ANL/MCS-TM-number

MPICH Working Note: The Second-Generation ADI for the MPICH Implementation of MPI

by

William Gropp
Ewing Lusk

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Abstract

In this paper we describe an abstract device interface (ADI) that may be used to efficiently implement the Message Passing Interface (MPI). After experience with a first-generation ADI that made certain assumptions about the devices and tradeoffs in the design, it has become clear that, particularly on systems with low-latency communication, the first-generation ADI design imposes too much additional latency. In addition, the first-generation design is awkward for heterogeneous systems, complex for noncontiguous messaging, and inadequate at error handling. The design in this note describes a new ADI that provides lower latency in common cases and is still easy to implement, while retaining many opportunities for customization to any advanced capabilities that the underlying hardware may support.

1 Introduction

Our goal is to define an abstract device interface (ADI) on which MPI [1] can be efficiently implemented. This discussion is in terms of the MPICH implementation, a freely available, widely used version of MPI. The ADI provides basic, point-to-point message-passing services for the MPICH implementation of MPI. The MPICH code handles the rest of the MPI standard, including the management of datatypes, and communicators, and the implementation of collective operations with point-to-point operations. Because many of the issues involve understanding details of the MPI specification (particularly the obscure corners that users can ignore but that an implemenation must handle), familarity with Version 1.1 of the MPI standard is assumed in this paper [2].

This note also discusses changes to the MPICH implementation to improve performance, particularly with respect to the organization of `MPI_Request`. To fully understand these comments, some familiarity with the MPICH implementation of MPI is required.

To make the discussion more concrete, we sometimes use "ADI" to refer to a particular implementation of the ADI. However, the ADI is defined by the specification of the interface, not by any particular implementation.

The rest of this paper is organized as follows. Section 2 describes the guiding design principles and objectives. Section 3 describes the routines for moving contiguous data. Section 4 discusses the support for the noncontiguous and heterogeneous data types. Section 5 discusses the other, supporting functions provided by the ADI. Section 6 discusses some of the unusual cases that the MPI standard requires be handled. Finally, Section 7 shows some simple message-passing programs that use only the ADI functions defined here.

2 Principles

The design tries to adhere to the following principles:

- No extra memory motion, even of single words. A single cache miss can generate enormous delays (on the order of one microsecond in some systesms).
- Minimal number of procedure calls.
- Efficient support of `MPI_Bsend`, particularly for short messages.
- The common contiguous case should be simple and fast.
- The general data structure case should be managed by the ADI, even if it involves simply moving data into and out of contiguous data buffers (putting those buffers under control of the ADI).
- The ADI should be usable by itself (without the entire MPI implementation), simplifying testing and tuning of the ADI.

Most of these are clearly desirable. One descision that is not obvious is the support for general datatypes. For reasons of maximum efficiency, particularly for regular but not contiguous datatypes (e.g., strided vectors), it is important that the ADI have the opportunity to handle these types. On the other hand, particularly when porting to a new platform, requiring that the ADI handle the general MPI datatypes imposes a substantial burden on the implementor. The design in this document shows how the general datatypes can be supported using only routines that move contiguous arrays of bytes, and the model implementation contains this support.

This design seems to require more functions (and hence more code) than the previous ADI design but really does not. It simply separates out the individual cases that need to be implemented.

The routines described in this document are expected to be implemented as macros, allowing the use of multiple instances of the routines for implementations that support multiple ADIs. In addition, most of the routines contain an MPI communicator argument that is used as a container for any special information; this may be omitted by particular implementations of the ADI to shorten the argument lists.

One objective that is hard to achieve is to limit the number of redundant memory operations. This particularly impacts error checking; if all of the error checks are made in one place (say, at the top of a routine like `MPI_Send`), and the same data is used later, possibly in a different routine, the data will be loaded twice. Because the ADI interface is not intended for general use, we place the error checking nearest to the first use of the data. For items like the message tag and buffer, this testing will be in the ADI routines. For items like the datatype and communicator, it may be most natural to test these before calling the ADI routines. For example, the ADI has separate routines for contiguous and noncontiguous data, requiring that the datatype be tested first.

Note that some errors can be detected most efficiently by the ADI routines (for example, in routines that take a datatype and buffer, a null or invalid buffer is detected most easily as the data is being moved, since an MPI datatype may specify an absolute address offset for a buffer, allowing a null buffer as an argument). These routines should also detect message truncation on a receive, and handle any resource limitations. Thus, there is no uniformly convenient place to do all error checking.

In implementing this ADI, there are some operations that will be common to many different implementations. For example, many implementations will need routines to manipulate message queues and the pack and unpack non-contiguous data. The model implementation will provide these.

