MPI 5

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- Learn about basic collective communication.
MPI Collective Communications

- **Point-to-Point (P2P) Communication** sends data from one point to another, one task sends while another receives.
  - **Blocking** (MPI_Send() and MPI_MPI_Recv()) send routing will only return after it is safe to modify the buffer.
  - **Non-blocking** (MPI_Isend() and MPI_Irecv()) send/receive routines return immediately.
  - Improper use of blocking receive/send will result in **deadlock**, where two processors can’t progress because each of them is waiting on the other.

- **Collective communication** is defined as communication between more than two (usually many) processors. A few forms:
  - One-to-many
  - Many-to-one
  - Many-to-many
Collective communication routines must involve all processes within the scope of a communicator.

- All processes are by default, members in the communicator MPI_COMM_WORLD.
- Additional communicators can be defined by the programmer.

Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.

It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.
Types of Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.

- **Data Movement** - broadcast, scatter/gather, all to all.

- **Collective computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.
Barrier for Synchronization

- MPI has a special function that is dedicated to synchronizing processes:
  
  ```c
  MPI_Barrier( MPI_COMM comm )
  ```

  - comm: the group of processes

- Blocks until all processes in the group call it.

- The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function.
Barrier for Synchronization

- Process zero first calls MPI_Barrier at the first time snapshot (T 1).
- While process zero is hung up at the barrier, process one and three eventually make it (T 2).
- When process two finally makes it to the barrier (T 3), all of the processes then begin execution again (T 4).
- MPI_Barrier can be useful for many things. One of the primary uses of MPI_Barrier is to synchronize a program so that portions of the parallel code can be timed accurately.
Barrier for Synchronization

- All collective operations in MPI are blocking, which means that it is safe to use all buffers passed to them after they return.
  - In particular, this means that all data was received when one of these functions returns. (However, it does not imply that all data was sent!) So MPI_Barrier is not necessary (or very helpful) before/after collective operations, if all buffers are valid already.

- Please also note, that MPI_Barrier does not magically wait for non-blocking calls.
  - If you use a non-blocking send/recv and both processes wait at an MPI_Barrier after the send/recv pair, it is not guaranteed that the processes sent/received all data after the MPI_Barrier. Use MPI_Wait (and friends) instead.
Data Movement

Before

After

ROOT

B

B

B

B

B

MPI_BCAST

ROOT

ABCDE

A

B

C

D

E

MPI_SCATTER

ROOT

A

B

C

D

E

MPI_GATHER

ROOT

A

B

C

D

E

MPI_ALLGATHER

ROOT

ABCDEF

AFKPU

BGLQV

CHMRW

DINSX

EJOTY

MPI_ALL_TO_ALL

0

1

2

3

4

RANK

0

1

2

3

4
A **broadcast** is one of the standard collective communication techniques.

During a broadcast, one process sends the same data to all processes in a communicator.

One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.
Broadcast: MPI_Bcast

- **C Syntax:** MPI_Bcast( data, count, datatype, root, comm)
- **C++ Syntax:** MPI::Comm.Bcast( data, count, datatype, root)
- No tag!!
- A broadcast has a specified root process and every process receives one copy of the message from the root.
- All processes must specify the same root (and communicator).
- The root argument is the rank of the root process.
- The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.
Lab: MPI_Bcast()

- Broadcasting with MPI_Bcast
- Implement
  - data variable is assigned to have value “100” at rank 0.
  - Broadcast the data into all processes using collective communication (MPI_Bcast).
  - print value at each process before and after the communication to verify it.
Lab: MPI_Bcast()

- Broadcasting with MPI_Bcast

```cpp
#include <iostream>
#include <cstdlib> // has exit(), etc.
#include <ctime>
#include "mpi.h" // MPI header file

using namespace std;

//****************************************************************************80
int main (int argc, char **argv)
//****************************************************************************80
{
    int data;
    MPI::Status status;
    int tag = 0;

    MPI::Init (argc, argv); // Initialize MPI.
    int num_proc = MPI::COMM_WORLD.Get_size(); // get number of processes
    int proc_id = MPI::COMM_WORLD.Get_rank(); // get this process's number (ranges from 0 to nprocs - 1)

    if (proc_id == 0) { // root (proc_id = 0)
        data = 100;
    }

    cout << "Before Bcast, data = " << data << " in proc_id = " << proc_id << endl;
    MPI::COMM_WORLD.Bcast(&data, 1, MPI_INT, 0); // MPI_Bcast
    cout << "After Bcast, data = " << data << " in proc_id = " << proc_id << endl;

    MPI::Finalize(); // clean up for MPI
    return 0;
}
```
Collective vs. P2P?

