ANL/MCS-TM-number

MPICH Working Note: The implementation of the second generation MPICH ADI

by

William Gropp
Ewing Lusk

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Abstract

The MPICH implementation of the MPI standard is built on a lower level communications layer called the *abstract device interface*. The purpose of this interface is to make it easy to port the MPICH implementation by separating into a separate module that handles just the communication between two processes. This note describes an implementation for this interface that is flexible and efficient. In addition, this implementation supports multiple devices (e.g., TCP/IP and shared memory or TCP/IP and propritary interconnect) in a single MPI application.

1 Introduction

The goals are to allow for multi-device, multi-message-protocol implementations of the Abstract Device without (significantly) impacting performance.

Remember to differentiate between the ADI *design* and any one of its *implementations*.

Outline of paper...

2 Protocols

The MPI standard is designed to give an MPI implementation freedom in chosing exactly how and when data is moved from one process to another.

In this implementation, the information is divided into two parts: the message *envelope*, containing information about the message (such as tag, communicator, and length), and the *data*. The envelope is relatively small, and is delivered to the destination immediately.[1] The data may or may not be delivered at once. In fact, there are four possibilities that we will allow for in our implemenation:

Short The data is delivered within the message envelope.

Eager The data is delivered without waiting for the receiver to request it

Rendezvous The data is not delivered until the receiver requests it

Get The data is copied by the receiver using shared memory or remote memory operations.

There are other approaches possible, but these four *protocols* are the only ones considered in this ADI implementation.

Within the eager and rendezvous protocols, the method by which the data is delivered once requested may be *blocking* or *nonblocking*. In a nonblocking transfer, once the data is requested, the sender can call a system service routine to begin the transfer and then return to the user without waiting for the transfer to complete (until a later *wait* call). This is similar to the MPI nonblocking operations (`MPI_Isend()`...`MPI_Wait()`).

No single choice of protocol is optimal. In many applications, a combination of these protocols is appropriate. For example, for messages of intermediate length, the eager protocol may offer the best combination of performance and reliability, while for very long data, only the rendezvous (or get) protocols maintain correctness. The ADI implementation design here supports using multiple protocols.

To simplify the implementation, only contiguous data areas are handled. The more general MPI datatypes are handled by packing and unpacking data into contiguous buffers, as described in [?].

3 Devices

stuff about supporting multiple devices

Most obvious combinations: TCP plus propritary. Also TCP + shared memory. Others include TCP + ATM + Shared memory + propritary.

Note that in some cases involving shared memory and other protocols, it may be better to use an agent to move messages.

[1] See the comments on resource restrictions later in this paper.

4 Design

We describe the design by describing the steps that the implementation carries out to perform the basic operations of sending and receiving a message.

4.1 Sending

First, we determine the device to use, based on the destination (if there are multiple devices, this can be an array of pointers to structures, indexed by global rank).

Next, we determine which protocol to use (short, eager, rendezvous, get). This is probably based on the size of the message, but could be based on something else, like the number of pending completions, datatype, etc.

We then start the send with the appropriate operation, chosen from the protocol.

Now, if the operation is *not* already complete (e.g., nonblocking or rendezvous or get send), we need to know how to

- test for completion
- wait for completion
- cancel
- push

The polling design of the MPICH implementation puts the "push" partly into the test call, so that the MPICH usage is really

- test for completion
- push-and-then-test for completion

There is also a common push-device, but it doesn't keep track of each individual item.

The most general way to handle this case is to store the *functions* in the request. An alternate method, in use in the first generation system, is to store an integer that selects the function. This limits the choices and makes it hard to add new approaches (see below).

Note that we can combine the test and push-test with the choice that a null push-test function means that the request is complete.

Finally, for reasons the will be clear when discussing the noncontiguous operations, it is helpful to have a routine to call to finish the request locally. This can be used, for example, to free any local buffers. The name of this function is `finish`.

