ANL/MCS-TM-number

MPICH Working Note: Creating a new MPICH device using the Channel interface

by

William Gropp
Ewing Lusk

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Abstract

The MPICH implementation of MPI uses a powerful and efficient layered approach to simplify porting MPI to new systems. One interface that can be used is the *channel* interface; this interface defines a collection of simple data-transfer operations. This interface can adapt to additional functionality, such as asynchronous or nonblocking transfers or remote memory operations. This paper describes this interface, some of the special issues, and gives instructions on creating a new MPICH implementation by implementing just a few routines.

1 Introduction

These operations are based on a simple ability to send data from one process to another process. No more functionality is required than what is provided by Unix in the `select`, `read`, and `write` operations. The 'ADI' code uses these simple operations to provide the operations, such as `MPID_Post_recv`, that are used by the MPI implementation.

There are a number of very subtle issues having to do with how much buffering is expected of the routines defined here, and exactly how large messages are transmitted. Rather than cover all of the subtle issues first, we first present a simplified interface that ignores these issues. This interface is then refined in the rest of the paper. In order to make the definition more concrete, we present an implementation in terms of MPI calls in Section 5.

The issue of buffering is a difficult one, and we could have defined an interface that assumed no buffering, requiring the ADI that calls this interface to perform the necessary buffer management and flow-control. The rationale for not making this choice is that many of the systems that are used for implementing the interface defined here maintain their own internal buffers and flow controls, and implementing another layer of buffer management can impose an unnecessary performance penalty.

2 The bare minimum

The simplest set of routines sends or receives data from another process. All MPI message passing is converted into these lower-level messages (or data exchanges).

Messages are sent in two parts: a *control part*, containing information on the MPI message tag, size, and communicator, as well as information about the message itself, and the *data part*, containing the actual data. There are separate routines to send and receive the control and data parts, along with a routine to check to see if any control messages are available.

In order to reduce the latency of short messages, small amounts of data may be sent with the control part of the message instead of being sent in a separate data part.

`MPID_ControlMsgAvail` Indicates whether a control message is available.

`MPID_RecvAnyControl` Reads the next control message. If no messages are available, blocks until one can be read.

`MPID_SendControl` Sends a control message.

`MPID_RecvFromChannel` Receives data from a particular channel.

`MPID_SendChannel` Sends data on a particular channel.

In Unix terms, `MPID_ControlMsgAvail` is similar to `select` (with a `fd` mask of all the file descriptors that can be read for control messages), and `MPID_RecvAnyControl` is like a `select` followed by a `read`, while the others are similar to `read` and `write` on the appropriate file descriptors.

The bindings are

```
int MPID_ControlMsgAvail( void )
void MPID_RecvAnyControl( MPID_PKT_T *pkt, int size, int *from )
void MPID_SendControl( MPID_PKT_T *pkt, int size, int dest )
void MPID_RecvFromChannel( void *buf, int maxsize, int from )
void MPID_SendChannel( void *buf, int size, int dest )
```

In these calls, `void *buf, int size` is a buffer (of contiguous bytes) of `size` bytes. The int `dest` is the destination process (`dest` is the rank in `MPI_COMM_WORLD` of the destination process). The value `from` is the source of the message (also relative to `MPI_COMM_WORLD`). The value `MPID_PKT_T *pkt` is a pointer to a control message (of type `MPID_PKT_T`; this is called a *control packet* or *packet* for short, and is defined in the file '`packet.h`').

2.1 Buffering issues

For correct operation of the channel interface, it is imperative that send operations not block; that is, the completion of an `MPID_SendControl` should not require that the destination processor read the control data before the `MPID_SendControl` can complete.[1]

This is because a control message is sent for any kind of MPI message, whether it is `MPI_Send`, `MPI_Isend`, `MPI_Ssend`, etc. However, in some cases, it may be more efficient to *not* require that the control message be delivered without requiring a matching receive. The routine `MPID_SendControlBlock` may be used for this. If this routine is not provided, then `MPID_SendControl` will be used instead.

The binding for `MPID_SendControlBlock` is the same as for `MPID_SendControl`.

A slight complication is the fact that the control packet itself may contain the message data, if that data is small enough. The reason for this is that it is more efficient to send the data with the control information if the data length is short enough (and doing so does not cause the sending of a control message to block). The size of amount of data that will be sent in a control packet, called the *payload*, can be controlled at configure time with the option `-pkt_size=n`. For example, `-pkt_size=4` will limit the payload to four bytes. The length must be greater than zero. The configure option `-var_pkt` allows you to control the allowed payload size at run time (upto the limit defined with `-pkt_size` or the default value). For most devices, the default payload size is 1024 bytes.

