Collective Communication

The goals of this lesson are:

- understanding collective communication

1. General Considerations

As opposed to point to point communication, where only two processes where involved in the communication, collective operations involve all processes in a communicator. If one of the processes doesn't participate, the behavior is unpredictable; the entire communication operation fails.

Collective operations can can be divided in the following three categories:

1. Synchronization – execution can not continue until all processes have reached the synchronization point (barrier)
2. Data transfer – ex: broadcast, scatter, gather, all to all
3. Computation – ex: reduction – one process gathers data from all other processes and applies some mathematical operations on it.

Initially MPI collective routines were all blocking. MPI-3 has introduced non-blocking collective routines.
MPI collective routines can only be carried out on MPI predefined data types. Custom data types (which will be discussed in the next lesson) are not supported by collective routines.

2. Collective Routines

2.1 Synchronization

```
int MPI_Barrier( MPI_Comm comm )

Blocks the current process until all other processes in the current communicator
have reached this routine.

Parameters:
- comm - the communicator used in collective communication
```

2.2 Data Transfer

2.2.1 Broadcast

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

int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
               comm, MPI_Request *request)
```

Broadcasts a message from the process with rank "root" to all other processes of the communicator
Parameters:

- buffer – data to be sent
- count – size of the data buffer
- datatype – mpi data type
- root – sender
- comm – communicator
- request – request handle

2.2.2 Scatter

The root process sends equally sized data in the application buffer to all other processes in the communicator. Each process will be sent a different data segment.

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

int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm, MPI_Request *request)
```

Parameters:

- sendbuf - address of send buffer
- sendcount - number of elements sent to each process
- sendtype - data type of send buffer elements
- recvcount - number of elements in receive buffer
- recvtype - data type of receive buffer elements
- root - rank of sending process
- comm – communicator

2.2.3 Gather

The root process gathers data from all other processes and places it into a receive buffer ordered by the sender's rank.

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

int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void
               *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
               comm, MPI_Request *request)
```

Parameters:

- sendbuf - starting address of the send buffer
- sendcount - number of elements in send buffer
- sendtype - data type of send buffer elements

- recvcount -number of elements in receive buffer
- recvtype - data type of receive buffer elements
- root - rank of receiving process
- comm – communicator
- recvbuf - starting address of the receive buffer
- request - communication request

See also: **MPI_Allgather**

2.2.3 All to all

Sends data from each process to all other processes. The receive buffer is built up with respect to the ranks.

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

```
int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int
            recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

Parameters:
- sendbuf - starting address of the send buffer
- sendcount - number of elements in send buffer
- sendtype - data type of send buffer elements
- recvcount - number of elements received from any process
- recvtype - data type of receive buffer elements
- comm – communicator
- recvbuf - starting address of the receive buffer
- request - communication request

2.3 Computation

2.3.1 MPI Reduce

Combines the values sent by all processes using a predefined operator and places result in the receive buffer of the root process.

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

int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype
            datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)
```

Parameters:
- sendbuf - address of the send buffer
- count - number of elements in send buffer
- datatype - data type of elements of send buffer

- op - reduce operation
- root - rank of root process
- comm – communicator
- recvbuf - address of the receive buffer
- request - communication request

See also: MPI_Allreduce, MPI_Reduce_scatter

Predefined MPI operations:

MPI Operation	Description
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR

3. Examples

Example 1: Scatter. The program splits and array and sends one line to each process

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4

int main (int argc, char *argv[])
{
	int numprocs, rank, sendcount, recvcount, source;
	float sendbuf[SIZE][SIZE] = {
		{1.0, 2.0, 3.0, 4.0},
		{5.0, 6.0, 7.0, 8.0},
		{9.0, 10.0, 11.0, 12.0},
		{13.0, 14.0, 15.0, 16.0}  };
	float recvbuf[SIZE];
```

```
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

        if (numprocs == SIZE) {
                source = 1;
                sendcount = SIZE;
                recvcount = SIZE;
                MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,MPI_FLOAT,source,
                        MPI_COMM_WORLD);

                printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0], recvbuf[1],recvbuf[2],
                        recvbuf[3]);
        } else
                printf("Must specify %d processors. Terminating.\n",SIZE);

        MPI_Finalize();
}
```