One final complication comes from the requirement in MPI 1.1 that MPI objects like `MPI_COMM_WORLD` be compile-time constants. This is much stronger than the MPI 1.0 requirement that they be unchanging between `MPI_INIT` and `MPI_FINALIZE`. This essentially requires that the values for these objects be predefined integers, particularly when considering the Fortran versions (which require `PARAMETER` values). As a result, the types of the user-visiable objects like `MPI_Comm`, `MPI_Datatype`, etc. are ints. These ints are translated into pointers to the appropriate structures; the ADI never sees the original int values. Thus, the device interface is given `struct MPIR_DATATYPE *` instead of `MPI_Datatype` and `struct MPIR_COMMUNICATOR *` instead of `MPI_Comm`. Both of these structures contain a `self` field that provides the integer value that corresponds to the pointer. It is the responsibility of the code that calls the ADI routines to ensure that the pointers are valid.

The rest of this papers is organized as follows. Section 3 describes the routines for sending and receiving contiguous data. Section 4 describes the routines for non-contiguous data, and describes how they can be implemented by using the contiguous routines. Section 5 describes various service and informational routines. Section 6 describes special considerations and issues in implementing the ADI. Finally, Section 7 presents some example message-passing programs that use only the ADI routines.

3 The Contiguous Routines

The contiguous user-buffer case is an important special case, and one that is handled by all hardware. This design provides special entry points for operations on contiguous messages.

3.1 Sending

The basic contiguous send routine has the form

```
MPID_SendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, &error_code )
```

comm Communicator for the operation (`struct MPIR_COMMUNICATOR *`, not `MPI_Comm`).

buf Address of the buffer.

len Length in bytes of the buffer.

src_lrank Rank in communicator of the sender (source local rank).

tag Message tag.

context_id Numeric context id.

dest_grank Rank of the destination, as known to the device. Usually this is the rank in `MPI_COMM_WORLD` but may differ if dynamic process management is supported.

msgrep The message representation form. This is an `MPID_Msgrep_t` (enum int) data item that will be delivered to the receiving process. Implementations are free to ignore this value.

error_code The result of the action (an MPI error code). This may be set only if the value would not be `MPI_SUCCESS`. In other words, an implementation is free to not set this unless there is an error. The error code was made an argument so that an implementation as a macro would be easier.

Similar versions exist for the other MPI send modes (i.e., `MPID_SsendContig` and `MPID_RsendContig`).

One point that deserves discussion is the use of a global rank for the destination instead of a rank relative to the communicator. The advantage of using a global rank is that many ADI implementations can then ignore the communicator argument (except for error processing). In addition, for communicators congruent to `MPI_COMM_WORLD`, no translation from relative to global ranks is needed.

3.1.1 Buffered Sends

MPI programs can suffer in performance if a naive implemenations of `MPI_Bsend` is used. The standard shows an implementation that uses nonblocking sends and extra copies for each message. Many systems can deliver short messages without any special effort; the routine `MPID_BsendContig` does so if possible. However, rather than block as `MPID_SendContig` does if the message cannot be delivered, `MPID_BsendContig` returns `MPIR_ERR_MAY_BLOCK` as the error code. In this case, the MPI implementation (not the ADI) can use the model solution in the standard.

3.1.2 Receiving

The routine to receive a message is much like the send routines, with the addition of an `MPI_Status` argument:

```
MPID_RecvContig( comm, buf, maxlen, src_lrank, tag, context_id,
                 &status, &error_code )
```

The address of the `status` may be null, in which case all status information is discarded. This is not supported by the MPI standard, but is not prohibited either, and this allows study of what efficiencies can be gained.

Note that the receive is given the `src_lrank`, the local rank of the source in the communicator. If the implementation of `MPID_RecvContig` requires the absolute rank, it can get this from the communicator with the code (this is MPICH-specific)

```
comm->lrank_to_grank[src_lrank]
```

Note that `MPID_RecvContig` has both a `status` and an `error_code` argument. An MPI status has the field `MPI_ERROR`;[1] this could be used instead. Doing so, however, would eliminate the possibility of using a null `status`. Moreover, in the case of no error, there is no additional cost.

[1] Added in version 1.1 of the MPI standard.

3.2 Summary of Blocking Routines

These routines correspond to the MPI routines `MPI_Send`, `MPI_Bsend`, `MPI_Rsend`, `MPI_Ssend`, and `MPI_Recv` respectively.

```
MPID_SendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, &error_code )
MPID_BsendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, &error_code )
MPID_RsendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, &error_code )
MPID_SsendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, &error_code )
MPID_RecvContig( comm, buf, maxlen, src_lrank, tag, context_id,
                 &status, &error_code )
```

3.3 Nonblocking Routines

Just as with MPI, the nonblocking versions are critical and can be used to implement the blocking versions. The nonblocking routines are much like the blocking versions, except that they take the `MPI_Request` handle for the operation (not just a device-specific handle, as in the first-generation ADI). However, the `MPID` routines do not allocate the `MPI_Request` as the `MPI` routines do; rather, an MPI routine allocates the request and passes it to the `MPID` routine. The semantics of the communication are the same as for their MPI counterparts.

There is no ADI support for persistent requests; this feature is discussed further in Section 6.1.