If you can not guarentee that sends will not block, then consider using a send that returns an indication that it would block. In this case, let the ADI process some incoming messages (by calling `MPID_check_incoming( MPID_NOTBLOCKING)` , and then trying to send again. In Unix terms, this is similar to checking for and `EWOULDBLOCK` return from `write`.[2]

Also see Section 4.2.

2.2 Message ordering

Control messages between any pair of processors must arrive in the order in which they were sent. There is no required ordering of messages sent by different processors (but see the discussion on fairness below).

When a message is sent using `MPID_SendControl` and `MPID_SendChannel`, it is guarenteed that the sending process performs the `MPID_SendChannel` after the `MPID_SendControl` for that message and before performing a `MPID_SendControl` for any other message. In other words, if the connection between two processes is a stream, then the data part of a message immediately follows the control part. This applies only when using the eager protocol, described below.

3 Using nonblocking operations

Nonblocking operations provide the ability to both provide greater efficiency through the overlap of computation and communication and greater robustness by allowing some data transfers to be left uncompleted until they can be received at their destination. These are not required, but can be used if they are available.

If they are *not* available, the macros `PI_NO_NRECV` and `PI_NO_NSEND` must be defined.

`MPID_IRecvFromChannel( buf, size, partner, id )` Start a nonblocking receive of data from `partner`. The value `id` is used to complete this receive, and is an output parameter from this call.

`MPID_WRecvFromChannel( buf, size, partner, id )` Complete a nonblocking receive of data. Note that in many implementations, only the value of `id` will be used.

[1] This brings up the infamous progress rule; for the purposes of this discussion, it is enough to consider only the case of two processes and a single MPI communicator.

[2] In HP-UX systems, that's `EAGAIN`.

`MPID_RecvStatus( id )` Test for completion of a nonblocking receive.

`MPID_ISendChannel( buf, size, partner, id )` Start a nonblocking send of data to `partner`. The value of `id` is used to complete this send, and is an output parameter from this call.

`MPID_WSendChannel( buf, size, partner, id )` Complete a nonblocking send of data. Note that in many implementations, only the value of `id` will be used.

`MPID_TSendChannel( id )` Test for completion of a nonblocking send of data.

4 Different data exchange mechanisms

This section describes the different ways in which data can be transfered from one process to another. The advantages and disadvantages of each are also described here.

4.1 Eager

In the *eager* protocol, data is sent to the destination immediately. If the destination is not expecting the data (e.g., no `MPI_Recv` has yet been issued for it), the receiver must allocate some space to store the data locally.

This choice often offers the highest performance, particularly when the underlying implementation provides suitable buffering and handshakes. However, it can cause problems when large amounts of data are sent before their matching receives are posted, causing memory to be exhausted on the receiving processors.

This is the default choice in MPICH.

4.2 Rendevous

In the *rendevous* protocol, data is sent to the destination only when requested (the control information describing the message is always sent). When a receive is posted that matches the message, the destination sends the source a request for the data. In addition, it provides a way to for the sender to return the data.

This choice is the most robust but, depending on the underlying system software, may be less efficient than the eager protocol. Also, some legacy programs may fail when run using a rendevous protocol. This is really a combination of errors in the program (relying on an exhaustible resource) and inefficient implementations of `MPI_Bsend`.

This choice can be selected with the configure option `-use_rndv`.

4.3 Get

In the *get* protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process's memory to another. A typical implementation might use `memcpy`.

This choice offers the highest performance, but requires special hardware support such as shared memory or remote memory operations. In many ways, it functions like the rendevous protocol, but uses a different set of routines to transfer the data.

To implement this protocol, special routines must be provided to prepare the address for remote access and to perform the transfer. The implementation of this protocol allows data to be transfered in several pieces, for example, allowing arbitrary sized messages to be transfered using a limited amount of shared memory.

5 Using message-passing systems

The first implementations of MPICH were on top of existing message-passing systems, and these are still some of the most important implementations. This section gives an example by showing how the basic MPICH Channel interface can be implemented. Rather than use a particular vendor's system, we use a few MPI routines within `MPI_COMM_WORLD`. These do not illustrate the macros necessary to use the get protocol, since MPI does not support remote memory operations.

```
int flag;
MPI_Status status;
#define MPID_RecvAnyControl( pkt, size, from ) \
    { MPI_Recv(pkt,size,MPI_BYTE,MPI_ANY_SOUCE,0,MPI_COMM_WORLD,\
               &status); \
      *from = status.MPI_SOURCE;}
#define MPID_RecvFromChannel( buf, size, channel ) \
      MPI_Recv(buf,size,MPI_BYTE,channel,1+channel, channel, \
               MPI_COMM_WORLD, &status );
#define MPID_ControlMsgAvail( ) \
    (MPI_Iprobe( MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, \
                 &flag, &status),flag)
#define MPID_SendControl( pkt, size, channel ) \
      MPI_Bsend(pkt,size,MPI_BYTE,channel,0, MPI_COMM_WORLD);
#define MPID_SendControlBlock( pkt, size, channel ) \
      MPI_Send(pkt,size,MPI_BYTE,channel,0, MPI_COMM_WORLD);
#define MPID_SendChannel( buf, size, channel ) \
      MPI_Send(buf,size,MPI_BYTE,channel,1+channel, \
               MPI_COMM_WORLD);
```

