Collective Communications

Ned Nedialkov

Department of Computing and Software
McMaster University, Hamilton, Ontario
Canada

Introduction

Collective communication involves all the processes in a communicator

We will consider

- Broadcast
- Reduce
- Gather
- Scatter

Broadcast

Broadcast: a single process sends data to all processes in a communicator

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

buffer starting address of buffer (in/out)

count number of entries in buffer

datatype data type of buffer

root rank of broadcast root

comm communicator

MPI_Bcast sends a copy of the message on process with rank root to each process in comm

Must be called in each process

Data is sent in root and received by all other processes

buffer is 'in' parameter in root and 'out' parameter in the rest of processes

Cannot receive broadcasted data with MPI_Recv

Example: reading and broadcasting data

Code adapted from P. Pacheco, PP with MPI

```
/* getdata2.c */

/* Function Get_data2
 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using three calls to MPI_Bcast.
 */
```

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

void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr,
               int my_rank)
{
   if (my_rank == 0)
      {
         printf("Enter a, b, and n\n");
         scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
      }
   MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_ptr, 1, MPI_INT,   0, MPI_COMM_WORLD);
}
```

Note: this code is not efficient

It is more efficient, e.g., to store the data at a_ptr, b_ptr, and n_ptr in one array and broadcast a single message

Then each process must extract these data

A more efficient version could be (by NN)

```
/* getdata3.c */
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr,
               int my_rank)
{
   // buffer to store a, b, and n
   char *buf = malloc(2*sizeof(float)+sizeof(int));
   float *a  = (float*)buf;
   float *b  = a+1;
   int   *n  = (int*)(b+1);

   if (my_rank == 0)
     {
       printf("Enter a, b, and n\n");
       scanf("%f %f %d", a, b, n);
     }
```

```
    MPI_Bcast(buf, 2*sizeof(float)+sizeof(int), MPI_CHAR,
              0, MPI_COMM_WORLD);

    *a_ptr = *a;
    *b_ptr = *b;
    *n_ptr = *n;

    free (buf);
}
```

Reduce

Data from all processes are combined using a binary operation

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

sendbuf address of send buffer

recvbuf address of receive buffer

count number of entries in send buffer

datatype data type of elements in send buffer

op reduce operation; predefined, e.g. MPI_MIN, MPI_SUM, or user defined

root rank of root process

comm communicator

Must be called in all processes in a communicator

Example: trapezoid with reduce

Code adapted from P. Pacheco, PP with MPI

```
/* redtrap.c */

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

extern void Get_data2(float* a_ptr, float* b_ptr,
                      int* n_ptr, int my_rank);
extern float Trap(float local_a, float local_b,
                  int local_n, float h);

int main(int argc, char** argv)
{
   int          my_rank, p;
   float        a, b, h;
   int          n;
   float        local_a, local_b, local_n;
   float        integral;   /* Integral over my interval */
   float        total;      /* Total integral            */

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   Get_data2(&a, &b, &n, my_rank);

   h = (b-a)/n;
   local_n = n/p;

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   integral = Trap(local_a, local_b, local_n, h);

   /* Add up the integrals calculated by each process */
   MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
              MPI_SUM, 0, MPI_COMM_WORLD);

   if (my_rank == 0)
     {
       printf("With n = %d trapezoids, our estimate\n", n);
       printf("of the integral from %f to %f = %f\n",
              a, b, total);
     }

   MPI_Finalize();
   return 0;
}
```

Example: dot product

Code adapted from P. Pacheco, PP with MPI

```
/* parallel_dot.c -- compute a dot product of a vector
 *  distributed among the processes.
 * Uses a block distribution of the vectors.
 * Input:
 *     n: global order of vectors
 *     x, y:  the vectors
 *
 * Output:
 *     the dot product of x and y.
 *
 * Note: Arrays containing vectors are statically allocated.
 * Assumes n, the global order of the vectors, is divisible
 * by p, the number of processes.
 */
#include <stdio.h>
#include "mpi.h"

#define MAX_LOCAL_ORDER 100

void Read_vector(char* prompt, float local_v[], int n_bar,
                 int p,int my_rank);
```

```
float Parallel_dot(float local_x[], float local_y[],
                   int n_bar);

main(int argc, char* argv[])
{
   float  local_x[MAX_LOCAL_ORDER];
   float  local_y[MAX_LOCAL_ORDER];
   int    n;
   int    n_bar;  /* = n/p */
   float  dot;
   int    p, my_rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0)
     {
       printf("Enter the order of the vectors\n");
       scanf("%d", &n);
     }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   n_bar = n/p;

