Outline

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
Outline

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*:

- `MPI_Bcast()` – Broadcast (one to all)
- `MPI_Reduce()` – Reduction (all to one)
- `MPI_Allreduce()` – Reduction (all to all)
- `MPI_Scatter()` – Distribute data (one to all)
- `MPI_Gather()` – Collect data (all to one)
- `MPI_Alltoall()` – Distribute data (all to all)
- `MPI_Allgather()` – 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.
Outline

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

```c
MPI_Scatter(u, 4, MPI_INT, v, 4, MPI_INT, 0, MPI_WORLD_COMM);
```

```
u = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
v = 0 1 2 3
drew
```

```
v = 4 5 6 7
```

```
v = 8 9 10 11
```

```
v = 12 13 14 15
```

```
Rank
```

0 1 2 3 5 6 7 8 9 10 11 12 14 15
The calling sequence for MPI_Scatter() is

```c
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 × NP) data items of type sendtype where NP 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 on the left. After the all-to-all operation

\[
\text{MPI\_Alltoall}(u, 2, \text{MPI\_INT, } v, 2, \text{MPI\_INT, } \text{MPI\_WORLD\_COMM});
\]

the data will be distributed as shown on the right:

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

```c
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 \((\text{sendcount} \times NP)\) data items of type sendtype.
- Each process receives recvcount elements of type recvtype into recvbuf.
- All arguments are significant on all processes.
Outline

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 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.

```c
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 recvcount, // 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 \((\text{recvcount} \times NP)\) 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.
What will be stored in array b[] on each of four processes if each executes the following code fragment?

```c
int a = rank;
int b[4] = {0, 0, 0, 0};
MPI_Gather( &a, 1, MPI_INT,
             b, 1, MPI_INT,
             3, MPI_COMM_WORLD );
```
What will be stored in array \( b[] \) on each of four processes if each executes the following code fragment?

```c
int a = rank;
int b[4] = {0, 0, 0, 0};
MPI_Gather( &a, 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() collects data from all processes on a single process. In some instances (we’ll see an example when we talk about parallel sorting algorithms) 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:

```c
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 recvcount,         // items to receive per process
    MPI_Datatype recvtype, // type of receive buffer data
    MPI_Comm comm)         // MPI communicator to use
```

Outline

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:
  
  MPI_Scatterv()  – Distribute data (one to all)
  MPI_Gatherv()   – Collect data (all to one)
  MPI_Alltoallv() – Distribute data (all to all)
  MPI_Allgatherv()– Collect data (all to all)
MPI_Scatterv()

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

```c
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 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
```

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

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
/*
 * 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 = 0; // Rank 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;
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 nProc, pRank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &pRank );

    // 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];
if ( pRank == MASTER )
{
    // Master process allocates array and fills
    // it with data. The values in the array are
    // $100 \times (\text{rank}+1)$ plus the an offset from
    // 0..\text{SIZE\_PER\_PROCESS}.
    u = new int [ nProc \times \text{SIZE\_PER\_PROCESS} ];
    printf( "Master: Scattering =" );
    for ( int i = 0; i < nProc ; i++ )
    {
        for ( int j = 0; j < \text{SIZE\_PER\_PROCESS} ; j++ )
        {
            int k = i \times \text{SIZE\_PER\_PROCESS} + j;
            u[k] = 100 \times ( i + 1 ) + j;
            printf( "\%5d", u[k] );
        }
    }
    printf( "\n" );
}
// 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, MPI_COMM_WORLD );

// Each process, including the master, adds a  
// distinguishable value to received data.

