maximum height.

```
INTEGER maxht, globmx
.
. (calculations which determine maximum height)
.
CALL MPI_REDUCE (maxht, globmx, 1, MPI_INTEGER,
                 MPI_MAX, 0, MPI_COMM_WORLD, ierr)
IF (myrank .EQ. 0) THEN
.
. (Write output)
.
ENDIF
```

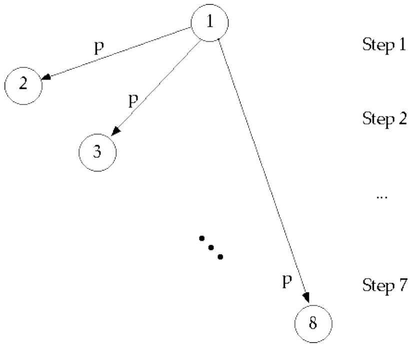
Remarks

- User-defined Operations:
    - Users can define their own reduce operations.
    - Reduce operations are defined by the `MPI_Op_create` function.
- Reduce Variations:
    - `MPI_Allreduce` allows the result to appear in the receive buffers of all processes in the group.
    - `MPI_Reduce_scatter` scatters a vector, which results from a reduce operation, across all processes.
    - `MPI_Scan` performs a partial reduction in which process i receives data from processes 0 through i, inclusive. (*prefix operation*)

Performance Considerations

- A great deal of hidden communication takes place with collective communication.
- Performance depends greatly on the particular implementation of MPI.
- Because there may be forced synchronization, not always best to use collective communication!

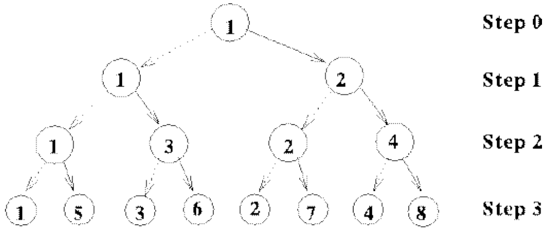
Example: Broadcast To 8 Processes (Simple Approach)



For N processes and a message length of p:

- Amount of data transferred: $(N-1)p$
- Number of steps: $N-1$

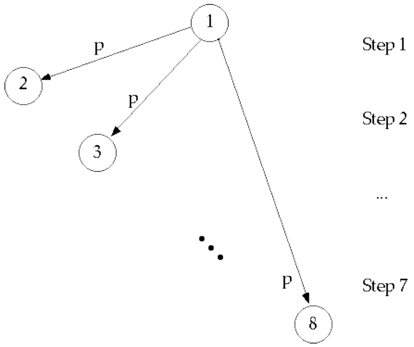
Example: A Better Approach *(Recursive Doubling)*



For N processes and a message length of p:

- Dottet lines indicate "virtual transfers"
- Amount of data transferred: $(N-1)p$
- Number of steps: $\log_2 N$

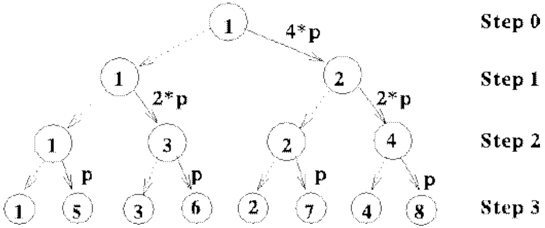
Example: Scatter To 8 Processes



For N processes and a message length of p:

- Amount of data transferred: $(N-1)p$
- Number of steps: $N-1$

Example: A Better Approach



For N processes and a message length of p:

- Dottet lines indicate “virtual transfers”
- Amount of data transferred: $\log_2(N)Np/2$
- Number of steps: $\log_2 N$

Summary

- Collective communication routines provide convenient calls for standard communication patterns.
- Depending on the implementation, they may be much more efficient than hand-coding (or not).
- Collective communication makes extensive use of communicators.

When trying to tune your program to a certain machine/MPI combination, read carefully the accompanying documentation!

- What comes next?
    - Virtual topologies
    - Outlook