4.2 Receiving

This is a little harder, because to start with we don't know even know the device that a message is ariving on. There are two obvious ways to approach this: either determine which device a message is available on and then receive that message, or for each device, try to receive a message (without blocking when no message is available). So we start could start with commands to check to see if a message is available, or to wait for one. But this leaves us with the problem of what to do with the message if checking for the message might receive the envelope. So, instead of checking for a whether a message is available, it is better to try and receive a message with

- check-device

which has blocking and non-blocking versions *for each device*.

The multi-device version just calls the `check_device` for each of the devices. A `wait` may then turn into a poll. Note that the multi-device code could optimize for the case of a single *active* device, and use a wait when a blocking check-device is needed.

This requires each device to provide its own `check_device`; this really isn't so bad as the message-channel (eager/rendezvous) versions really need different code from the shared-memory (get) versions.

Note that this solves a number of problems:

1. It doesn't matter whether to low-level code wants to receive a *pointer* to a control packet or the actual packet.
2. The packets are likely different in detail, though we can probably restrict a few fields. Note that for shared memory, we usually want the address of the data, whereas for "stream/channel" interfaces, we receive the data into local data
3. Heterogenity information will be needed by only some of the packets.
4. Note that we may receive forwarding packets that are intended to send data to another node. This is just a data packet, so again the actual form is unimportant.

The check device code itself could use either a lookup table based on the packet type field (small integer) or a `switch` statement. In the lookup table case, this selects the routine that matches the kind of send that was used. The table is initialized when the device starts up. Using a `switch` is simpler but a little less flexible. Still, since the `check_device` code itself is accessed through a function pointer, we could eventually add a more complex version.

Now the situation is much like the send case. If the operation is not complete, we set up pointers to functions to

- test for completion
- wait for completion
- cancel
- push

Note that the `push` operation can change the others as a way to implement a finite-state-machine approach to message delivery.

Now, for receives, there are two situations: the message is expected, and the message is unexpected. The above handles the case of an expected message. In the unexpected case, we also need another action:

- accept unexpected message

This can start the process of receiving the data for an unexpected message that is now expected. One question is whether this should be a separate function or just another case of `push`. We have chosen to use the `push` function in order to reduce the number of function pointers in the request, and because it is a push operation. Note that the `push` for an unexpected message does not need to complete immediately; it can start the receive and return, setting the `test`, `wait` etc. functions appropriately.

4.3 Synchronous sends

Rather than support a completely separate operation, this will just use a transfer with a rendezvous semantic (rendezvous or get). In order to ensure that `MPI_Ssend( .., cnt = 0, ..  )` works, we must be careful about handling 0-length messages.

The rest of this applies to the the implementation of the rendezvous protocol.

Details: For heterogenity, there needs to be a agreed form of `send_id` that can be mapped back into the sender's request. An easy solution would be to use 8 bytes for the id, and transmit it without modification. Another is to use 4 bytes, and to convert longer address as in the Fortran version.

This is really a device choice, and we can limit the 8-byte version to the heterogeneous systems.

Details: A special form of the eager message, the "within the packet" form, is also used. There needs to be a `get` and `channel` version of this.

4.4 Freeing requests

The MPI standard allows

```
MPI_Isend( ..., &request )
MPI_Request_free( &request )
```

and

```
MPI_Irecv( ..., &request )
MPI_Request_free( &request )
```

In order for us to properly complete this, we need to continue to *push* the requests until they complete. For this to work, we need to put the request on a global list of these requests, and run through this list frequently. At the very least, the *device push* should process this list. For this, we need a list *next* pointer in the request.

4.5 Single device optimizations

Much of the function pointer interface can be removed in the single device case:

- No lookup on send or on receive
- Switch statements can replace function pointers for test/wait/push

The first of these is fairly easy and may help on fast systems. The second may not be worth the trouble, since if there is a function to call, the overhead is already relatively high.

4.6 Requests

How are requests represented? At the MPI level, they are opaque (pointers in the MPICH implementation). But eventually, the request needs to be defined as a structure containing some data. There is clearly data common to the MPI implementation, but each device may have its own needs for special data.