- Broadcasting with MPI_Send and MPI_Recv

```cpp
if (rank_id == 0) {
    data = 100;
    for (int i = 1; i < num_proc; i++) {  // send our data to everyone
        MPI::COMM_WORLD.Send(&data, count, datatype, i, tag);
    }
} else {
    // If we are a receiver process, receive the data from the root
    MPI::COMM_WORLD.Recv(&data, count, datatype, 0, tag, status);
}
```

- Broadcasting with MPI_Bcast

```cpp
if (rank_id == 0) {
    data = 100;
}
MPI::COMM_WORLD.Bcast(&data, count, datatype, 0);
```

- Time comparison?
Collective vs. P2P?

- Time comparison?

<table>
<thead>
<tr>
<th>Processors</th>
<th>P2P broadcast</th>
<th>MPI_Bcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0344</td>
<td>0.0344</td>
</tr>
<tr>
<td>4</td>
<td>0.1025</td>
<td>0.0817</td>
</tr>
<tr>
<td>8</td>
<td>0.2385</td>
<td>0.1084</td>
</tr>
<tr>
<td>16</td>
<td>0.5109</td>
<td>0.1296</td>
</tr>
</tbody>
</table>
Data Movement

Before

After

MPI_BCAST

MPI_SCATTER

MPI_GATHER

MPI_ALLGATHER

MPI_ALL_TO_ALL

0 1 2 3 4 RANK 0 1 2 3 4
Scatter: MPI_Scatter

MPI_Scatter

- **Processors**
  - p0
  - p1
  - p2
  - p3

- **Memory**
  - A
  - B
  - C
  - D

- **Processors**
  - p0
  - p1
  - p2
  - p3

- **Memory**
  - A
  - B
  - C
  - D
Scatter: MPI_Scatter

- **MPI_Scatter** involves a designated root process sending data to all processes in a communicator.

- **MPI_Scatter** sends *chunks of an array* to different processes.

- Check out the illustration below for further clarification.

- The main difference from **MPI_Bcast** is that the send and receive details are in general different and so must both be specified in the argument lists.
Scatter: MPI_Scatter

- C Syntax: MPI_Scatter(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root, comm)
- C++ Syntax: MPI::Comm.Scatter(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root)
- No tag!!
- Note that the sendcount (at the root) is the number of elements to send to each process, not to send in total. Therefore if sendtype = recvtype, sendcount = recvcount.
- The sendbuf, sendcount, sendtype arguments are significant only at the root. The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.
Lab: MPI_Scatter()

- Scattering with MPI_Scatter()
  - `int data_array[8] = {};`
  - `int recv_buffer[8] = {};`
  - `data_array` is assigned to 100,...107 at `rank_id = 0`
  - scattering two elements to other processes that will be saved to `recv_buffer`. 
Lab: MPI_Scatter()

● Scattering with MPI_Scatter

```c
#include <mpi.h>

int main (int argc, char **argv)
{
    int data_array[8] = {};
    int recv_buffer[8] = {};
    MPI::Status status;

    MPI::Init (argc, argv); // Initialize MPI.
    int num_proc = MPI::COMM_WORLD.Get_size(); // get number of processes
    int proc_id = MPI::COMM_WORLD.Get_rank(); // get this process's number (ranges from 0 to nprocs - 1)

    if (proc_id == 0) { // root (proc_id = 0)
        for (int i=0; i<8; i++) {
            data_array[i] = 100+i;
        }
    }

    int send_count = 2;
    int recv_count = 2;

    cout << "Before Scatter, data_array = "; . . .

    // MPI_Scatter
    MPI::COMM_WORLD.Scatter(&data_array, send_count, MPI::INT, &recv_buffer, recv_count, MPI::INT, 0);

    cout << "After Scatter, data_array = "; . . .

    MPI::Finalize(); // clean up for MPI
    return 0;
}
```

Data Movement

**Before**

- **ROOT**: 
  - B

**After**

- **MPI_BCAST**: 
  - B
  - B
  - B
  - B
  - B

- **ROOT**: 
  - ABCDE

**MPI_SCATTER**: 

- A
- B
- C
- D
- E

- **ROOT**: 
  - A
  - B
  - C
  - D
  - E

**MPI_GATHER**: 

- A
- B
- C
- D
- E

**MPI_ALLGATHER**: 

- A
- B
- C
- D
- E

**MPI_ALL_TO_ALL**: 

- ABCDE
- FGHIJ
- KLMNO
- PQRST
- UVWXYZ

- **RANK**
  - 0
  - 1
  - 2
  - 3
  - 4
Gather: MPI_Gather

MPI_Gather
Gather: MPI_Gather

- **MPI_Gather** is the inverse of MPI_Scatter.

- **MPI_Gather** takes elements from many processes and gathers them to one single process.

- This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.
Gather: MPI_Gather

- **C Syntax:** MPI_Gather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root, comm )

- **C++ Syntax:** MPI::Comm.Gather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root )

- No tag!!

- Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers them to the root process.