3.4 Summary of Nonblocking Routines

There is no nonblocking buffered send; that function can be implemented with the routines already defined, probably with little additional overhead.

```
MPID_IsendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, request, &error_code )
MPID_IrsendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgrep, request, &error_code )
MPID_IssendContig( comm, buf, len, src_lrank, tag, context_id, dest_grank,
                 msgpre, request, &error_code )
MPID_IrecvContig( comm, buf, maxlen, src_lrank, tag, context_id,
                 request, &error_code )
```

3.5 Message Completion

Nonblocking messages need to be completed eventually; at the MPI level, this is done with the `MPI_Wait/MPI_Test` family of routines. The corresponding `MPID` routines are a subset of these and have two forms: a quick test for completion (indicating that no further action is required) and a test-and-complete (similar to `MPI_Test`). There are separate routines for send and receive operations.

In addition, there is a "wait-until-something-happens" routine called `MPID_DeviceCheck`. This can be null but can be used to allow `MPI_Wait` to wait more passively (for example, with a `select`) rather than spinning on tests. It takes one argument of type `MPID_BLOCKING_TYPE`; it can be used for both blocking and nonblocking checks of the device.

The simplest case is testing for requests that have already completed. This is done by testing the field `is_complete` in the request structure. The ADI should set this field in the request when the operation is complete. [2]

[2] Early versions of this document included a `MPID_RecvDone` and a `MPID_SendDone` call, but in the end is was clearer to make this an explicit part of the `MPI_Request` structure.

If a request is not complete, the routines MPID_xxxxIcomplete can be used. These return true if the request is complete (after calling); that is, they return true if the corresponding MPID_xxxxDone routine would return true. These routines are expected to call the device and perform some additional processing. Note that MPID_RecvIcomplete sets the status variable; just as for a blocking receive, status may be null.

```
MPID_RecvIcomplete( handle, &status, &error_code )
MPID_SendIcomplete( handle, &error_code )
```

If an error has been detected, these routines return true and set the error_code appropriately. For a send operation (MPID_SendIcomplete), an error is unlikely but could occur, for example, when the destination process disappears. The use of I in Icomplete indicates that these routines do not block waiting for the operation to complete. These really correspond to MPI_Test, which, despite its name, does more than just test (since, on a successful test, it also completes the request).

The routine MPID_DeviceCheck may be used to wait for something to happen. This routine may return at any time; it *must* return if the is_complete value for any request is changed during the call. For example, if a request would be completed only when an acknowledgment message or data message arrived, then MPID_DeviceCheck could block until a message (from anywhere) arrived. This call is typically used in the implementation of MPI_Waitxxx, as shown below. Note that calling MPID_DeviceCheck itself is *not* enough to ensure that a request is completed.

There could be separate send versions for each type of send (ready, synchronous, etc.); but since the ADI may map the different send operations into such parts as "short eager" or "long rendezvous" messages, it makes more sense for the ADI to take the responsibility of remembering what kind of ADI operation the request is performing. In the sample implementation, this information is stored in the MPI request in the completer field (see Section 6.1).

An illustrative use of these calls in the implementation of MPI_Wait is shown below.

```
...
int err = MPI_SUCCESS;
switch ((*request)->type) {
    case MPIR_SEND:
    if ((*request)->shandle.is_complete) {
        do {
            MPID_DeviceCheck( MPID_BLOCKING );
            } while (!MPID_SendIcomplete( *request, &err ));
        }
    MPID_SendFree( request );
    request = 0;
    return err;
    break;
...
```

In the case of MPID_Waitall, the code is a little different because of the need to wait for completion in any order; this difference explains the need for MPID_DeviceCheck rather than just a MPID_SendComplete routine:

```
...
int err = MPI_SUCCESS;
nleft = n;
while (nleft) {
    for (i=0; i<n; i++) {
    request = array_of_requests[i];
    switch ((request)->type) {
      case MPIR_SEND:
      if (request->shandle.is_complete) {
        MPID_SendFree( request );
        array_of_requests[i] = 0;
```

```
        if (nleft-- == 0) return err;
        }
      else {
        if (MPID_SendIcomplete( request, &err )) {
            MPID_SendFree( request );
            array_of_requests[i] = 0;
            if (nleft-- == 0) return err;
            }
        }
      break;
      case MPIR_RECV:
      ...
      }
    MPID_DeviceCheck( MPID_BLOCKING );
    }
```

This code is a little sketchy (particularly on the error handling) and is not appropriate for large `n`, but it does show the use of `MPID_DeviceCheck` to allow the operations in the `MPI_Waitall` to complete in any order without requiring the code to constantly call nonblocking test routines.

The implementation of `MPI_Wait` shown above suggests that there be the additional functions

```
MPID_SendComplete( handle, &error_code )
MPID_RecvComplete( handle, &status, &error_code )
```

which *block* until the specified operation completes. These can be implemented with `MPID_DeviceCheck` and `MPID_xxxIcomplete`, but some devices may be more efficient with these calls (for example, in the MPICH implementation, the Intel `nx` and Meiko `meiko` devices would implement these directly). The MPICH implementation of `MPI_Waitall` etc. makes use of these.