Should we define a common form? This is pretty much true now, with the possible exception of the "nonblocking id", which may be an int (for nx), a small structure (IBM MPL or aioread/aiowrite), or something else. The `send_id` may be 4 or 8 bytes, depending on the system.

One approach is to provide the basics in the structure (and make them part of the MPIR handle rather than MPID handle) and use a pointer for any other data. This is the most flexible but potentially introduces an additional memory allocation. Again, the single device and multi-device could use slightly different forms, though I don't want to maintain that.

What to do?

Choose common form with 4 int area for any special device needs. Device can put a pointer to more data there if needed.

Rather than have separate device and MPIR parts of the request, there is just a single request type for each operation. That is, there is an `MPIR_RHANDLE` for receive requests, an `MPIR_SHANDLE` for send requests, and similar structures for persistent and user operations. An `MPI_Request` is simply a union of these types.

5 Implementation sketch

There is an MPI to channel layer that selects the protocol based on message size. This could even be a macro, using a global (or device specific?) value.

The `check_incoming` routine remains about the same, with unexpected message processing doing this:

```
save information on the incoming message (tag, lrank, len, context)
based on device/packet type, save additional information
    (e.g., data for eager send, send_id for get/rendezvous.  Careful
     of rendezvous for Ssend).
```

So, a send looks something like this

```
send( ..., dest, ... )

    { MPID_Device *dev = devset->dev[dest];
      if (len < dev->long_len)
        (*(dev->short->send))( ... );
      else if (len < dev->vlong_len)
        (*(dev->long->send))( ... );
      else
        (*(dev->vlong->send))( ... );
     }
```

The single device case can use just the if-else list (dev is fixed, use `MPID_dev`), and the fixed protocol case can use the explicit routines. The shared-memory version will use a slightly different version.

A receive is more complex, since MPI guarentees that a blocking receive on one communicator does not affect any other communication. So a multi-device blocking receive looks something like

```
     Is message already available?
     Yes ->
        Call recv_unexpected code (see below)
     No ->
        (test has already added message to list of posted receives (for
         thread safety))
     (a nonblocking receive exits here)
     while (message not received)
        if (req->push) (req->push)( req )
        for each device
           check_device( (ndev > 1) ? non-blocking : blocking )
```

This assumes that, at least in the single device case, you can block in check device to await a message. If this is *not* true, then the `check_device` routine should return.

In the case of finding an unexpected message,

```
recv_unexpected
    Save matched data (tag, source)
    Check for message data available (short or eager delivery)
    Save functions (test,wait,push)
    if (req->push) (req->push)( req )
```

Note that this does *not* necessarily complete the message; this allows non-blocking receives that match an unexpected message to act as non-blocking operations.

Remaining concerns:

If the lowest levels send a message with a non-blocking operation, is it ever safe to `wait` on it? Or do we have to spin in a test loop? Should the wait entry be defined ONLY when it can be safely called? For example, in a rendezvous system, the non-blocking operation will be started only in response to a direct request; could you wait then? Do we leave this to the device? Implementation of push?

MPI-2 dynamic process management.

The biggest issue here is the support of dynamically resized “global” lists. Mostly, the implementation needs to remember that “grank” is not in `MPI_COMM_WORLD` but is the result of the `lrank_to_grank` mapping in the communicator.

6 Noncontiguous and Heterogeneous

The simplest implementation for the noncontiguous and the heterogeneous cases is to simply to copy the message into and out of a temporary buffer. A refinment of this delays the allocation of the buffer until it is needed. For example, in a rendezvous send, the allocation of the buffer need not happen until the "ok to send" message is received. This is easily handled by modifying the routines in the send request that are called when the "ok to send" is received. The same can be done on the receive side.

To simplify the handling of heterogeneous systems, the implementation includes a `msgrep` field, which is a (very small) integer that may be sent between heterogeneous processors. Completely homogeneous systems can ignore this (through the use of macros, they could even eliminate the argument).