- The elements are ordered by the rank of the process from which they were received.
Lab: MPI_Gather()

- Gathering with MPI_Gather()
  - `int data_array[8] = {};
  - `int recv_buffer[8] = {};
  - `data_array` is assigned to 100, 101 at `rank_id = 0`, 102, 103 at `rank_id = 1`, ..., 106, 107 at `rank_id = 3`.
  - Gathering all elements to root (`rank_id = 0`) and compute average of them.
Lab: MPI_Gather()

Gathering with MPI_Gather

```cpp
//**********************************************************************************
int main (int argc, char **argv)
//**********************************************************************************
{
    int data_array[8] = {};  
    int recv_buffer[8] = {};  
    MPI::Status status;

    MPI::Init (argc, argv);   // Initialize MPI.
    int num_proc = MPI::COMM_WORLD.Get_size();  // get number of processes
    int proc_id = MPI::COMM_WORLD.Get_rank();    // get this process's number (ranges from 0 to nprocs - 1)

    // assign data_array
    for (int i=0; i<2; i++) {
        data_array[i] = 100+i+proc_id*2;
    }

    int send_count = 2;
    int recv_count = 2;

    cout << "Before Gather, data_array = "; . . .
    MPI::COMM_WORLD.Gather(&data_array, send_count, MPI::INT, &recv_buffer, recv_count, MPI::INT, 0);
    cout << "After Gather, data_array = "; . . .

    // get average of recv_buffer at root and print it
    if (proc_id == 0) { ... }

    MPI::Finalize();       // clean up for MPI
    return 0;
}
```
Data Movement

Before

ROOT

B

MPI_BCAST

After

B

B

B

B

B

ROOT

ABCDE

MPI_SCATTER

A

BCDE

C

D

E

ROOT

A

B

C

D

E

MPI_GATHER

A

BCDE

C

D

E

MPI_ALLGATHER

A

BCDE

BCDE

BCDE

BCDE

MPI_ALL_TO_ALL

AFKPU

BGLQV

CHMRW

DINSX

EJOTY

0 1 2 3 4 RANK 0 1 2 3 4
Gather: MPI_Allgather

- So far, we have covered two MPI routines that perform *many-to-one* or *one-to-many* communication patterns, which simply means that many processes send/receive to one process.

- Oftentimes it is useful to be able to send many elements to many processes (i.e. a *many-to-many* communication pattern).

- MPI_Allgather has this characteristic.

- Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes.
Gather: MPI_Allgather

- C Syntax: MPI_Allgather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, comm)
- C++ Syntax: MPI::Comm.Allgather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype)
- Just like MPI_Gather, the elements from each process are gathered in order of their rank, except this time the elements are gathered to all processes.
- No root, No tag!!
- In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast.
Gathering with MPI_Allgather()

- `int data_array[8] = {};`
- `int recv_buffer[8] = {};`
- `data_array` is assigned to 100, 101 at `rank_id = 0`, 102, 103 at `rank_id = 1`, ..., 106, 107 at `rank_id = 3`.
- Gathering all elements to all processes and compute average of them at `rank_id = 1`.
Lab: MPI_Allgather()

Gathering with MPI_Allgather

```c
int main (int argc, char **argv) {
    int data_array[8] = {};  
    int recv_buffer[8] = {};  
    MPI::Status status;  
    MPI::Init (argc, argv);     // Initialize MPI.  
    int num_proc = MPI::COMM_WORLD.Get_size(); // get number of processes  
    int proc_id = MPI::COMM_WORLD.Get_rank();    // get this process's number (ranges from 0 to nprocs - 1)  
    // assign data_array  
    for (int i=0; i<2; i++) {
        data_array[i] = 100+i+proc_id*2;
    }  
    int send_count = 2;
    int recv_count = 2;
    cout << "Before Gather, data_array = "; 
    MPI::COMM_WORLD.Allgather(&data_array, send_count, MPI::INT, &recv_buffer, recv_count, MPI::INT);
    cout << "After Gather, data_array = "; 
    // get average of recv_buffer at root and print it  
    if (proc_id == 0) { ...  
    }
    MPI::Finalize();       // clean up for MPI  
    return 0;
}
```
Data Movement

Before

ROOT

B

After

MPI_BCAST

B

B

B

B

B

ROOT

ABCD

E

MPI_SCATTER

A

B

C

D

E

ROOT

ABC

D

E

MPI_GATHER

A

B

C

D

E

MPI_ALLGATHER

A

B

C

D

E

MPI_ALL_TO_ALL

AFKPU

BGLQV

CHMRW

DINSX

EJOTY

0 1 2 3 4

RANK 0 1 2 3 4
MPI_Alltoall

- Data movement operation.
- 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_Alltoall

- C Syntax: MPI_Alltoall(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, comm )
- C++ Syntax: MPI::Comm.Alltoall(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype)
- No root, No tag!!