Example 2: Gather. The root process receives randomly generated arrays from each process.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include "mpi.h"
#define MAXPROC 8    /* Max number of procsses */
#define NAMELEN 80   /* Max length of machine name */
#define LENGTH 24    /* Lengt of send buffer is divisible by 2, 4, 6 and 8 */

int main(int argc, char* argv[]) {
  int i, j, np, me;
  const int nametag  = 42;    /* Tag value for sending name */
  const int datatag  = 43;    /* Tag value for sending data */
  const int root = 0;         /* Root process in scatter */
  MPI_Status status;          /* Status object for receive */

  char myname[NAMELEN];             /* Local host name string */
  char hostname[MAXPROC][NAMELEN];  /* Received host names */

  int x[LENGTH];        /* Send buffer */
  int y[LENGTH];        /* Receive buffer */

  MPI_Init(&argc, &argv);                /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np);    /* Get nr of processes */
  MPI_Comm_rank(MPI_COMM_WORLD, &me);    /* Get own identifier */
```

```
gethostname(myname, NAMELEN);    /* Get host name */

/* Check that we have an even number of processes and at most MAXPROC */
if (np>MAXPROC || np%2 != 0) {
  if (me == 0) {
    printf("You have to use an even number of processes (at most %d)\n", MAXPROC);
  }
  MPI_Finalize();
  exit(0);
}

/* Each process initializes its local array */
for (i=0; i<LENGTH/np; i++) {
  x[i] = (LENGTH/np)*me+i;
}

if (me == 0) {    /* Process 0 does this */

  printf("Process %d on host %s is gathering array x from all %d processes\n\n", \
         me, myname, np);

  /* Gather the array x from all proceses, place it in y */
  MPI_Gather(x, LENGTH/np, MPI_INT, y, LENGTH/np, MPI_INT, root, MPI_COMM_WORLD);

  /* Print out the gathered array */
  printf("Process %d on host %s got elements\n", me, myname);
  for (i=0; i<LENGTH; i++) {
    printf(" %d", y[i]);
  }
  printf("\n\n");

  /* Print out the local array x on process 0 */
  printf("Process %d on host %s had elements", me, myname);
  for (i=0; i<LENGTH/np; i++) {
    printf (" %d", x[i]);
  }
  printf ("\n");

  /* Receive messages with hostname and the original data */
  /* from all other processes */
  for (i=1; i<np; i++) {
    MPI_Recv (&hostname[i], NAMELEN, MPI_CHAR, i, nametag, MPI_COMM_WORLD, \
              &status);
    MPI_Recv (&y, LENGTH/np, MPI_INT, i, datatag, MPI_COMM_WORLD, &status);
    printf("Process %d on host %s had elements", i, hostname[i]);
```

```
 for (j=0; j<LENGTH/np; j++) {

       printf(" %d", y[j]);
     }
     printf("\n");
   }

   printf("Ready\n");


 } else { /* all other processes do this */

   MPI_Gather(&x, LENGTH/np, MPI_INT, &y, LENGTH/np, MPI_INT, root, \
              MPI_COMM_WORLD);
   /* Send own name back to process 0 */
   MPI_Send (&myname, NAMELEN, MPI_CHAR, 0, nametag, MPI_COMM_WORLD);
   /* Send the array to process 0 */
   MPI_Send (&x, LENGTH/np, MPI_INT, 0, datatag, MPI_COMM_WORLD);

 }

 MPI_Finalize();
 exit(0);
}
```

Example 3: Broadcast, Reduce. The root process loads data from file and broadcasts it. Each process computes a partial sum. The final sum is aggregated by reducing the partial results.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 100

int main(int argc, char **argv)
{
        int myid, numprocs;
        int data[MAXSIZE], i, x, low, high, myresult=0, result;
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
	if(0 == myid) {
		/* open input file and intialize data */
		if( NULL == (fp = fopen("file.txt", "r")) ) {
			printf("Can't open the input file.");
			exit(1);
		}
		for(i=0; i<MAXSIZE; i++) {
			fscanf(fp, "%d", &data[i]);
		}
	}

	/* broadcast data */
	MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);

	/* add portion of data */
	x = MAXSIZE/numprocs;   /* must be an integer */
	low = myid * x;
	high = low + x;
	for(i=low; i<high; i++) {
		myresult += data[i];
	}
	printf("I got %d from %d\n", myresult, myid);

	/* compute global sum */
	MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
	if(0 == myid) {
		printf("The sum is %d.\n", result);
	}

	MPI_Finalize();
}
```

4. Exercises:

1. Write a program that searches an element inside an array.

a. Use MPI_Broadcast for sending the array. If the element is found, print the maximum position index. For computing the maximum position, you need to use MPI_Reduce.

b. Use scatter for sending the array. If the element is found many times, print all its positions. Use MPI_Gather for sending back the positions.