MPI 6

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 collective communication in detail.
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.

![Diagram of collective operations](image-url)
Data Movement

Before

ROOT

B

MPI_BCAST

B

B

B

B

B

After

ROOT

ABCDE

MPI_SCATTER

A

ABCDE

C

D

E

ROOT

A

B

C

D

E

MPI_GATHER

A

ABCDE

C

D

E

ROOT

A

B

C

D

E

MPI_ALLGATHER

A

ABCDE

B

ABCDE

C

ABCDE

D

ABCDE

E

MPI_ALL_TO_ALL

ABCDE

FGHIJ

KLMNO

PQRST

UVWXY

0

1

2

3

4

RANK

0

1

2

3

4

CSCI 4850/5850 HPC
MPI_Reduce

- Collective computation operation.
- Applies a reduction operation on all tasks in the group and places the result in one task.
MPI_Reduce

The diagram illustrates the process of MPI_Reduce, where multiple processors (p0, p1, p2, p3) participate in reducing data. The reduction operation is characterized by the summation symbol (∑) indicating the aggregation of data from different processors. The memory segment on the right shows the aggregated result "X" after the reduction operation.
**MPI_Reduce**

- C Syntax: `MPI_Reduce( sendbuffer, recvbuffer, count, datatype, operation, root, comm )`
- C++ Syntax: `MPI::Reduce ( sendbuffer, recvbuffer, count, datatype, operation, root)`
- No tag!!
- All processes must specify the same root (and communicator).
- The `sendbuffer` parameter is an array of elements of type `datatype` that each process wants to reduce.
- The `recvbuffer` is only relevant on the process with a rank of `root`. 
MPI Reduction Operations

- **MPI_MAX** - Returns the maximum element.
- **MPI_MIN** - Returns the minimum element.
- **MPI_SUM** - Sums the elements.
- **MPI_PROD** - Multiplies all elements.
- **MPI_LAND** - Performs a logical \textit{and} across the elements.
- **MPI_LOR** - Performs a logical \textit{or} across the elements.
- **MPI_BAND** - Performs a bitwise \textit{and} across the bits of the elements.
- **MPI_BOR** - Performs a bitwise \textit{or} across the bits of the elements.
- **MPI_MAXLOC** - Returns the maximum value and the rank of the process that owns it.
- **MPI_MINLOC** - Returns the minimum value and the rank of the process that owns it.
• It is also useful to see what happens when processes contain multiple elements.

• The illustration below shows reduction of multiple numbers per process.

\[\text{MPI}_\text{Reduce}\]

\begin{verbatim}
  0 5 1 1 2 3 2 7 8 3 4 2
  0 18 14
\end{verbatim}

• The processes from the above illustration each have two elements.

• The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the \(i\)th element from each array are summed into the \(i\)th element in result array of process 0.
# include <iostream>
# include <cstdlib>     // has exit(), etc.
# include <ctime>
# include "mpi.h"       // MPI header file

using namespace std;

int main( int argc, char *argv[]) 
{
    int N = 1000;
    int tmp_sum = 0;
    int partial_sum = 0;
    int total_sum = 0;
    int id, p, startval, endval;

    // Initialize MPI
    MPI::Init(argc, argv);

    // Determine this process's rank
    id = MPI::COMM_WORLD.Get_rank();

    // Determine the number of available processes
    p = MPI::COMM_WORLD.Get_size();

    // Calculate partial sum (from id=0 ~ p-1). Master should work, too!
    startval = N*id/p + 1;
    endval =  N*(id+1)/p;
    for(int i=startval; i<=endval; i++) //Need to modify based on number of processors
    {
        partial_sum += i;
    }
}
// workers (id != 0) should send their partial sum to master
if(id != 0)
{
    MPI::COMM_WORLD.Send(&partial_sum, 1, MPI::INT, 0, 0);
}
else // master should get partial sum from all workers
{
    for (int j = 1; j < p; j++)     // should start j=1 to avoid self-receive from ID=0
    {
        MPI::COMM_WORLD.Recv(&tmp_sum, 1, MPI::INT, j, 0);
        partial_sum += tmp_sum; // master should add each partial sum from workers
    }
}
if(id == 0)
{
    total_sum = partial_sum;       // master should have its own partial sum
    cout << "The sum from 1 to " << N << " is: " << total_sum << endl;
}