3.6 Freeing requests

The MPI standard allows requests to be "freed" by the user at any time with the `MPI_Request_free` call. This means that the user will never issue a wait or test call with this request. However, the MPI implementation must still complete the operation and recover the space used by the request itself (the user is responsible for not using the data buffers improperly). Note that `MPI_Request_free` is *not* like a `MPI_Wait` call; it is a local call. To inform the ADI that the request is being released, the additional functions

```
MPID_SendFree( handle )
MPID_RecvFree( handle )
```

are defined. Once a handle has been given as an argument to one of these routines, it is the responsibility of the device to free the request structure.

3.7 Shared Data Structures

The nonblocking operations must share a data structure with the MPI implementation; this is the part of the `MPI_Request` that the device needs. Since the part of the implementation above the ADI that uses the requests needs to access very few of the fields in the request, the ADI defines the requests as structures of type `MPIR_RHANDLE` (receive), `MPIR_SHANDLE` (send), `MPIR_PRHANDLE` (persistent receive), and `MPIR_PSHANDLE` (persistent send). The file '`mpid/ch2/req.h`' shows one possible definition. An ADI may define its own request structures as long as the shared fields are maintained. These shared fields include

MPIR_OPTYPE handle_type Type of handle.

int is_complete Is the request complete?

MPI_Status s Status for request on completion (receive and persistent receive only).

int errval Error code (send and persistent send only).

There is also an `MPIR_COOKIE` definition that is used to check for valid requests; this is defined in '`mpid/ch2/cookie.h`'.

Unlike the first generation ADI, this ADI is responsible for more of the "bookeeping" details; this requires that the ADI know more about MPI objects. For example, the following code is legal:

```
MPI_Type_struct( ..., &newtype );
MPI_Type_commit( &newtype );
MPI_Send_init( ..., newtype, ..., &request );
MPI_Type_free( &newtype );
MPI_Start( &request );
....
```

Note the the user-defined datatype `newtype` was freed even before the communication was started. The MPICH implementation maintains reference counts on all objects that may be used in this way (including datatypes, communicators, groups, and error handlers); the ADI must update these reference counts as needed, and must be prepared to delete an object if it decrements a reference count. This is most likely to be encounted when implementing receives where the message is not yet available.

4 Noncontiguous Operations

Noncontiguous operations are much like their contiguous counterparts, except that the `buf, len` pair is replaced by `buf, count, datatype`. The `datatype` is an `struct MPIR_DATATYPE *`, and the `count` (number of elements) replaces the `len` (number of bytes). The other arguments are unchanged. The routine names replace `Contig` with `Datatype`; thus, the general version of `MPID_IrsendContig` is `MPID_IrsendDatatype`.

In a heterogeneous system, *all* communication takes place through the noncontiguous routines (because these are the only ones with the `struct MPIR_DATATYPE *` needed for converting data formats). A typical use of these routines in the MPI implementation is something like

```
int MPI_Send( ... )
{
...
#if !defined(MPID_HAS_HETEROGENEOUS)
if (datatype->is_contig) {
    MPID_SendContig( ... &error_code );
    } else
#endif
    {
    MPID_SendDatatype( ... &error_code );
    }
return error_code;
}
```

4.1 Pack and Unpack

MPI contains routines to pack and unpack data. These routines are `MPI_Pack`, `MPI_Unpack`, and `MPI_Pack_size`. An MPI implementation must provide these; further, a user may send data that has be constructed with `MPI_Pack` with datatype `MPI_PACKED` and receive it either with datatype `MPI_PACKED` *or* with any MPI datatype with the same type signature that went into the packed data. Because of this, the device must provide the routines to pack and unpack data. Of course, many implementations of the device may use the model implementation's version of these routines.

The MPI pack and unpack routines are designed to handle data on a communicator-wide basis. That is, data is packed relative to a communicator; a natural implementation is to pick a data representation that is a good choice for all members of the communicator (including the sender!). However, a common use of these routines in an implementation is to pack and unpack data sent with the point-to-point operations. In order to provide for this use, the ADI's version of the pack and unpack routines include a `partner` field, which is either a rank in the communicator of the partner (sender or receiver) or `MPI_ANY_SOURCE`, which indicates "any member of the communicator" and most closely matches the MPI `MPI_Pack` and `MPI_Unpack` routines.

Note that just as send and receive seem very symmetric but are not, so are pack and unpack in fact unsymmetric. On the pack (sender) side, the sender has some freedom in choosing just how to represent the data. On the receiver's side, the data must be dealt with as it arrives. Thus, on the sending side, the sender picks the message representation (`msgrep`, a value passed to the receiver) and the routine to set the data up in this way (the message action `msgact` in the routines below). On the receiving side, only the `msgrep` is available to the receiver; from this it must determine how to unpack the data.

