Understanding Communications

Ned Nedialkov

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

Outline

- Buffering
- Safe programs
- Send modes and protocols
- Performance issues

Buffering

Suppose we have

```
if (rank == 0)
   MPI_Send(sendbuf,..., 1, ...)
if (rank == 1)
   MPI_Recv(recvbuf,..., 0, ...)
```

Assume that process 1 is not ready to receive

There are 3 possibilities for process 0

(a) stops and waits until process 1 is ready to receive

(b) copies the message at sendbuf into a system buffer (can be on process 0, process 1, or somewhere else) and returns from MPI_send

(c) fails

- As far as buffer space is available, (b) is a reasonable alternative
- An MPI implementation is permitted to copy the message to be sent into internal storage, but it is not required to do so
- What if not enough space is available?
  - In applications communicating large amounts of data, there may not be enough memory (left) in buffers
  - Until receive starts, no place to store the send message
  - Practically, (a) results in a serial execution
- A programmer should not assume that the system provides adequate buffering

Example

Consider a program executing

process 0	process 1
MPI_Send to process 1	MPI_Send to process 0
MPI_Recv from process 1	MPI_Recv from process 0

Such a program may work in many cases, but it is certain to fail for message of some size that is large enough

Example Cont.

Try executing with various BUF_SIZE

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

#define BUF_SIZE 100

int main(int argc, char* argv[])
{
   int my_rank, npes;
   char *sendbuf, *recvbuf;
   MPI_Status status;

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

   sendbuf = malloc(BUF_SIZE);
   recvbuf = malloc(BUF_SIZE);
```

```
    if (my_rank==0)
      {
        MPI_Send(sendbuf, BUF_SIZE, MPI_CHAR, 1,
                 0,MPI_COMM_WORLD);
        MPI_Recv(recvbuf, BUF_SIZE, MPI_CHAR, 1,
                 0,MPI_COMM_WORLD,&status);
      }
    if (my_rank==1)
      {

        MPI_Send(sendbuf, BUF_SIZE, MPI_CHAR, 0,
                 0,MPI_COMM_WORLD);
        MPI_Recv(recvbuf, BUF_SIZE, MPI_CHAR, 0,
                 0,MPI_COMM_WORLD,&status);
      }

    free(recvbuf);
    free(sendbuf);
    printf(" *** PE %d done \n", my_rank);

    MPI_Finalize();
    return 0;
}
```

Techniques

Ordered send and receive. If a process is sending to another, the destination will do a receive that matches the send before doing a send on its own

Pairing send and receive. Pairing by ordering can be difficult in complex applications

An alternative is to use `MPI_Sendrecv`. It performs both send and receive such that if no buffering is available, no deadlock will occur

Buffered sends. MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in buffer)

See `MPI_Bsend` and `MPI_Brecv`

Nonblocking communication. With buffering, a send may return before a matching receives is posted

With no buffering, communication is deferred until a place for receiving is provided

Safe programs

Synchronous send: `MPI_Ssend`

Does not return until a matching receive has been posted and the matching receive has begun reception of the data

Parameters in `MPI_Ssend` and `MPI_Send` are the same— no dependence on system buffering if `MPI_Ssend` is used

For portability, one can try replacing all `MPI_Send` with `MPI_Ssend`
(Try this in the previous program)

Safe programs: do not require buffering for their correct operation

Write safe programs using

- matching send with receive
- `MPI_Sendrecv`
- allocating own buffers
- nonblocking operations

Remarks

Nonblocking operations are more important for ensuring safe programs than for improving performance

Many implementations cannot overlap communication with computation without extra hardware in the form of a communication processor (or controller)

That is, overlapping also depends on the hardware environment

Nonblocking operations allow an implementation to obtain extra performance, by overlapping communication with computation, but it is not required to

Sending data

Sending a message consists of sending an envelope followed by the data

An envelope contains information such as tag, communicator, length, source, and destination

Sending data immediately is called an eager protocol

Receiving data: matching receive exists

Receiver provides a location for the data arriving behind the envelope

MPI implementation can maintain a queue of receives that have been posted

That is, a queue of messages that are expected

When an incoming message matches a receive in this queue, the receive is marked as completed and removed from the queue

Receiving data: no matching receive exists

Receiving process must remember that a message has arrived, and it must store the data somewhere

We keep a queue of messages that are unexpected

When `MPI_Recv` is called, it first checks the unexpected queue. If the message is there, it can remove it from the queue, copy the data, and complete

However, the receiving process must store the data somewhere

What if the message is too big?

Rendezvous protocol

Send only the envelope to the destination process

When the receiver wants the data (and has a place to put the data), it tells the sender "send me the data"

The sender can send the data

This is the rendezvous protocol

What if too many envelopes arrive?

MPI has a limit, reasonably large, for the number of unexpected messages that can be handled

Performance issues

The rendezvous protocol can handle arbitrary large messages in any number

The eager protocol can be faster for shorter messages

Possible mapping of MPI send modes onto eager and rendezvous protocols could be

MPI call	message size	protocol
MPI_Send	$\leq$ 16KB	eager
MPI_Send	$>$ 16KB	rendezvous
MPI_Ssend	any	rendezvous always
MPI_Rsend	any	eager always

`MPI_Rsend` is sending in ready mode, when we know that a receive has been posted

Most MPI implementations provide some parameters to allow MPI users to tune for performance