   Read_vector("the first vector",
               local_x, n_bar, p, my_rank);
```

```
   Read_vector("the second vector",
               local_y, n_bar, p, my_rank);

   dot = Parallel_dot(local_x, local_y, n_bar);

   if (my_rank == 0)
      printf("The dot product is %f\n", dot);

   MPI_Finalize();
}



/*********************************************************************/
void Read_vector(
                  char*   prompt      /* in  */,
                  float   local_v[]   /* out */,
                  int     n_bar       /* in  */,
                  int     p           /* in  */,
                  int     my_rank     /* in  */)
{
   int i, q;
   float temp[MAX_LOCAL_ORDER];
   MPI_Status status;
```

```
   if (my_rank == 0)
     {
        printf("Enter %s\n", prompt);
        for (i = 0; i < n_bar; i++)
          scanf("%f", &local_v[i]);
        for (q = 1; q < p; q++)
          {
             for (i = 0; i < n_bar; i++)
               {
               scanf("%f", &temp[i]);
               printf("temp %f\n", temp[i]);
               }
             MPI_Send(temp, n_bar, MPI_FLOAT, q, 0,
                      MPI_COMM_WORLD);
          }
     }
   else
        MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0,
                 MPI_COMM_WORLD, &status);
}  /* Read_vector */
```

```
/*********************************************************/
float Serial_dot(float  x[], float  y[], int n)
{
   int     i;
   float   sum = 0.0;

   for (i = 0; i < n; i++)
     sum = sum + x[i]*y[i];
   return sum;
} /* Serial_dot */


/*********************************************************/
float Parallel_dot(float  local_x[], float  local_y[],
                   int  n_bar)
{
   float  local_dot;
   float  dot = 0.0;
   float  Serial_dot(float x[], float y[], int m);

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
              MPI_SUM, 0, MPI_COMM_WORLD);
   return dot;
} /* Parallel_dot */
```

Allreduce

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm)
```

Similar to MPI_Reduce except the result is returned to the receive buffer of each process in comm

Gather

Gathers together data from a group of processes

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

sendbuf starting address of send buffer

sendcount number of elements in send buffer

sendtype data type of send buffer elements

recvbuf address of receive buffer (significant only at root)

recvcount number of elements for any single receive (significant only at root)

recvtype data type of recv buffer elements (significant only at root)

root root rank of receiving process

comm communicator

MPI_Gather collects data, stored at sendbuf, from each process in comm and stores the data on root at recvbuf

Data is received from processes in order, i.e. from process 0, then from process 1 and so on

Usually sendcount, sendtype are the same as recvcount,recvtype

root and comm must be the same on all processes

The receive parameters are significant only on root

Amount of data sent/received must be the same

```
int MPI_Allgather(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm )
```

The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

Scatter

Sends data from one process to all other processes in a communicator

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

sendbuf starting address of send buffer (significant only at root)

sendcount number of elements sent to each process (significant only at root)

sendtype data type of send buffer elements (significant only at root)

recvbuf address of receive buffer

recvcount number of elements for any single receive

recvtype data type of recv buffer elements

root rank of sending process

comm communicator

MPI_Scatter splits data at sendbuf on root into p segments, each of sendcount elements, and sends these segments to processes 0, 1, ..., p-1 in order