On the senders side, the routine `MPID_Msg_rep` determines the message representation and action from the communicator, destination, and datatype. Note for example that if the datatype is `MPI_PACKED`, the `msgact` will be `MPID_MSG_OK` (just copy) while the `msgrep` will be whatever is used for packed data (e.g., `MPID_MSGFORM_XDR`).

```
MPID_Msg_rep( comm, partner, datatype, &msgrep, &msgact )
```

comm Communicator for the operation.

partner Rank of "partner" process or `MPI_ANY_SOURCE`.

datatype `struct MPIR_DATATYPE *` of input buffer. Allows the detection of the special case of `MPI_PACKED`.

msgrep Message representation; this is an output value.

msgact Action for message representation; this output value may be used as input to the pack/unpack routines to indicate how to process the message. This is a `MPID_Msg_pack_t` type (enum int).

On the receiver's side, the message action can be determined from the `msgrep` and the `datatype`. Just as for sending, if the datatype is `MPI_PACKED`, then the `msgact` may be just copy.

```
MPID_Msg_act( comm, partner, datatype, msgrep, &msgact )
```

comm Communicator for the operation.

partner Rank of "partner" process or `MPI_ANY_SOURCE`.

datatype MPI datatype of input buffer. Allows the detection of the special case of `MPI_PACKED`.

msgrep Message representation; this is an input value (from the message envelope).

msgact Action for message representation; this output value may be used as input to the unpack routines to indicate how to process the message.

```
MPID_Pack( src, count, datatype, dest, maxcount, &position,
           comm, partner, msgrep, msgact, &error_code )
```

src Address of the input buffer.

count Number of item in input buffer.

datatype MPI datatype of input buffer.

dest Address of the output buffer.

maxcount Size of dest in bytes.

position Current location in `dest`. Set to 0 on first call; updated by routine (allows multiple calls to `MPID_Pack`).

comm Communicator for the operation.

partner Rank of "partner" process or `MPI_ANY_SOURCE`.

msgrep Message representation type.

msgact Action for message representation.

error_code The result of the action (an MPI error code). This may be set only if the value would not be `MPI_SUCCESS`. In other words, an implementation is free to not set this unless there is an error. The error code was made an argument so that an implementation as a macro would be easier.

```
MPID_Unpack( src, maxcount, msgrep, &in_position,
             dest, count, datatype, &out_position,
             comm, partner, &error_code )
```

src Address of the input buffer.

maxcount Size of `src` in bytes.

msgrep Message representation.

in_position Current Location in `src`. Set to 0 on first call; updated by routine.

dest Address of the output buffer.

count Maximum number of `datatype` items to unpack.

datatype MPI datatype of input buffer.

out_position Current location in `dest`. Set to 0 on first call.

comm Communicator for the operation.

partner Rank of "partner" process or `MPI_ANY_SOURCE`. routine for subsequent calls

error_code The result of the action (an MPI error code). This may be set only if the value would not be `MPI_SUCCESS`. In other words, an implementation is free to not set this unless there is an error. The error code was made an argument so that an implementation as a macro would be easier.

```
MPID_Pack_size( count, datatype, msgact, &size )
```

count Number of `datatype` items to pack.

datatype MPI datatype to pack.

msgact Message action

size Number of bytes needed to pack data (this may be larger than actually needed, though it should be "close").

If the destination buffer is too small to contain the packed/unpacked data, the error code `MPI_ERR_TRUNCATE` is returned. All `position` arguments are in bytes. The `msgrep` value is an integer that is accompanies the message (in the envelope) from the sender to the receiver and allows the implementation to pick the message representation dynamically (rather than fixing on XDR or network-byte-order for all messages).

4.2 Implementation of Noncontiguous Operations

A sample implementation of noncontiguous operations is provided in the MPICH implementation. This implementation uses a few features of the MPICH implementation. In particular, there are special `MPIR_Pack2` and `MPIR_Unpack2` routines that copy data to and from an `MPI_Datatype` and a contiguous array of bytes. These routines also handle checks on buffer sizes and can be used to move data that is shorter than expected. Depending on the length of the message, two different approaches are used. If the message is short enough, it is copied directly into the message envelope (which usually has a small data area). If the message is large, it may be copied into a temporary buffer, and that buffer may be sent as contiguous data. Alternatively, a long message may be sent in several separate messages (as is currently done in the `ch_shmem` device). As an implementation issue, this requires having a way to incrementally process a datatype. This requires saving some information on the current position in the datatype in the `request`.

The noncontiguous operations need to be careful of the datatype reference counts, particularly in the case of a nonblocking receive or a persistent operation. Note that the code

```
    MPI_Irecv( ..., datatype, ..., &request );
    MPI_Type_free( &datatype );
    ...
    MPI_Wait( request, &status );
```

must work, even though the user freed the datatype before completing the receive.

