Advanced Point-to-Point Communications

Ned Nedialkov

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

• Communication modes
  – standard
  – buffered
  – synchronous
  – ready

• Persistent communications

• Buffered mode (in more detail)

• Send-Receive
Communication modes

Standard mode

It is up to MPI to decide whether outgoing messages will be buffered.

With buffering, send may complete before a receive is posted.

If no buffering is available, the send will not complete until a matching receive has been posted and the data has been moved to the receiver.

A blocking send completes when the call returns; a nonblocking send completes when a matching Wait or Test call returns successfully.

Thus, a send can be started whether or not a matching receive has been posted.
Buffered mode

A buffered-mode send can be started whether or not a matching receive has been posted

It may complete before a matching receive is posted

Buffer space is provided by the application

An error occurs if a buffered-mode send is called and there is insufficient buffer space
Synchronous mode

A synchronous-mode send can be started whether or not a matching receive has been posted.

It completes only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send.

A communication does not complete at either end before both processes rendezvous at the communication.
Ready mode

A ready-mode send may be started only if a matching receive has already been posted.
Otherwise, the outcome is undefined.

On some systems ready-mode allows the removal of the hand-shake operation and results in improved performance.
Prefixes

Three additional send functions are provided for the three additional communication modes

- **B** buffered
- **S** synchronous
- **R** ready

There is only one receive mode and it matches any of the send modes
Persistent communications

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation.

With persistent communications, one may be able to optimize for performance by

- binding the list of communication arguments to a persistent communication request once and
- repeatedly using the request to initiate and complete message communication.

Persistent communications can minimize the software overhead associated with redundant message setup.

Persistent communication routines are non-blocking.

Using persistent communications is a four-step process.
Persistent communications: steps

Step 1: Create persistent requests

The desired routine is called to setup buffer location(s) which will be sent/received

Available routines are:

- `MPI_Send_init` creates a persistent standard send request
- `MPI_Bsend_init` creates a persistent buffered send request
- `MPI_Ssend_init` creates a persistent synchronous send request
- `MPI_Rsend_init` creates a persistent ready send request
- `MPI_Recv_init` creates a persistent receive request
Step 2: Start communication transmission

Data transmission is begun by calling either of the MPI_Start routines:

- **MPI_Start** activates a persistent request operation
- **MPI_Startall** activates a collection of persistent request operations

Step 3: Wait for communication completion

Because persistent operations are non-blocking, the appropriate **MPI_Wait** or **MPI_Test** routine must be used to insure their completion

Step 4: Deallocate persistent request objects

When there is no longer a need for persistent communications, the programmer should explicitly free the persistent request objects using the **MPI_Request_free()** routine
Example

Adapted from http://www.llnl.gov/computing/tutorialsmpi_performance/samples/persist.c

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

/* Modify these to change timing scenario */
#define TRIALS 10
#define STEPS 15
#define MAX_MSGSIZE (1<<STEPS)  /* 2^STEPS */
#define REPS 1000
#define MAXPOINTS 10000

char sbuff[MAX_MSGSIZE], rbuff[MAX_MSGSIZE];
int msgsizes[MAXPOINTS];
double results[MAXPOINTS];

int main(int argc, char *argv[])
{

int numtasks, rank, tag = 999;
int n, i, j, k;

double mbytes, tbytes, ttime, t1, t2;

MPI_Status stats[2];
MPI_Request reqs[2];

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

/*** task 0 **************************************************/
if (rank == 0)
{
    /* Greetings */
    printf("\n********* Persistent Communications*********\n");
    printf(" Trials=%8d\n", TRIALS);
    printf(" Reps/trial=%8d\n", REPS);
    printf(" Message Size Bandwidth (bytes/sec)\n");
}
/* Initializations */
n = 1;
for (i = 0; i <= STEPS; i++)
{
    msgsizes[i] = n;
    results[i] = 0.0;
    n = n * 2;
}
for (i = 0; i < MAX_MSGSIZE; i++)
sbuff[i] = 'x';

/*/ Begin timings */
for (k = 0; k < TRIALS; k++)
{
    n = 1;
    for (j = 0; j <= STEPS; j++)
    {
        /* Setup persistent requests for both the send and receive */
        MPI_Recv_init (&rbuff, n, MPI_CHAR, 1, tag,
                        MPI_COMM_WORLD, reqs);
        MPI_Send_init (&sbuff, n, MPI_CHAR, 1, tag,
                        MPI_COMM_WORLD, reqs + 1);
    }
t1 = MPI_Wtime();
for (i=1; i<=REPS; i++)
{
    MPI_Startall (2, reqs);
    MPI_Waitall (2, reqs, stats);
}
t2 = MPI_Wtime();

