Ateneo de Manila University

Parallel Programming with MPI: Collective Communications

Ateneo High Performance Computing Group
1st Semester
http://www.math.admu.edu.ph/ahpc/
william.s.yu@ieee.org
Collective Communication
Collective Communications in MPI

- Communication is coordinated among a group of processes, as specified by communicator.
- Message tags are not used.
- All collective operations are blocking.
- All processes in the communicator group must call the collective operation.
- Three classes of collective operations:
  - Data movement
  - Collective computation
  - Synchronization
Pre-MPI Message-Passing

- A typical (pre-MPI) global operation might look like:
  \[ \text{broadcast(type, address, length)} \]
- As with point-to-point, this specification is a good match to hardware and easy to understand
- But also too inflexible
MPI Basic Collective Operations

- Two simple collective operations:
  
  MPI_BCAST(start, count, datatype, root, comm)

  MPI_REDUCE(start, result, count, datatype, operation, root, comm)

- The routine MPI_BCAST sends data from one process to all others.
- The routine MPI_REDUCE combines data from all processes returning the result to a single process.
MPI_BCAST

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INOUT</td>
<td>buffer</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>
MPI_BCAST Binding

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

void MPI::Comm::Bcast(void* buffer, int count,
                       const MPI::Datatype& datatype,
                       int root) const = 0

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT,
          COMM, IERROR)

$type$ BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
**MPI_REDUCE**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendbuf</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
</tr>
<tr>
<td>IN</td>
<td>op</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
</tr>
</tbody>
</table>
Binding for MPI_REDUCE

int MPI_Reduce(void* sendbuf, void* recvbuf,
              int count, MPI_Datatype datatype,
              MPI_Op op, int root, MPI_Comm comm)

void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count, const
                       MPI::Datatype& datatype,
                       const MPI::Op& op,
                       int root) const = 0

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
            ROOT, COMM, IERROR)

<typename> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
MPI Basic Collective Operations

- Broadcast and reduce are very important mathematically

- Many scientific programs can be written with just
  
  MPI_INIT
  MPI_COMM_SIZE
  MPI_COMM_RANK
  MPI_SEND
  MPI_RECV
  MPI_BCAST
  MPI_REDUCE
  MPI_FINALIZE

- Some won’t even need send and receive
Available Collective Patterns

- Schematic representation of collective data movement in MPI
**MPI Collective Routines**

- Many routines:

  - `MPI_ALLGATHER`
  - `MPI_ALLGATHERV`
  - `MPI_ALLREDUCE`
  - `MPI_ALLTOALL`
  - `MPI_ALLTOALLV`
  - `MPI_BCAST`
  - `MPI_GATHER`
  - `MPI_GATHERV`
  - `MPI_REDUCE`
  - `MPI_REDUCE_SCATTER`
  - `MPI_REDUCE_SCATTERV`
  - `MPI_SCAN`
  - `MPI_SCATTER`
  - `MPI_SCATTERV`

- All versions deliver results to all participating processes.

- V versions allow the chunks to have different sizes.

- `MPI_ALLREDUCE`, `MPI_REDUCE`, `MPI_REDUCE_SCATTER`, and `MPI_SCAN` take both built-in and user-defined combination functions.
# Built-in Collective Computation Operations

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Operation</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_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or (xor)</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise xor</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum value and location</td>
</tr>
</tbody>
</table>
Defining Your Own Collective Operations

MPI_OP_CREATE(user_function, commute, op)
MPI_OP_FREE(op)

user_function(invec, inoutvec, len, datatype)

The user function should perform:

inoutvec[i] = invec[i] op inoutvec[i];

for i from 0 to len-1.

user_function can be non-commutative (e.g., matrix multiply).
MPI_SCATTER

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

IN  sendbuf      address of send buffer
IN  sendcount    number of elements sent to each process
IN  sendtype     data type of send buffer elements
OUT recvbuf      address of receive buffer
IN  recvcount    number of elements in receive buffer
IN  recvtype     data type of receive buffer elements
IN  root         rank of sending process
IN  comm         communicator
MPI_SCATTER Binding

int MPI_Scatter(void* sendbuf, int sendcount, 
                MPI_Datatype sendtype, void* recvbuf, 
                int recvcount, MPI_Datatype recvtype, 
                int root, MPI_Comm comm)

void MPI::Comm::Scatter(const void* sendbuf, 
                        int sendcount, 
                        const MPI::Datatype& sendtype, 
                        void* recvbuf, int recvcount, 
                        const MPI::Datatype& recvtype, 
                        int root) const = 0
MPI_SCATTER Binding (cont.)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 
   RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, 
ROOT, COMM, IERROR
MPI_GATHER

MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, 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 in send buffer</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 for any single receive</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
<td>data type of recv buffer elements</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
<td>rank of receiving process</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
MPI_GATHER Binding

int MPI_Gather(void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)

void MPI::Comm::Gather(const void* sendbuf,
                        int sendcount,
                        const MPI::Datatype& sendtype,
                        void* recvbuf, int recvcount,
                        const MPI::Datatype& recvtype,
                        int root) const = 0
MPI_GATHER Binding (cont.)

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 
RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

ctype> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, 
ROOT, COMM, IERROR
Synchronization

MPI_BARRIER(comm)

Function blocks until all processes in “comm” call it

```c
int MPI_BARRIER(MPI_Comm comm)
void Intracomm::Barrier() const

MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```
# Simple C Example

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

main(int argc, char **argv)
{
    int rank, size, myn, i, N;
    double *vector, *myvec, sum, mysum, total;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* In the root process read the vector length, initialize
    the vector and determine the sub-vector sizes */
    if (rank == 0) {
        printf("Enter the vector length : ");
        scanf("%d", &N);
        vector = (double *)malloc(sizeof(double) * N);
        for (i = 0, sum = 0; i < N; i++)
            vector[i] = 1.0;
        myn = N / size;
    }

    /* Broadcast the local vector size */
    MPI_Bcast(&myn, 1, MPI_INT, 0, MPI_COMM_WORLD);
    /* allocate the local vectors in each process */
    myvec = (double *)malloc(sizeof(double)*myn);
    /* Scatter the vector to all the processes */
```
MPI_Scatter(vector, myn, MPI_DOUBLE, myvec, myn, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* Find the sum of all the elements of the local vector */
for (i = 0, mysum = 0; i < myn; i++)
  mysum += myvec[i];

/* Find the global sum of the vectors */
MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

/* Multiply the local part of the vector by the global sum */
for (i = 0; i < myn; i++)
  myvec[i] *= total;

/* Gather the local vector in the root process */
MPI_Gather(myvec, myn, MPI_DOUBLE, vector, myn, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if (rank == 0)
  for (i = 0; i < N; i++)
    printf("[%d] %f\n", rank, vector[i]);

MPI_Finalize();
return 0;