Example: parallel matrix times vector

Code adapted from P. Pacheco, PP with MPI

```
/* parallel_mat_vect.c -- computes a parallel
 * matrix-vector product.
 * Matrix is distributed by block rows.
 * Vectors are distributed by blocks.
 *
 * Input:
 *     m, n: order of matrix
 *     A, x: the matrix and the vector to be multiplied
 *
 * Output:
 *     y: the product vector
 *
 * Notes:
 *     1.  Local storage for A, x, and y
 *         is statically allocated.
 *     2.  Number of processes (p) should evenly
 *         divide both m and n.
 */
```

```
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"

int main(int argc, char* argv[])
{
  int                my_rank, p;
  LOCAL_MATRIX_T     local_A;
  float              global_x[MAX_ORDER];
  float              local_x[MAX_ORDER];
  float              local_y[MAX_ORDER];
  int                m, n;
  int                local_m, local_n;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank == 0)
    {
      printf("Enter the order of the matrix (m x n)\n");
      scanf("%d %d", &m, &n);
    }
  MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
   local_m = m/p;
   local_n = n/p;

   Read_matrix("Enter the matrix",
               local_A, local_m, n, my_rank, p);
   Print_matrix("We read",
                local_A, local_m, n, my_rank, p);

   Read_vector("Enter the vector",
               local_x, local_n, my_rank, p);
   Print_vector("We read",
                local_x, local_n, my_rank, p);

   Parallel_matrix_vector_prod(local_A, m, n, local_x,
                               global_x, local_y, local_m,
                               local_n);
   Print_vector("The product is", local_y, local_m,
                my_rank, p);

   MPI_Finalize();
   return 0;
}
```

```
/* matvec.h */
#define MAX_ORDER 100

typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];

void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A,
                 int local_m, int n, int my_rank, int p);
void Read_vector(char* prompt, float local_x[],
                 int local_n, int my_rank, int p);
void Parallel_matrix_vector_prod(LOCAL_MATRIX_T local_A,
                                 int m,
                                 int n, float local_x[],
                                 float global_x[],
                                 float local_y[],
                                 int local_m, int local_n);
void Print_matrix(char* title, LOCAL_MATRIX_T local_A,
                  int local_m, int n, int my_rank, int p);
void Print_vector(char* title, float local_y[],
                  int local_m, int my_rank, int p);
```

```c
/* parmatvec.c */
#include "mpi.h"
#include "matvec.h"

void Parallel_matrix_vector_prod
( LOCAL_MATRIX_T  local_A, int m, int n,
  float local_x[], float global_x[], float local_y[],
  int   local_m, int local_n)
{
  /* local_m = m/p, local_n = n/p */
  int i, j;

  MPI_Allgather(local_x, local_n, MPI_FLOAT,
                global_x, local_n, MPI_FLOAT,
                MPI_COMM_WORLD);

  for (i = 0; i < local_m; i++)
    {
      local_y[i] = 0.0;
      for (j = 0; j < n; j++)
        local_y[i] = local_y[i] +
          local_A[i][j]*global_x[j];
    }
}
```

```c
/* readvec.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"

void Read_vector(char *prompt, float  local_x[], int local_n,
                 int my_rank, int p)
{
   int    i;
   float temp[MAX_ORDER];

   if (my_rank == 0)
     {
       printf("%s\n", prompt);
       for (i = 0; i < p*local_n; i++)
         scanf("%f", &temp[i]);
     }

   MPI_Scatter(temp, local_n, MPI_FLOAT,
               local_x, local_n, MPI_FLOAT,
               0, MPI_COMM_WORLD);

}
```

```c
/* readmat.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"


void Read_matrix(char *prompt, LOCAL_MATRIX_T  local_A,
                 int local_m, int n, int my_rank,
                 int p)
{
   int i, j;
   LOCAL_MATRIX_T  temp;

   /* Fill dummy entries in temp with zeroes */
   for (i = 0; i < p*local_m; i++)
      for (j = n; j < MAX_ORDER; j++)
         temp[i][j] = 0.0;

   if (my_rank == 0)
      {
         printf("%s\n", prompt);
         for (i = 0; i < p*local_m; i++)
            for (j = 0; j < n; j++)
               scanf("%f",&temp[i][j]);
      }
```

```
    MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT,
                local_A, local_m*MAX_ORDER, MPI_FLOAT,
                0, MPI_COMM_WORLD);
}
```

```c
/* printvec.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"

void Print_vector(char *title, float local_y[],
                  int local_m, int my_rank,
                  int p)
{
   int    i;
   float temp[MAX_ORDER];

   MPI_Gather(local_y, local_m, MPI_FLOAT,
              temp, local_m, MPI_FLOAT,
              0, MPI_COMM_WORLD);

   if (my_rank == 0)
     {
        printf("%s\n", title);
        for (i = 0; i < p*local_m; i++)
          printf("%4.1f ", temp[i]);
        printf("\n");
     }
}
```

```c
/* printmat.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"

void Print_matrix(char *title, LOCAL_MATRIX_T  local_A,
                  int local_m, int n, int my_rank, int p)
{
   int    i, j;
   float temp[MAX_ORDER][MAX_ORDER];

   MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT,
              temp, local_m*MAX_ORDER, MPI_FLOAT,
              0, MPI_COMM_WORLD);

   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++)
        {
          for (j = 0; j < n; j++)
            printf("%4.1f ", temp[i][j]);
          printf("\n");
        }
   }
}
```

General remarks

Amount of data sent must match amount of data received

Blocking versions only

No tags: calls are matched according to order of execution

A collective function can return as soon as its participation is complete