For example, an implementation of `MPID_SendDatatype` might send small amounts of data in the envelope (the data packet that contains the information on message tag, size, context, etc.). Large amounts of data are first copied into a buffer allocated just for this operation; the buffer is freed when the operation completes. This is sketched below. Note the use of a function `MPID_GetLen` to get the length of the message in the chosen destination `dest_format`; this format is passed to the sending routine. (The actual choice for the destination format is described in a forthcoming report.) Homogeneous implementations do not need `dest_format`, of course. (These examples use the `msgrep` field; a homogeneous implementation can ignore these, and even, through the use of macro redefinitions, eliminate them from the functions.)

```
len = MPID_GetLen(count, datatype, msgrep );
if (len <= MPID_MAX_PKT_SIZE) {
    MPID_PKT_T pkt;
    ... set pkt fields ...
    MPIR_Pack2( ..., &pkt.data );
    MPID_SendPkt( &pkt, .... );
    }
else {
    void *buf = malloc(len);
    if (!buf) {*error_code=MPI_ERR_EXHAUSTED;return;}
    MPIR_Pack2( ..., buf );
    MPID_SendContig( comm, buf, len, ...,
                          msgrep, error_code );
    free( buf );
    return;
    }
```

The routines `MPID_GetLen` and `MPID_SendPkt` are not part of the ADI specification; they are merely examples of routines that may be internal to the ADI implementation.

The implementation of `MPID_IsendDatatype` is similar; it uses fields in the `MPI_Request` to save the data:

```
len = MPID_GetLen(count, datatype, dest_format);
if (len <= MPID_MAX_PKT_SIZE) {
    MPID_PKT_T pkt;
    ... set pkt fields ...
```

```
    MPIR_Pack2( ..., &pkt.data );
    MPID_SendPkt( &pkt, .... );
    }
else {
    void *buf = malloc(len);
    if (!buf) {*error_code=MPI_ERR_EXHAUSTED;return;}
    MPIR_Pack2( ..., buf );
    request->shandle.start = buf;
    MPID_IsendContig( comm, buf, len, ...,
                  dest_format, &request, error_code );
    return;
    }
```

The MPID wait/test code for a send request for noncontiguous messages in the above implemenation includes

```
    if (request->shandle.start) {
        free( request->shandle.start );
        }
```

The field `start` in the request is an example that a particular ADI implementation may choose; it represents a pointer to a memory area allocated by the ADI for handling message data.

More sophisticated implementations could avoid allocating a buffer for the entire message; that is a "quality of implementation" issue. Note that a three-case implementation may be most appropriate: small amounts of data in the envelope, modest amounts in an allocated buffer, and large amounts sent in multiple parts.

5 Other Functions

The ADI must also provide a number of support functions for the MPI implementation. These are summarized below.

5.1 Queue Information

In addition to routines to receive messages, MPI provides routines to determine whether a message could be received. The corresponding ADI functions are `MPID_Probe` (blocking test) and `MPID_Iprobe` (nonblocking test). Since these perform message matching in the same way as the receive routines do, they have similar argument lists.

```
MPID_Probe( comm, tag, context_id, src_lrank, &error_code, &status )
MPID_Iprobe( comm, tag, context_id, src_lrank, &flag, &error_code, &status )
```

Just as for the receive routines (both blocking and wait/test on a receive request), the `status` argument may be null.

Note that `MPID_Probe` and `MPID_Iprobe` may need both relative rank and global rank, depending on how the message is tested for. See the Meiko device (in '`mpid/meiko`') for an example. However, since the communicator is passed to the device, it is as easy for the device to convert relative rank to absolute rank as it is for the MPI implementation. This is the same as for the corresponding `MPID_Recvxxxx` routines.

5.2 Miscellaneous Functions and Values

The ADI must provide routines to initialize and terminate use of the ADI (much like `MPI_Init` and `MPI_Finalize`) and to provide some information about the environment. First, here are the functions.

MPID_Init(int *argc, char ***argv, void *config_info, *error_code) Initialize the ADI. We may also want to include `char *envp`. Note the the MPI-2 Forum is considering severe restrictions on how an implementation may use the command line. The item `config_info` can be used to pass special information needed by the `MPID_Init` routine; current '`src/env/initutil.c`' passes `(void *)0` for this parameter.

MPID_End() Terminate the ADI.

MPID_Wtime(double *time) Give time in seconds. This has the same interpretation as `MPI_Wtime`; it is an elapsed time from some fixed but unspecified time in the past. This form simplifies implementing this routine as a macro; to further simplify that, the implementation should provide a '`mpid_time.h`' include file.

MPID_Wtick(double *tick) Give resolution of `MPID_Wtime` in seconds.

MPID_Node_name(char *name, int len) Give the name of the node.

MPID_Abort(MPI_Comm comm, int error_code, char *facility, char *string) Abort an MPI job with a message; if possible, return `error_code` to the invoking environment. If `string` is null, prints a default message. The string `facility` is used to indicate who called the routine; for example, the user (from `MPI_Abort`) or the MPI implementation. If null, it is ignored.

In addition, the ADI provides global variables that contain the size of `MPI_COMM_WORLD` and the rank of the process in `MPI_COMM_WORLD`. These values are `MPID_MyWorldSize` and `MPID_MyWorldRank`, respectively. These refer to `MPI_COMM_WORLD` only. This approach will simplify the eventual extension to dynamic process creation. By making these global variables, it is easier to handle the common special case of `MPI_COMM_WORLD` and to check for valid argument ranges for source and destination; it also simplifies writing error messages.