Note the use of `MPI_Bsend` instead of `MPI_Send` when sending control packets. This emphasizes that sending a control packet should not block the program.

The nonblocking operations are very easy; the communication `id` is just an `MPI_Request` object.

```
#define MPID_IRecvFromChannel( buf, size, channel, id ) \
    MPI_IRecv( buf, size, MPI_BYTE, channel, 1+channel,
               MPI_COMM_WORLD, &(id) )
#define MPID_WRecvFromChannel( buf, size, channel, id ) \
    MPI_Wait( &(id), &status )
#define MPID_RecvStatus( id ) \
    (MPI_Test( &(id), &flag, &status ), flag)
#define MPID_ISendChannel( buf, size, channel, id ) \
     MPI_Isend( buf, size, MPI_BYTE, channel, 1+myid,
        MPI_COMM_WORLD, &(id) );
#define MPID_WSendChannel( buf, size, channel, id ) \
    MPI_Wait( &id, &status )
#define MPID_TSendChannel( id ) \
    (MPI_Test( &(id), &flag, &status ), flag)
```

The out-of-band operations are also very simple; they are just nonblocking MPI calls with a message tag that is allocated for each operation. Note the use of the MPI “ready send” to return the requested data.

```
#define MPID_CreateSendTransfer( buf, size, partner, id ) {*(id) = 0;}
#define MPID_CreateRecvTransfer( buf, size, partner, id ) \
       {*(id) = CurTag++;TagsInUse++;}

#define MPID_StartRecvTransfer( buf, size, partner, id, rid ) \
     MPI_Irecv( buf, size, MPI_BYTE, partner, id, MPI_COMM_WORLD, &(rid) )
#define MPID_EndRecvTransfer( buf, size, partner, id, rid ) \
     { MPI_Wait( &(rid), &status );\
     if (--TagsInUse == 0) CurTag = 1024; else if (id == CurTag-1) CurTag--;}
#define MPID_TestRecvTransfer( rid ) \
    (MPI_Test( &(rid), &flag, &status ), flag)

#define MPID_StartSendTransfer( buf, size, partner, id, sid ) \
     MPI_Irsend( buf, size, MPI_BYTE, partner, 1+id,
```

```
                 MPI_COMM_WORLD, &(sid) )
#define MPID_EndSendTransfer( buf, size, partner, id, sid ) \
    MPI_Wait( &(sid), &status )
#define MPID_TestSendTransfer( sid ) \
    (MPI_Test( &(sid), &flag, &status ), flag)
```

Note that this code makes little attempt to reuse the special tags; if, however, all tags are accounted for, it resets the CurTag counter to the initial value. More sophisticated (and costly) methods could keep track of the tags in use. More careful code would check for tag overflow. These have been left out to simplify the example.

6 Polling for data

MPI was designed to allow a variety of implementation strategies. In particular, it does not require 'immediate' response to an MPI_Recv (which might require an interrupt). However, this strategy may require the device to be *polled* relatively frequently. The ADI macro MPID_CHECK_DEVICE should be used for this; if your new device needs polling, you should modify this macro.

Currently, there is no provision for providing periodic polling (e.g., using SIGALRM). Note that since Unix makes no provision for sharing or chaining signal handlers, any package that uses a Unix signal effectively both denies the use of that signal to other libraries and to the user, and depends on no libraries or the user redefining the signal(s). This was judged too fragile and anti-social to depend upon.

7 Special packet data

The format of the control message has some optional elements that can be included

MPID_PKT_INCLUDE_LINK Provide a pointer to MPID_PKT_T in the packet.

MPID_PKT_INCLUDE_LEN Provide the length of the packet in bytes

MPID_PKT_INCLUDE_SRC Include to source (rank of the sender in MPI_COMM_WORLD in the packet.

You can use MPID_PKT_LEN_GET(MPID_PKT_T *)pkt,len) to set len to the number of bytes in the packet (note that this is a macro with the effect l = (pkt)->pkt_len). A sample that uses this is in 'mpich/mpid/ch_tcp/tcppriv.c'.

