Lecture 29: Collective Communication and Computation in MPI

William Gropp

www.cs.illinois.edu/~wgropp
Collective Communication

- All communication in MPI is within a group of processes
- Collective communication is over all of the processes in that group
- MPI_COMM_WORLD defines all of the processes when the parallel job starts
- Can define other subsets
  - With MPI dynamic processes, can also create sets bigger than MPI_COMM_WORLD
  - Dynamic processes not supported on most massively parallel systems
Collective Communication as a Programming Model

- Programs using only collective communication can be easier to understand
  - Every program does roughly the same thing
  - No “strange” communication patterns

- Algorithms for collective communication are subtle, tricky
  - Encourages use of communication algorithms devised by experts
A Simple Example: Computing pi

**MPI_Bcast**(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += f(x);
}

mypi = h * sum;

**MPI_Reduce**(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
Notes on Program

• MPI_Bcast is a “one-to-all” communication
  ◆ Sends value of “n” to all processes
• MPI_Reduce is an “all-to-one” computation, with an operation (sum, represented as MPI_SUM) used to combine (reduce) the data
• Works with any number of processes, even one.
  ◆ Avoids any specific communication pattern, selection of ranks, process topology
Communication and computation is coordinated among a group of processes in a communicator.

Groups and communicators can be constructed “by hand” or using topology routines.

Non-blocking versions of collective operations added in MPI-3

Three classes of operations: synchronization, data movement, collective computation.
Synchronization

• **MPI_Barrier( comm )**
• Blocks until all processes in the group of the communicator `comm` call it.
• Almost never required in a parallel program
  ♦ Occasionally useful in measuring performance and load balancing
  ♦ In unusual cases, can increase performance by reducing network contention
  ♦ Does *not* guarantee that processes exit at the same (or even close to the same) time
Collective Data Movement

- One to all
  - Broadcast
  - Scatter (personalized)
- All to one
  - Gather
- All to all
  - Allgather
  - Alltoall (personalized)
- “Personalized” means each process gets different data
Collective Data Movement

Broadcast

Scatter

Gather
Comments on Broadcast

- All collective operations must be called by all processes in the communicator.

- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast.
  - MPI_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive.

- Example of orthogonallity of the MPI design: MPI_Recv need not test for “multisend”
More Collective Data Movement

\[ \begin{array}{cccc}
P0 & A & B & C & D \\
P1 & B & C & D & A \\
P2 & C & D & A & B \\
P3 & D & A & B & C \\
\end{array} \]

Allgater

\[ \begin{array}{cccc}
P0 & A & B & C & D \\
P1 & A & B & C & D \\
P2 & A & B & C & D \\
P3 & A & B & C & D \\
\end{array} \]

\[ \begin{array}{cccc}
P0 & A0 & A1 & A2 & A3 \\
P1 & B0 & B1 & B2 & B3 \\
P2 & C0 & C1 & C2 & C3 \\
P3 & D0 & D1 & D2 & D3 \\
\end{array} \]

Alltoall

\[ \begin{array}{cccc}
P0 & A0 & B0 & C0 & D0 \\
P1 & A1 & B1 & C1 & D1 \\
P2 & A2 & B2 & C2 & D2 \\
P3 & A3 & B3 & C3 & D3 \\
\end{array} \]
Notes on Collective Communication

- MPI_Allgather is equivalent to
  - MPI_Gather followed by MPI_Bcast
  - But algorithms for MPI_Allgather can be faster

- MPI_Alltoall performs a “transpose” of the data
  - Also called a personalized exchange
  - Tricky to implement efficiently and in general
    - For example, does not require $O(p)$ communication, especially when only a small amount of data is sent to each process
Special Variants

- The basic routines send the same amount of data from each process
  - E.g., MPI_Scatter(&v,1,MPI_INT,...) sends 1 int to each process
- What if you want to send a different number of items to each process?
  - Use MPI_Scatterv
- The “v” (for vector) routines allow the programmer to specify a different number of elements for each destination (one to all routines) or source (all to one routines).
- Efficient algorithms exist for these cases, though not as fast as the simpler, basic routines
Special Variants (Alltoall)

• In one case (MPI_Alltoallw), there are two “vector” routines, to allow more general specification of MPI datatypes for each source
  ♦ Recall that only the type signature needs to match; this allows different layouts in memory for each data being sent
Collective Computation

• Combines communication with computation
  ♦ Reduce
    • All to one, with an operation to combine
  ♦ Scan, Exscan
    • All prior ranks to all, with combination
  ♦ Reduce_scatter
    • All to all, with combination

• Combination operations either
  ♦ Predefined operations
  ♦ User defined operations
Collective Computation

P0  A
P1  B
P2  C
P3  D

Reduce  A+B+C+D

A+B+C+D

A+B+C

A+B

A

P0  A
P1  B
P2  C
P3  D

Scan  A+B+C+D

A+B+C

A+B

A
Collective Computation

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th></th>
<th></th>
<th></th>
<th>A+B+C+D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
</tbody>
</table>

**Allreduce**

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th></th>
<th></th>
<th></th>
<th>A+B+C+D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
</tbody>
</table>