// Terminate MPI
MPI::Finalize();

return EXIT_SUCCESS;
Lab: Sum of the first N Integers using MPI_Reduce

```cpp
MPI::COMM_WORLD.Reduce(&partial_sum, &tmp_sum, 1, MPI::INT, MPI::SUM, 0);
cout << "The partial_sum = " << partial_sum << " for proc_id = " << id << endl;
cout << "The tmp_sum = " << tmp_sum << " for proc_id = " << id << endl;

if(id == 0)
{
    total_sum = tmp_sum;   // master should have it's own partial sum
    cout << "The sum from 1 to " << N << " is: " << total_sum << endl;
}

//Terminate MPI
MPI::Finalize();

return EXIT_SUCCESS;
```
MPI_Allreduce

- Many parallel applications will require accessing the reduced results across all processes rather than the root process.
- In a similar complementary style of MPI_Allgather to MPI_Gather, MPI_Allreduce will reduce the values and distribute the results to all processes.
MPI_Allreduce

- C Syntax: MPI_Allreduce(sendbuffer, recvbuffer, count, datatype, operation, comm)
- C++ Syntax: MPI::COMM.Allreduce(sendbuffer, recvbuffer, count, datatype, operation)
- No tag, no root!!
- MPI_Allreduce is identical to MPI_Reduce with the exception that it does not need a root process id (since the results are distributed to all processes).
Lab: Sum of the first N Integers using MPI_Allreduce

```cpp
MPI::COMM_WORLD.Allreduce(&partial_sum, &tmp_sum, 1, MPI::INT, MPI::SUM);
cout << "The partial_sum = " << partial_sum << " for proc_id = " << id << endl;
cout << "The tmp_sum = " << tmp_sum << " for proc_id = " << id << endl;

if(id == 1)
{
    total_sum = tmp_sum;  // master should have it's own partial sum
    cout << "The sum from 1 to " << N << " is: " << total_sum << endl;
}

//Terminate MPI
MPI::Finalize();

return EXIT_SUCCESS;
```
Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”

- For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
Collective vs. Point-to-Point Communications

- **Point-to-point** communications are matched on the basis of tags and communicators.
- **Collective** communications **don’t use tags**.
- They’re matched solely on the basis of the communicator and the order in which they’re called.
Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$x$ is a vector with $n$ components

$y = Ax$ is a vector with $m$ components

$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1}$

$i$-th component of $y$

Dot product of the $i$th row of $A$ with $x$. 
Matrix-vector multiplication

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{array}
\]

\[
\begin{array}{c}
  x_0 \\
  x_1 \\
  \vdots \\
  x_{n-1}
\end{array}
\]

\[
\begin{array}{c}
  y_0 \\
  y_1 \\
  \vdots \\
  y_{m-1}
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
C style arrays

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
\end{pmatrix}
\]

stored as

0 1 2 3 4 5 6 7 8 9 10 11
Serial matrix-vector multiplication

```c
void Mat_vect_mult(
    double A[] /* in */,
    double x[] /* in */,
    double y[] /* out */,
    int m /* in */,
    int n /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vect_mult */
```
Example 1: Matrix-vector Multiplication (Columnwise)

Schematic of parallel decomposition for vector-matrix multiplication, $A = B \times C$. The vector $A$ is depicted in yellow. The matrix $B$ and vector $C$ are depicted in multiple colors representing the portions, columns, and elements assigned to each processor, respectively.
Example 1: Matrix-vector Multiplication (Columnwise)

\[
A = B \cdot C
\]

\[
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\end{bmatrix} = \begin{bmatrix}
b_{0,0}c_0 + b_{0,1}c_1 + b_{0,2}c_2 + b_{0,3}c_3 \\
b_{1,0}c_0 + b_{1,1}c_1 + b_{1,2}c_2 + b_{1,3}c_3 \\
b_{2,0}c_0 + b_{2,1}c_1 + b_{2,2}c_2 + b_{2,3}c_3 \\
b_{3,0}c_0 + b_{3,1}c_1 + b_{3,2}c_2 + b_{3,3}c_3 \\
\end{bmatrix}
\]

Reduction (SUM)
HW5: Matrix-vector Multiplication

- \( B_{0,0} \sim B_{3,3} = 1 \sim 16, \ C_{0} \sim C_{3} = 1 \) are loaded in root process.
- The columns of matrix \( B \) and elements of column vector \( C \) must be distributed to the various processes using MPI commands called \textit{scatter} operations (for this lab, just assume 4 processes).
- Each processor now has a column of \( B \), called \( B_{\text{part}} \), and an element of \( C \), called \( C_{\text{part}} \). Each process can now perform an independent vector-scalar multiplication.
- Once this has been accomplished, every process will have a part of the final column vector \( A \), called \( A_{\text{part}} \).
- The column vectors on each process can be added together with an MPI reduction command that computes the final sum on the root process.