/* Compute bandwidth and save best result over all TRIALS */
ttime = t2 − t1;
tbytes = sizeof(char) * n * 2.0 * (float)REPS;
mbytes = tbytes/ttime;
if (results[j] < mbytes)
    results[j] = mbytes;

/* Free persistent requests */
MPI_Request_free (reqs);
MPI_Request_free (reqs+1);
n=n*2;
}    /* end j loop */
}    /* end k loop */

/* Print results */
for (j=0; j< STEPS; j++) {
    printf("%9d %16d\n", msgsizes[j], (int) results[j]);
}

} /* end of task 0 */

/**** task 1 ****************************/
if (rank == 1)
{
    /* Begin timing tests */
    for (k=0; k< TRIALS; k++)
    {
        n=1;
        for (j=0; j< STEPS; j++)
        {
            /* Setup persistent requests for both the send and receive */
            MPI_Recv_init (&rbuff, n, MPI_CHAR, 0, tag,
                            MPI_COMM_WORLD, &reqs[0]);
            MPI_Send_init (&sbuff, n, MPI_CHAR, 0, tag,
                           MPI_COMM_WORLD, &reqs[1]);

            for (i=1; i< REPS; i++)
            {
                MPI_Startall (2, reqs);
            }
        }
    }
}
MPI_Waitall (2, reqs, stats);

} /* Free persistent requests */
MPI_Request_free (&reqs[0]);
MPI_Request_free (&reqs[1]);
n=n*2;
} /* end j loop */
} /* end k loop */
} /* end task 1 */

MPI_Finalize();

return 0;
} /* end of main */
Buffered mode

An application must specify a buffer to be used for buffering messages in buffered mode

Buffering is done by sender

- `MPI_Buffer_attach` allocates user buffer space
- `MPI_Buffer_detach` frees user buffer space
- `MPI_Bsend` buffer send, blocking
- `MPI_Ibsend` buffer send, non-blocking
Example

Adapted from http://www.llnl.gov/computing/tutorials/mpi_performance/samples/buffsend.c

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

#define NELEM 100000

int main(int argc, char *argv[])
{
    int numtasks, rank, rc, i,
        dest = 1, tag = 111, source = 0, size;
    double data[NELEM];
    void   *buffer;
    MPI_Status status;

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

    if (numtasks != 2)
    ```

{
    printf("Please run this test with 2 tasks."
            " Terminating\n")
    MPI_Finalize();
}
printf("MPI task %d started...\n", rank);

/*** Send task *****************************************************/
if (rank == 0)
{
    /* Initialize data */
    for (i=0; i<NELEM; i++)
        data[i] = (double)random();

    /* Determine size of buffer needed including any required MPI overhead */
    MPI_Pack_size (NELEM, MPI_DOUBLE,
                   MPI_COMM_WORLD, &size);
    size = size + MPI_BSEND_OVERHEAD;
    printf("Using buffer size=%d Overhead=%d\n", size, MPI_BSEND_OVERHEAD);

    /* Attach buffer, do buffered send, and then detach buffer */
buffer = (void*)malloc(size);
rc = MPI_Buffer_attach(buffer, size);
if (rc != MPI_SUCCESS)
{
    printf("Buffer_attach failed. Return code=%d\n" "Terminating\n", rc);
    MPI_Finalize();
}
rc = MPI_Bsend(data, NELEM, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
printf("Sent message. Return code=%d\n", rc);
MPI_Buffer_detach(&buffer, &size);
free(buffer);

/*** Receive task ****************************/
if (rank == 1)
{
    MPI_Recv(data, NELEM, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
    printf("Received message. Return code=%d\n", rc);
}
MPI_Finalize();
return 0;
Send-Receive

A send-receive operation combines, in one call, sending of one message to a destination and receiving of another message from a source.

Useful for communications patterns, where each node both sends and receives messages:

- Exchange of data between two processes
- Shift operation across a chain of processes

A message send by send-receive can be received by a regular receive and vice versa.
MPI_Sendrecv

```c
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

Executes blocking send and receive

Same communicator, different tags

Send and receive buffers must be disjoint and may have different lengths and datatypes

See also MPI_Sendreceive_replace