**Exscan**

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th></th>
<th></th>
<th></th>
<th>A+B+C+D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>A+B+C+D</td>
</tr>
</tbody>
</table>
MPI Collective Routines: Summary

- Many Routines, including: Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Alltoallw, Bcast, Exscan, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv
- All versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- Allreduce, Exscan, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions.
- Most routines accept both intra- and inter-communicators
  - Intercommunicator versions are collective between two groups of processes
MPI Built-in Collective Computation Operations

- `MPI_MAX` - Maximum
- `MPI_MIN` - Minimum
- `MPI_PROD` - Product
- `MPI_SUM` - Sum
- `MPI_LAND` - Logical and
- `MPI_LOR` - Logical or
- `MPI_LXOR` - Logical exclusive or
- `MPI_BAND` - Bitwise and
- `MPI_BOR` - Bitwise or
- `MPI_BXOR` - Bitwise exclusive or
- `MPI_MAXLOC` - Maximum and location
- `MPI_MINLOC` - Minimum and location
How Deterministic are Collective Computations?

• In exact arithmetic, you always get the same results
  ♦ but roundoff error, truncation can happen
• MPI does *not* require that the same input give the same output every time
  ♦ Implementations are **encouraged** but **not required** to provide *exactly* the same output given the same input
  ♦ Round-off error may cause slight differences
• Allreduce **does** guarantee that the *same* value is received by **all** processes for each call
• Why didn’t MPI mandate determinism?
  ♦ Not all applications need it
  ♦ Implementations of collective algorithms can use “deferred synchronization” ideas to provide better performance
Defining your own Collective Operations

- Create your own collective computations with:
  ```c
  MPI_Op_create( user_fcn, commutes, &op );
  MPI_Op_free( &op );
  
  user_fcn( invec, inoutvec, len, datatype );
  ```

- The user function should perform:
  ```c
  inoutvec[i] = invec[i] op inoutvec[i];
  ```
  for i from 0 to len-1.

- The user function can be non-commutative.
Understanding the Definition of User Operations

- The declaration is
  void user_op(void *invec, void *inoutvec, int *len, MPI_Datatype *dtype)

  ♦ Why pointers to len, dtype?
    - An attempt to make the C and Fortran-77 versions compatible (Fortran effectively passes most arguments as pointers)
  ♦ Why a void return?
    - No error cases expected

- Both assumptions turned out to be poor choices

- Why the “commutes” flag?
  ♦ Not all operations are commutative. Can you think of one that is not?
An Example of a Non-Commutative Operation

- Matrix multiplication is not commutative
- Consider using MPI_Scan to compute the product of 3x3 matrices from each process
  - MPI implementation is free to use both associativity and commutativity in the algorithms unless the operation is marked as non commutative
- Try it yourself – write the operation and try it using simple rotation matrices
Define the Groups

• MPI_Comm_split(MPI_Comm oldcomm, int color, int key, MPI_Comm *newcomm)
  ♦ Collective over input communicator
  ♦ Partitions based on “color”
  ♦ Orders rank in new communicator based on key
  ♦ Usually the best routine for creating a new communicator over a proper subset of processes
    • Don’t use MPI_Comm_create
  ♦ Can also be used to reorder ranks
    • Question: How would you do that?
Define the Groups

- MPI_Comm_create_group(
  MPI_Comm oldcomm,
  MPI_Group group, int tag,
  MPI_Comm *newcomm)

  ♦ New in MPI-3
  - Collective only over input group, not oldcomm
  - Requires formation of group using MPI group creation routines
    - MPI_Comm_group to get an initial group
    - MPI_Group_incl, MPI_Group_range_incl, MPI_Group_union, etc.
Collective Communication Semantics

- Collective routines on the same communicator must be called in the same order on all participating processes.
- If multi-threaded processes are used (MPI_THREAD_MULTIPLE), it is the user's responsibility to ensure that the collective routines follow the above rule.
- Message tags are not used.
  - Use different communicators if necessary to separate collective operations on the same process.
NonBlocking Collective Operations

• MPI-3 introduced nonblocking versions of collective operations
  ♦ All return an MPI_Request, use the usual MPI_Wait, MPI_Test, etc. to complete.
  ♦ May be mixed with point-to-point and other MPI_Requests
  ♦ Few implementations are fast or offer much concurrency (as of 2015)
  ♦ Follow same ordering rules as blocking operations

• Even MPI_Ibarrier
  ♦ Useful for distributed termination detection
Neighborhood Collectives

- Collective operation on an MPI communicator with a defined topology
  - For Cartesian (MPI_CART), immediate neighbors in coordinate directions
    • Cooresponds to using MPI_Cart_shift with disp=1 in each coordinate
  - For Graph (MPI_DIST_GRAPH), immediate neighbors (as returned by MPI_Dist_graph_neighbors)

- MPI_Neighbor_alltoall
  - Sends distinct messages to each neighbor
  - Receives distinct messages from each neighbor

- MPI_Ineighbor_alltoall for nonblocking version

- Provides an alternative for halo exchanges