5.3 Context Management

The ADI is responsible for sending "context ids" as well as the message tags. Many implementations will implement the context ids as an additional integer that is sent with the message. For reasons of efficiency, the number of context ids may be smaller than can be expressed with a C `int` (for example, there may be a separate message queue for each context or a fixed number of bits in the message envelope). The maximum value of a context id is given by `MPID_MAX_CONTEXT_ID`.

When a communicator is created or freed, the ADI must be notified by calling `MPID_CommInit` and `MPID_CommFree`, respectively. These make sure that the ADI can manage any communicator-specific datastructures that it uses.

The routine `MPID_CommInit` is called when all other fields in the communicator have been set, except for the attributes and contexts ids. The routine `MPID_CommFree` is called before the communicator is freed. In other words, the ADI may use any field in the communicator as part of the initialization or freeing of a communicator. A typical use is in the case of a heterogeneous system; the appropriate message representation for all communication within the communicator can be determined when the communicator is created, rather than on each operation. Their form is

```
MPID_CommInit( oldcomm, newcomm )
MPID_CommFree( comm )
```

The communicators are, as always, of type `struct MPIR_COMMUNICATOR *`.

5.4 Support for MPI Attributes

MPI provides a number of predefined attributes; these need to be supported by the device. The support is through either global variables or defined constants; the choice is left to the implementation.

MPID_TAG_UB Maximum legal tag value

MPID_STDOUT Nodes capable of writing to `stdout`. This corresponds to the MPI attribute `MPI_IO`.

MPID_STDIN The node capable of reading from `stdin`.

MPID_WTIME_IS_GLOBAL Indicates whether `MPID_Wtime` provides a globally synchronized time.

MPID_HOST Rank of host, if any.

Note that `MPID_STDIN` has no corresponding value in the MPI specification. Unfortunately, the MPI specification has no way to distinguish between nodes capable of writing and nodes capable of reading; in addition, the requirement that all non-MPI routines be "local" means that at most one process can read from `stdin` at a time. While it is possible to implement this feature in a demand mode (`MPI_ANY_SOURCE`) by modifying the system `read` routine used by the application, it is often simpler to connect only a specific node (e.g., rank zero in `MPI_COMM_WORLD`) to `stdin`.

If these values are not defined, the implementation picks defaults (in '`src/env/initutil.c`'.

5.5 Collective Routines

Currently, there is no specific plan to provide a collective interface in the ADI, other than what is present in the function interface in the MPICH implementation (see '`src/coll/intra_fns.c`'. However, there will be some additional routines for communicator/context id management (as part of a separate effort to reduce the cost of handling `MPI_COMM_WORLD` and its duplicates). In fact, the interfaces provided by the first generation ADI (e.g., `MPID_Barrier`) are removed in favor of the `MPIR_COLLOPS` structure (which is in the MPICH implementation, not the ADI).

This is not meant to preclude the development of extensions to the ADI to support collective operations. In fact, such support is essential. However, the exact nature of this support requires additional research. One possibility is to provide support for simple barrier, broadcast, and gather operations. Other approaches might provide ways to optimize the point-to-point implementations. The interaction of the collective information with the topology of the physical (*not* logical) interconnect (and hence ADI support for MPI's topology routines) must also be taken into account. Because of the number of issues involved, ADI support for collective operations is left to a later note.

5.6 Changes from the First-Generation ADI

This section summarizes some of the changes from the previous ADI to this version.

The existing ADI functions, such as `MPID_Wtime` and `MPID_Init`, remain. The `ADIctx` argument is replaced by the MPI communicator. Some of the argument lists do change, however.

One important change to the `MPID_Probe` and `MPID_Iprobe` functions is to make them match the `MPID_Recvxxx` argument lists, so that the same code may be used in a probe as is used in a receive.

The routines `MPID_Myrank` and `MPID_Mysize` are replaced by the global variables `MPID_MyWorldRank` and `MPID_MyWorldSize`.

The macros `MPID_PACK_IN_ADVANCE` and `MPID_RETURN_PACKED` are removed; their function is handled by the `MPID_xxxDatatype` routines.

The macro `MPID_WTIME` is changed to return the time as an argument; just as for the error return in `MPID_SendContig`, etc., this makes it much easier to implement `MPID_WTIME` as a macro. The same is true for `MPID_WTICK`, though many implementations will probably continue to use the prototype routine which estimates the tick value by calling `MPID_WTIME`.

6 Special Considerations

A number of special issues need to be handled in the ADI. While these are often rarely used features in MPI, they affect the design and performance of the MPI implementation.

6.1 Structure of MPI_Request

The `MPI_Request` object is used for many purposes. The most obvious are the send and receive requests; internally, these are identified as requests with type `MPIR_SEND` and `MPIR_RECV`. MPI also has *persistent* communication operations; these look like sends and receives, but are actually different from an implemenation perspective. For example, a persistent send request needs to hold all of the data needed to start a send operation. This data is not needed as part of the request for other operations. The overloading of the request object is unfortunate, but we can reduce the confusion by putting the data needed by `MPI_Start` and `MPI_Startall` for persistent operations together and make it valid only for persistent requests. For this reason, there are two handle types: `MPIR_PERSISTENT_SEND` and `MPIR_PERSISTENT_RECV`. These are used to eliminate the separate check on whether the request is persistent. If generalized requests are added to MPI (MPICH already has some hooks for support for these), these two additional forms will be a small added complexity by comparison.

Persistent requests also require that additional information be saved. Since a persistent receive request may have wild-card fields (e.g., tag = `MPI_ANY_TAG`), these need to be preserved for the next use of the persistent request.

Another common problem is that an `MPI_Request` is used to represent an operation, such as a nonblocking send, at several different stages of completion. Thus, the *state* of the operation must be encoded within the request. In ADI-1, this was stored in `completer`. This value is no longer used; the sample implementation (see '`mpid/ch2/req.h`') uses pointers to functions in the request to indicate what routine to call to test or wait on a request; this pointer is changed depending on what needs to be done. Other implementations may want to use the `completer` approach.

Finally, it seems natural, at least in a receive request, to have an `MPI_Status` within the request itself, rather than separate fields. This approach simplifies moving data from the request to the `MPI_status`.

6.2 Freed Requests

MPI allows the user to do the following:

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

The MPI implementation is *required* to complete the communication even though no `MPI_Wait` or `MPI_Test` will be called by the user for the `request`.

Providing for the above example can be handled in a number of ways, none of them entirely satisfactory. One is to have a list of "abandoned" requests that the ADI would check; that is, the ADI must check that the list is empty (which should be the usual case). Another is to add an acknowledgement to communications, allowing the system to find and remove requests when they are actually completed. In cases where the ADI is involved in the details of data exchange (rather than passing data to another layer), a simple reference count or even a marker that the user had freed the request could be used. This probably needs its own interface, to allow the device to adapt to what it needs. However, how this is *implemented* does not affect the *design* of the ADI, other than to have a mechanism for indicating an "abandoned" request. This is accomplished with `MPID_Request_free`, which serves, for the ADI, the same purpose as `MPI_Request_free` serves in the MPI implementation.

6.3 Thread Safety

Most of this design is intrinsically thread safe. A possible exception is the test/wait routines, and this can be fixed by being precise about their definition. In particular, it is important that an `MPID_Wait` routine not assume that, since it was called, the message must still be waited on. In a multithreaded environment, some other thread may have completed the message between the time the MPI code decided to call `MPID_Wait` and when `MPID_Wait` actually got started. In particular, this situation forces `MPID_DeviceCheck` to be a no-op (at least nonblocking) in multithreaded versions. Note that the implementation examples in this document are not necessarily thread safe.

Because the ADI must consistently participate in any thread locks used in the MPI implementation, the ADI is responsible for defining what the thread lock interface looks like. The following macros are used to define the interface; they are shown with a sample implementation with a version of pthreads.

```
#define MPID_THREAD_DS_LOCK_DECLARE pthread_mutex_t mutex;
#define MPID_THREAD_DS_LOCK_INIT(p) pthread_mutex_init( &(p)->mutex, \
                                    pthread_mutexattr_default );
#define MPID_THREAD_DS_LOCK(p)      pthread_mutex_lock( &(p)->mutex );
```

```
#define MPID_THREAD_DS_UNLOCK(p)    pthread_mutex_unlock( &(p)->mutex );
#define MPID_THREAD_DS_LOCK_FREE(p) pthread_mutex_destroy( &(p)->mutex );
```

A typical use is

```
   typedef struct {
      MPID_THREAD_DS_LOCK_DECLARE
      other stuff
      } foo;
   ...
   foo *p = (foo*)malloc(sizeof(foo));
   MPID_THREAD_DS_INIT(p)
   ...
   MPID_THREAD_DS_LOCK(p)
   ...
   MPID_THREAD_DS_UNLOCK(p)
   ...
   MPID_THREAD_DS_FREE(p)
```

There are no global locks; each data-structure has its own lock. If the definitions are empty, the ADI provides no thread support (this is why the macro definitions include the statement-terminating semi-colon). Currently, the ADI does not have any direct support for the creation or scheduling of threads.

6.4 Cancelling a Message

Cancelling a message is a complex problem. The MPI standard has very strong requirements on the cancel operation, while the "typical" user often wishes only to handle cancelling a nonblocking receive that has not started. This case arises in multibuffering algorithms, where more nonblocking receives may be started than will eventually be used. In this case, cancelling the operation is usually quite simple; in the MPICH implementations, it usually means no more than removing the request from the posted receive queue (the queue of receives posted but which have not matched any send).

Cancelling requests in other states is much more complicated. For example, cancelling an `MPI_Isend` that has started but not completed can be very difficult. Again, we do not wish to add a lot of overhead to a request. A successful cancel of a receive must mark a request as complete, with the status field `count` in a receive set to `MPID_REQUEST_CANCELLED` (so that `MPI_Test_cancelled` can report it). There is also some text in the standard about cancels succeeding, suggesting that cancel can fail.

The MPID design only requires cancels of receives for which no matching message has yet arrived; this was the case that users actually wanted. Other cancel operations may fail. