

MPI Collective communication

CPS343

Parallel and High Performance Computing

Spring 2016

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - Gathering data
 - Other collective communication routines
- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - Gathering data
 - Other collective communication routines
- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

List of MPI collective communication routines

MPI provides the following routines for *collective communication*:

<code>MPI_Bcast()</code>	– Broadcast (one to all)
<code>MPI_Reduce()</code>	– Reduction (all to one)
<code>MPI_Allreduce()</code>	– Reduction (all to all)
<code>MPI_Scatter()</code>	– Distribute data (one to all)
<code>MPI_Gather()</code>	– Collect data (all to one)
<code>MPI_Alltoall()</code>	– Distribute data (all to all)
<code>MPI_Allgather()</code>	– Collect data (all to all)

We've already been introduced to the first three. Today we focus on the last four and on using all of these in programs.

1 MPI Collective communication functions

- List of collective communication functions
- **Scattering data**
- Gathering data
- Other collective communication routines

2 Example programs

- Scatter/Gather example
- All-to-all example
- Vector scatter example

Scattering data

A common parallel programming model involves

- ① read in or generate data on single (root) process
- ② distribute data to worker processes
- ③ perform work in parallel
- ④ collect data back on root process
- ⑤ output or save data to file

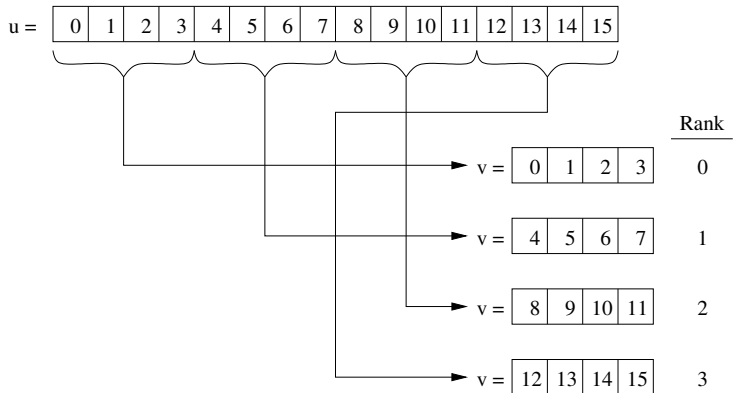
We've already seen how `MPI_Bcast()` can be used to send identical data to all processes. The model here, however, assumes that different data is sent to each process.

The `MPI_Scatter()` function does exactly this. Typically data is in an array on the root process and we want to send a different portion of the array to each worker process (often including the root).

MPI_Scatter() example

Suppose there are four processes including the root (process 0). A 16 element array u on the root should be distributed among the processes and stored locally in v . Every process should include the following line:

```
MPI_Scatter(u, 4, MPI_INT, v, 4, MPI_INT, 0, MPI_COMM_WORLD);
```



MPI_Scatter()

The calling sequence for `MPI_Scatter()` is

```
int MPI_Scatter(  
    void *sendbuf,           // pointer to send buffer  
    int sendcount,          // items to send per process  
    MPI_Datatype sendtype,   // type of send buffer data  
    void *recvbuf,          // pointer to receive buffer  
    int recvcount,          // number of items to receive  
    MPI_Datatype recvtype,   // type of receive buffer data  
    int root,               // rank of sending process  
    MPI_Comm comm)          // MPI communicator to use
```

- The contents of `sendbuf` on process with rank `root` are distributed in groups of `sendcount` elements of type `sendtype` to all processes.
- `sendbuf` should contain at least `sendcount × number_of_processes` data items of type `sendtype` where `number_of_processes` is the number of processes returned by `MPI_Comm_size()`.
- Each process, including the root, receives `recvcount` elements of type `recvtype` into `recvbuf`.