The `finish` function may be used to handle any local operations once the message is sent/received. For example, in a simple implementation of heterogeneous/non-contiguous nonblocking receive, a contiguous buffer is allocated and passed to `MPID_IrecvContig` to start the receive; once the receive completes, the `finish` function can be called to unpack the data and to free the temporary buffer.

7 Message queue management

A number of

The major interfaces used are

MPID_Search_unexpected_queue Look for a message that matches the `(tag,context_id,source)` in the unexpected queue. This is used for probe operations.

MPID_Search_unexpected_queue_and_post If not found in the unexpected queue, add to the posted receive queue

MPID_Msg_arrived Check the posted receive queue; if not found, add to unexpected queue

These interfaces have been designed to simplify a thread-safe implementation. For example, the `MPID_Search_unexpected_queue_and_post` needs to be an atomic operation; if it is not and is implemented as

```
MPID_Search_unexpected_queue
if (not found)
   MPID_Enqueue( posted_receive_queue ... )
```

then if another thread delivers the message after the unexpected queue is checked but before the `MPID_Enqueue` is called, the message will be lost (inserted by the second thread into the unexpected queue). The `MPID_Msg_arrived` is essentially the reverse of `MPID_Search_unexpected_queue_and_post`.

8 Miscelleous Routines

common argument processing

A Rationale for three modes

In this section we present a performance-based rationale for having three modes (short, eager, and rendezvous) in a message-passing send.

For this analysis, we will assume that the cost of sending a single message of n bytes is $s + rn$. We further assume that the size of the "control" packet (or header) is a fixed h bytes, and that the cost to copy a byte within a process is c. Finally, let f be the fraction of messages that arrive before the matching receive is posted. Then the cost of the three protocols can be summarized as

short	eager	rendezvous
s+r(n+h)+(1+f)cn	2s+r(n+h) + fcn	3s+r(n+2h)

The various cn terms represent the cost of moving the data within the receiving process. For example, in the short protocol case, the data is received in the control packet and must be copied into the user's buffer. If the message was unexpected, it needs to be copied twice.

With the values here, we can compute the message size where each protocol should be used. For example, the switch between the short and eager protocol should happen when

$$\begin{aligned}
s + r(n+h) + (1+f)cn &= 2s + r(n+h) + fcn && (1)\\
(1+f)cn &= s + fcn && (2)\\
cn &= s && (3)\\
n &= s/c && (4)
\end{aligned}$$

In other words, when the code of the extra data copy equals the latency to send a separate message, switch to the eager protocol.

We can do the same thing for eager to rendezvous.

$$\begin{aligned}
2s + r(n+h) + fcn &= 3s + r(n+2h) && (5)\\
fcn &= s + rh && (6)\\
n &= \frac{s+rh}{fc} && (7)
\end{aligned}$$

Again, when the cost of sending an additional control message $(s + rh)$ is the same as the cost of copying the data (fcn), switch to the rendezvous protocol. We can approximate this in the case that $s \gg rh$ to just

$$n \approx s/fc$$

which is just $(1/f)$ times the size for the switch from short to eager.

An important effect not included here is the cost in a rendezvous send of the receiver waiting for the sender to respond to the "go ahead" message. The effect of this is to move the point where the rendezvous protocol should be used *for performance reasons* to larger message sizes. For example, if there might be an average delay d, the formula for switching between eager and rendezvous becomes

$$n \approx \frac{s+d}{fc}.$$

if $d \gg s$ (for example, a timer interrupt every few hundred milliseconds), then n can be very large; larger perhaps than any message that is likely to be sent.

The analysis could be inverted to solve for d instead of n, if the largest allowed n for the eager protocol (because of system buffer limitations, for example) is already known. In this case, if the time d does represent an occasional timer interrupt, then the cost of this interrupt itself (particularly when it discovers nothing to do) needs to be included as well. Such implementation-specific analysis are left to the reader.