See 'packet.h' for more details on the packet formats and macros.

8 Implementation

One of the main advantages of the channel device is that any changes (bug fixes or performance improvements) to the ADI or MPI routines can be tracked nearly automatically.

The command NewDevice may be used to create a new ADI device that is based on this channel interface. The commands

```
cd mpich/mpid
NewDevice -raw mydev
```

create a directory 'ch_mydev' that contains a new device.[3] You may need to make some small changes to several of the files (in addition to 'channel.h'). To configure an MPICH for this new device, simply use

```
configure -device=ch_mydev
```

[3] When -raw switch is not present, NewDevice attempts to use a message-passing version of channel.h and to automatically replace the message passing calls in the channel.h file. This requires you to have a version of Chameleon for the message passing system.

This will work as long as configure can correctly deduce the compiler, options, and libraries. If you need special options, you may need to modify '`configure`' directly (if you have `autoconf` version 1.6, you can modify '`configure.in`' instead and regenerate the '`configure`' file from that).

Now, edit the following files: '`channel.h`' (definitions of routines described in this document), '`mpid.h`' (features of packets), '`mydevpriv.c`' (any routines you need).

9 Commentary

9.1 Limitations

1. The rendevous, get, and eager protocols are mutually exclusive.
2. Control packets must *not* block (the exception is for `MPID_SendControlBlock`).
3. The eager protocol allocates space (potentially large) for unexpected messages as they arrive.
4. Currently, you must handle sending and receiving messages from yourself.

9.2 Advantages

1. No signals used (other than those used by the routines that the "channel" operations are defined with).
2. High performance if adequate buffering supported by the channel operations.
3. No special code is required for the MPI synchronous send, ready send, non-blocking, or persistent operations
4.

10 Special considerations

10.1 Fairness

The implementation of `MPID_RecvAnyControl` should be *fair*; that is, if messages are available from more than one sender, there should be no bias in selecting which one is received.

This can be difficult to provide efficiently on some systems.

10.2 Error handling

The MPI standard gives an implementation wide latitude on handling errors. In the MPICH implementation, there are a few cases that should be handled by the channel interface. The most important case is that of message truncation: receiving more data than will fit in the user-specified buffer.

The MPID device definition will probably change in the near future to place MPI error codes where they can be found by the MPI routines (in the request, probably).

10.3 Uncompleted operations

In the case of the get and rendevous protocols, when a process enters `MPI_Finalize`, it may still need to complete a get or rendevous operation. This needs to be provided on a device by device basis.

11 A Simple Test Program

This section contains a simple test program for the channel operations that can be used to test their operation. It also show exactly how they are used. (This section is NOT YET CORRECT!)

This test program does NOT test that the `MPID_SendControl` is non-blocking. It does test that messages can be sent between two processes, and that large numbers can be sent. This helps test for possible resource exhaustion in the implementation of the routines.

```
#include "mpid.h"
#include <stdio.h>
/* Set to stdout to get trace */
static FILE *MPID_TRACE_FILE = 0;

int main(argc,argv)
int argc;
char **argv;
{
MPID_PKT_SHORT_T pkt;
int         from, ntest, i;

ntest = 10000;
MPID_Devinit( argc, argv );
if (MPID_WorldSize != 2) {
    fprintf( stderr, "\n", MPID_WorldSize );
    }
for (i=0; i<ntest; i++) {
    if (MPID_MyWorldRank == 0) {
MPID_SendControl( &pkt, sizeof(MPID_PKT_SHORT_T), 1 );
from = -1;
MPID_RecvAnyControl( &pkt, sizeof(MPID_PKT_SHORT_T), &from );
if (from != 1) {
    fprintf( stderr,
"0 received message from %d, expected 1\n", from );
    }
}
    else {
from = -1;
MPID_RecvAnyControl( &pkt, sizeof(MPID_PKT_SHORT_T), &from );
if (from != 0) {
    fprintf( stderr,
"1 received message from %d, expected 0\n", from );
    }
MPID_SendControl( &pkt, sizeof(MPID_PKT_SHORT_T), 0 );
}
    }
MPID_Devend();
return 0;
}
```

12 Future developements

The definition of this interface is changing slowly as more devices are added and as we continue to work for better performance. The areas that are likely to change the most include (a) collective operations, (b) support for faster `MPI_Bsend`, (c) enhanced support for systems with limited or inadequate buffering, and

(d) special support for systems that can not send messages to themselves. We solicit comments for these and other changes.

13 Acknowledgements

The design of this interface has benefitted from discussions with many people, including Jim Cownie of Meiko, Lloyd Lewins of Hughes, Eric Salo and Greg Chesson(??) of SGI, and Bjarne Herland of University of Bergen, Norway.