printf( "Process %2d: ", pRank );
for ( int i = 0; i < SIZE_PER_PROCESS; i++ )
{
    printf( " (%4d", v[i] );
    v[i] += 1000 * ( pRank + 1 );
    printf( " -> %4d)", v[i] );
}
printf( "\n" );
Each process participates in the gather. Source parameters are used by each process but only the master process makes use of destination parameters.

```c
MPI_Gather(v, SIZE_PER_PROCESS, MPI_INT,
           u, SIZE_PER_PROCESS, MPI_INT,
           MASTER, MPI_COMM_WORLD);
```

```c
if ( pRank == MASTER ) {
    // Master process displays assembled data
    printf("Master: Received = " );
    for ( int i = 0; i < nProc; i++ ) {
        for ( int j = 0; j < SIZE_PER_PROCESS; j++ ) {
            int k = i * SIZE_PER_PROCESS + j;
            printf("%5d", u[k] );
        }
    }
    printf("\n");
}
```
// clean up
delete [] u;
delete [] v;
MPI_Finalize();

return 0;
}
Outline

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
/**
 * 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 <cstdio>
#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;
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 nProc, pRank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &pRank );
// Construct array of data for this process.
// The values in the array have the form RXYY 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 [nProc * SIZE_PER_PROCESS];
int* v = new int [nProc * SIZE_PER_PROCESS];
for ( int i = 0; i < nProc; i++ ) {
    for ( int j = 0; j < SIZE_PER_PROCESS; j++ ) {
        int k = i * SIZE_PER_PROCESS + j;
        u[k] = 1000 * pRank + 100 * i + j;
        v[k] = 0;
    }
}
// Display constructed data
dumpData( pRank, nProc, 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( pRank, nProc, SIZE_PER_PROCESS, v, "After" );

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

return 0;
}
void dumpData( int pRank, int nProc, 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 pRank          - process rank
//   int nProc          - 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.
for ( int p = 0; p < nProc; p++ ) {
    if ( pRank == p ) {
        // It’s my turn to display data...
        printf( "Process %2d: %-8s =", pRank, label );
        for ( int i = 0; i < nProc; 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
}
}
Outline

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
/*
   * In this program a vector scatter operation distributes
   * elements in an array of integer among the processes.
   * Each process receives one more integer than their rank.
   * 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 + ... + N = N(N+1)/2 the array containing
   * all the integers to distribute needs to 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
   * process where they are summed. This could also be done
   * more efficiently with an all-reduce.
*/
#include <stdio.h>
#include <mpi.h>

const int MASTER = 0;  // Rank of the master process
int main( int argc, char* argv[] )
{
    // Initialize the MPI system and determine the number of
    // collaborating processes and rank of current process.
    int nProc, pRank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nProc );
    MPI_Comm_rank( MPI_COMM_WORLD, &pRank );

    // 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. Each process needs array v[] to receive
    // data into.
    int* u = NULL;
    int* v = new int [pRank + 1];
    int* sendcounts = NULL;
    int* displs = NULL;
if ( pRank == MASTER )
{
    // Master process allocates and fills data array, send count array, and displacement array.
    // Use the fact that 1+2+...+N = N(N+1)/2.
    u = new int [ nProc * ( nProc + 1 ) / 2 ];
    sendcounts = new int [ nProc ];
    displs = new int [ nProc ];
    int k = 0;
    printf( "Master: Scattering = |" );
    for ( int i = 0; i < nProc; i++ ) {
        for ( int j = 0; j <= i; j++ ) {
            u[k] = 100 + k;
            printf( "%5d", u[k++] );
        }
        printf( " |" );
        sendcounts[i] = i + 1;  // destination rank plus 1
        displs[i] = i * ( i + 1 ) / 2;  // offset to start of data
    }
    printf( " |" );
    printf( "\n" );
}
// Each process participates in the vector scatter.
// Each process will receive one more integer than
// their rank.
int recvcount = pRank + 1;
MPI_Scatterv( u, sendcounts, displs, MPI_INT,
              v, recvcount, MPI_INT,
              MASTER, MPI_COMM_WORLD );

// Each process sums the values they received
int sum = 0;
for ( int i = 0; i <= pRank; i++ ) sum += v[i];

// Each process participates in the gather.
MPI_Gather( &sum, 1, MPI_INT, u, 1, MPI_INT,
            MASTER, MPI_COMM_WORLD );
if ( pRank == MASTER ) {
    // Master process displays assembled data
    sum = 0;
    printf ( "Master: Received =" );
    for ( int i = 0; i < nProc; i++ ) {
        printf ( "%5d", u[i] );
        sum += u[i];
    }
    printf ( "\nSum = %d\n", sum );
}

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

return 0;