Note:

- All arguments are significant on root process
- All but first three arguments are significant on all other processes, but first three arguments must be supplied and so must be declared. In particular `sendbuf` must be declared but may be a `NULL` pointer.
- Usually `sendtype` and `recvtype` are the same and `sendcount` and `recvcount` are the same, but this is not required; type conversion is possible.
- Like `MPI_Bcast()`, all processes that belong to the specified communicator must participate in the scatter operation.

Another data distribution function

The `MPI_Scatter()` function is a “one-to-all” operation; one process distributes data to all the processes, including itself.

In some cases data from all processes must be redistributed as if each process called `MPI_Scatter()`. The MPI function `MPI_Alltoall()` can be used to do this.

MPI_Alltoall() example

Suppose there are four processes including the root, each with arrays as shown below on the left. After the all-to-all operation

```
MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);
```

the data will be distributed as shown below on the right:

array u	Rank	array v																
<table><tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr></table>	10	11	12	13	14	15	16	17	0	<table><tr><td>10</td><td>11</td><td>20</td><td>21</td><td>30</td><td>31</td><td>40</td><td>41</td></tr></table>	10	11	20	21	30	31	40	41
10	11	12	13	14	15	16	17											
10	11	20	21	30	31	40	41											
<table><tr><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td></tr></table>	20	21	22	23	24	25	26	27	1	<table><tr><td>12</td><td>13</td><td>22</td><td>23</td><td>32</td><td>33</td><td>42</td><td>43</td></tr></table>	12	13	22	23	32	33	42	43
20	21	22	23	24	25	26	27											
12	13	22	23	32	33	42	43											
<table><tr><td>30</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td></tr></table>	30	31	32	33	34	35	36	37	2	<table><tr><td>14</td><td>15</td><td>24</td><td>25</td><td>34</td><td>35</td><td>44</td><td>45</td></tr></table>	14	15	24	25	34	35	44	45
30	31	32	33	34	35	36	37											
14	15	24	25	34	35	44	45											
<table><tr><td>40</td><td>41</td><td>42</td><td>43</td><td>44</td><td>45</td><td>46</td><td>47</td></tr></table>	40	41	42	43	44	45	46	47	3	<table><tr><td>16</td><td>17</td><td>26</td><td>27</td><td>36</td><td>37</td><td>46</td><td>47</td></tr></table>	16	17	26	27	36	37	46	47
40	41	42	43	44	45	46	47											
16	17	26	27	36	37	46	47											

MPI_Alltoall()

The calling sequence for `MPI_Alltoall()` is

```
int MPI_Scatter(  
    void *sendbuf,           // pointer to send buffer  
    int sendcount,          // items to send per process  
    MPI_Datatype sendtype,  // type of send buffer data  
    void *recvbuf,          // pointer to receive buffer  
    int recvcount,          // number of items to receive  
    MPI_Datatype recvtype,  // type of receive buffer data  
    MPI_Comm comm)         // MPI communicator to use
```

- The contents of `sendbuf` on each process are distributed in groups of `sendcount` elements of type `sendtype` to all processes.
- `sendbuf` should contain at least `sendcount × number_of_processes` data items of type `sendtype`.
- Each process receives `recvcount` elements of type `recvtype` into `recvbuf`.
- All arguments are significant on all processes.

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - **Gathering data**
 - Other collective communication routines
- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

MPI_Gather()

MPI provides the `MPI_Gather()` function to collect distinct data elements from multiple processes and combine them in a single buffer on one process.

The gather operation is the inverse of the scatter operation and the calling sequence is similar.

```
int MPI_Gather(  
    void *sendbuf,           // pointer to send buffer  
    int sendcount,          // number of items to send  
    MPI_Datatype sendtype,  // type of send buffer data  
    void *recvbuf,          // pointer to receive buffer  
    int recvcnt,            // items to receive per process  
    MPI_Datatype recvtype,  // type of receive buffer data  
    int root,               // rank of receiving process  
    MPI_Comm comm)         // MPI communicator to use
```

Note:

- The contents from each process' `sendbuf` are sent to the root process and placed in consecutive groups in rank order in the root process' `recvbuf`.
- `recvbuf` must be declared on all processes, but may be `NULL` on non-root processes. The root process must have enough space to hold at least `recvcount × number_of_processes` elements of type `recvtype`.
- As in the case of `MPI_Scatter()`, all processes associated with the communicator must participate in the gather operation.
- All arguments are significant on the root process. All but `recvbuf`, `recvcount`, and `recvtype` are significant on all other processes.

MPI_Gather() example

Assume the variable `rank` contains the process rank. What will be stored in array `b[]` on each of four processes if each executes the following code fragment?

```
int b[4] = {0, 0, 0, 0};  
MPI_Gather( &rank, 1, MPI_INT, b, 1, MPI_INT, 3,  
           MPI_COMM_WORLD );
```


MPI_Gather() example

Assume the variable `rank` contains the process rank. What will be stored in array `b[]` on each of four processes if each executes the following code fragment?

```
int b[4] = {0, 0, 0, 0};  
MPI_Gather( &rank, 1, MPI_INT, b, 1, MPI_INT, 3,  
           MPI_COMM_WORLD );
```

Answer:

- rank 0: `b[] = {0, 0, 0, 0}`
- rank 1: `b[] = {0, 0, 0, 0}`
- rank 2: `b[] = {0, 0, 0, 0}`
- rank 3: `b[] = {0, 1, 2, 3}`

MPI_Allgather()

`MPI_Gather()` collects data from all processes on a single process. In some instances each process needs to gather the same data from all processes. To do this, MPI provides `MPI_Allgather()`.

This function works just like `MPI_Gather()` except the `recvbuf` is filled on all processes.

The calling sequence for `MPI_Allgather()` is:

```
int MPI_Allgather(  
    void *sendbuf,           // pointer to send buffer  
    int sendcount,          // number of items to send  
    MPI_Datatype sendtype,   // type of send buffer data  
    void *recvbuf,          // pointer to receive buffer  
    int recvcnt,            // items to receive per process  
    MPI_Datatype recvtype,   // type of receive buffer data  
    MPI_Comm comm)          // MPI communicator to use
```

MPI_Allgather() example

What will be stored in array `b[]` on each of four processes if each executes the following code fragment?

```
int b[4] = {0, 0, 0, 0};  
MPI_Allgather( &rank, 1, MPI_INT, b, 1, MPI_INT, 3,  
              MPI_COMM_WORLD );
```

MPI_Allgather() example

What will be stored in array `b[]` on each of four processes if each executes the following code fragment?

```
int b[4] = {0, 0, 0, 0};  
MPI_Allgather( &rank, 1, MPI_INT, b, 1, MPI_INT, 3,  
               MPI_COMM_WORLD );
```

Answer:

- rank 0: `b[] = {0, 1, 2, 3}`
- rank 1: `b[] = {0, 1, 2, 3}`
- rank 2: `b[] = {0, 1, 2, 3}`
- rank 3: `b[] = {0, 1, 2, 3}`

1 MPI Collective communication functions

- List of collective communication functions
- Scattering data
- Gathering data
- Other collective communication routines

2 Example programs

- Scatter/Gather example
- All-to-all example
- Vector scatter example

Four more collective communication routines

- The four routines we've just considered all communicate the same amount of data to or from processes.
- MPI provides four more routines that work identically to these four except the amount of data transferred can vary from process to process.
- These routines have the same names as their uniform data counterparts but with a "v" (for *vector*) appended:

<code>MPI_Scatterv()</code>	– Distribute data (one to all)
<code>MPI_Gatherv()</code>	– Collect data (all to one)
<code>MPI_Alltoallv()</code>	– Distribute data (all to all)
<code>MPI_Allgatherv()</code>	– Collect data (all to all)

MPI_Scatterv()

As an example of how these functions are used, we examine the calling sequence for `MPI_Scatterv()`

```
int MPI_Scatterv(  
    void *sendbuf,           // pointer to send buffer  
    int *sendcounts,         // array of send counts  
    int *displs,             // array of displacements  
    MPI_Datatype sendtype,   // type of send buffer data  
    void *recvbuf,           // pointer to receive buffer  
    int recvcnt,             // number of items to receive  
    MPI_Datatype recvtype,   // type of receive buffer data  
    int root,                // rank of sending process  
    MPI_Comm comm)           // MPI communicator to use
```

- Here `sendcounts` is an array of counts corresponding to the number of data items to be sent to each process.
- Likewise, `displs` is an array of offsets from the start of `sendbuf` to the start of the data to be sent to each process.

Other collective routines

- MPI provides even more collective routines. Other communication routines include
 - `MPI_Reduce_scatter()`: a reduction followed by a scatter.
 - `MPI_Scan()`: performs a prefix reduction on distributed data.
- Another important collective operation is a *barrier*; a synchronization point for all cooperating processes. The calling sequence is simple:

```
MPI_Barrier(MPI_Comm comm)
```
- Read the MPI documentation for information on these and other functions.

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - Gathering data
 - Other collective communication routines
- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

Scatter/Gather example prolog

```
// In this program a scatter operation distributes the  
// individual elements of an array of integers among the  
// processes. Each process modifies the value it receives  
// and then participates in a gather operation that  
// collects the modified data in the master process where  
// they are once again assembled into the original array.
```

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

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

```
const int MASTER_RANK = 0; // Rank of of the master process
```

```
// Set the number of elements that should be  
// sent to each process. The number of elements  
// in the entire array will be a multiple of  
// this value.
```

```
const int SIZE_PER_PROCESS = 2;
```

Scatter/Gather example main (1)

```
int main( int argc, char* argv[] )
{
    // Initialize the MPI system and determine the
    // number of collaborating processes and the rank
    // of the current process.
    int numProc, rank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    // The master process allocates and initializes
    // the u array of integers. Other processes will
    // need a valid pointer in their scatter/gather
    // calls, but it will be ignored and so can be
    // NULL. Each process needs array v to receive
    // data into.
    int* u = NULL;
    int* v = new int [SIZE_PER_PROCESS];
```

Scatter/Gather example main (2)

```
if ( rank == MASTER_RANK )
{
    // Master process allocates array and fills
    // it with data. The values in the array are
    // 100 * (rank+1) plus the an offset from
    // 0..SIZE_PER_PROCESS.
    u = new int [numProc * SIZE_PER_PROCESS];
    printf( "Master: Scattering =" );
    for ( int i = 0; i < numProc; i++ )
    {
        for ( int j = 0; j < SIZE_PER_PROCESS; j++ )
        {
            int k = i * SIZE_PER_PROCESS + j;
            u[k] = 100 * ( i + 1 ) + j;
            printf( "%5d", u[k] );
        }
    }
    printf( "\n" );
}
```

Scatter/Gather example main (3)

```
// Each process participates in the scatter;  
// the first three parameters ("the source")  
// are used if the process' rank matches the  
// next-to-last parameter. All processes use  
// the next three parameters ("the destination").  
MPI_Scatter( u, SIZE_PER_PROCESS, MPI_INT,  
            v, SIZE_PER_PROCESS, MPI_INT,  
            MASTER_RANK, MPI_COMM_WORLD );  
  
// Each process, including the master, adds a  
// distinguishable value to received data.  
printf( "Process %2d: ", rank );  
for ( int i = 0; i < SIZE_PER_PROCESS; i++ )  
{  
    printf( " (%4d", v[i] );  
    v[i] += 1000 * ( rank + 1 );  
    printf( " -> %4d)", v[i] );  
}  
printf( "\n" );
```

Scatter/Gather example main (4)

```
// Each process participates in the gather. Source
// parameters are used by each process but only the
// master process makes use of destination parameters.
MPI_Gather( v, SIZE_PER_PROCESS, MPI_INT,
            u, SIZE_PER_PROCESS, MPI_INT,
            MASTER_RANK, MPI_COMM_WORLD );

if ( rank == MASTER_RANK ) {
    // Master process displays assembled data
    printf( "Master: Received   =" );
    for ( int i = 0; i < numProc; i++ ) {
        for ( int j = 0; j < SIZE_PER_PROCESS; j++ )
        {
            int k = i * SIZE_PER_PROCESS + j;
            printf( "%5d", u[k] );
        }
    }
    printf( "\n" );
}
```

Scatter/Gather example main (5)

```
// clean up  
delete [] u;  
delete [] v;  
MPI_Finalize();  
  
return 0;  
}
```

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - Gathering data
 - Other collective communication routines

- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

Alltoall example prolog

```
// Demonstration of MPI_Alltoall()  
//  
// In this program an all-to-all operation distributes  
// the individual elements of an array from each process  
// to all the processes.  
  
#include <stdio>  
#include <unistd.h>  
#include <mpi.h>  
  
// Set the number of elements that should be sent to  
// each process. The number of elements in the entire  
// array will be a multiple of this value.  
const int SIZE_PER_PROCESS = 2;
```

Alltoall example main (1)

```
int main( int argc, char* argv[] )
{
    // Initialize the MPI system and determine the number
    // of collaborating processes and the rank of the
    // current process.
    int numProc, rank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
}
```

Alltoall example main (2)

```
// Construct array of data for this process.  
// The values in the array have the form RXYX where  
// "R" is rank of this process,  
// "X" is the group number (1..number_of_processes);  
// this is also the destination rank, and  
// "YY" is a counter value that starts at 00 and  
// works up to SIZE_PER_PROCESS-1.  
int* u = new int [numProc * SIZE_PER_PROCESS];  
int* v = new int [numProc * SIZE_PER_PROCESS];  
for ( int i = 0; i < numProc; i++ ) {  
    for ( int j = 0; j < SIZE_PER_PROCESS; j++ ) {  
        int k = i * SIZE_PER_PROCESS + j;  
        u[k] = 1000 * rank + 100 * i + j;  
        v[k] = 0;  
    }  
}
```

Alltoall example main (3)

```
// Display constructed data
dumpData( rank, numProc, SIZE_PER_PROCESS, u, "Before" );

// Each process participates in the all-to-all operation
MPI_Alltoall( u, SIZE_PER_PROCESS, MPI_INT,
              v, SIZE_PER_PROCESS, MPI_INT,
              MPI_COMM_WORLD );

// Display received data
dumpData( rank, numProc, SIZE_PER_PROCESS, v, "After" );

// Clean up
delete [] u;
delete [] v;
MPI_Finalize();

return 0;
}
```

Alltoall example dumpData (1)

```
void dumpData( int rank, int numProc, int dataPerProcess,
               int* v, const char* label, bool sync = true )
//
// Displays data stored in each process. Optionally uses
// MPI_Barrier() and usleep() to synchronize output in
// process rank order.
//
// Input:
//   int rank           - process rank
//   int numProc        - number of processes
//   int* v             - array of data to display
//   const char* label  - label for data (8 character max)
//   bool sync          - Synchronize with barrier if true
//                       (default = true)
//
// Display:
//   Integer data in array v. Displays 4 place values with
//   leading zeros.
//
```

Alltoall example dumpData (2)

```
{
    for ( int p = 0; p < numProc; p++ ) {
        if ( rank == p ) {
            // It's my turn to display data...
            printf( "Process %2d: %-8s =", rank, label );
            for ( int i = 0; i < numProc; i++ ) {
                for ( int j = 0; j < dataPerProcess; j++ ) {
                    int k = i * dataPerProcess + j;
                    printf( " %04d", v[k] );
                }
            }
            printf( "\n" );
            fflush( stdout );
        }
        if ( sync ) {
            MPI_Barrier( MPI_COMM_WORLD );
            usleep( 10000 ); // pause 0.01 seconds for I/O
        }
    }
}
```

- 1 MPI Collective communication functions
 - List of collective communication functions
 - Scattering data
 - Gathering data
 - Other collective communication routines
- 2 Example programs
 - Scatter/Gather example
 - All-to-all example
 - Vector scatter example

Vector Scatter example prolog

```
// Demonstration of MPI_Scatterv() and MPI_Gather()
//
// Demonstrate a vector scatter operation to distribute
// elements in an array of integers among the processes.
// Each process receives one more integer than its rank.
// For example, in the case of N processes:
//     Rank 0:  receives 1 integer,
//     Rank 1:  receives 2 integers,
//     ...
//     Rank N-1: receives N integers.
// Since  $1 + 2 + 3 + \dots + N = N(N+1)/2$ , the array containing
// the integers to distribute will hold  $N(N+1)/2$  integers.
//
// Each process sums the values it receives. A gather
// operation is used to collect these back on the master.

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

const int MASTER_RANK = 0; // Rank of of the master process
```


Vector Scatter example main (1)

```
int main( int argc, char* argv[] )
{
    // Initialize the MPI system
    int numProc, rank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    // The master process allocates and initializes the u[]
    // array of integers as well as arrays that hold the
    // number of items to send to each process and the
    // offsets to the start of each process' data in the
    // send buffer.
    int* u = NULL;
    int* v = NULL;
    int* sendcounts = NULL;
    int* displs = NULL;
```

Vector Scatter example main (2)

```
v = new int [rank + 1]; // receive buffer on each proc.
if ( rank == MASTER_RANK )
{
    // Master process allocates and fills the arrays
    u = new int [numProc * ( numProc + 1 ) / 2];
    sendcounts = new int [numProc];
    displs = new int [numProc];
    int k = 0;
    printf( "Master: Scattering:\n" );
    for ( int i = 0; i < numProc; i++ ) {
        for ( int j = 0; j <= i; j++ ) {
            u[k] = 100 + k;
            printf( " %4d", u[k++] );
        }
        printf( "\n" );
        sendcounts[i] = i + 1; // destination rank plus 1
        displs[i] = i * ( i + 1 ) / 2; // offset to start
    }
    printf( "\n" );
}
```

Vector Scatter example main (3)

```
// Each process (including the master) participates  
// in the vector scatter. Each process will receive  
// one more integer than their rank.  
int recvcnt = rank + 1;  
MPI_Scatterv( u, sendcounts, displs, MPI_INT,  
              v, recvcnt, MPI_INT,  
              MASTER_RANK, MPI_COMM_WORLD );  
  
// Each process sums the values they received  
int sum = 0;  
for ( int i = 0; i <= rank; i++ ) sum += v[i];  
  
// Each process participates in the gather.  
MPI_Gather( &sum, 1, MPI_INT,  
            u, 1, MPI_INT,  
            MASTER_RANK, MPI_COMM_WORLD );
```

Vector Scatter example main (4)

```
if ( rank == MASTER_RANK )
{
    // Master process displays assembled data
    printf( "Master: Received:\n" );
    for ( int i = 0; i < numProc; i++ )
    {
        printf( "u[%d] = %d\n", i, u[i] );
    }
}

// clean up
delete [] u;
delete [] v;
delete [] sendcounts;
delete [] displs;
MPI_Finalize();

return 0;
}
```

Example Output

```
$ mpiexec -n 4 scatterv
Master: Scattering:
  100 101 102 103 104 105 106 107 108 109
Master: Received:
u[0] = 100
u[1] = 203
u[2] = 312
u[3] = 430
```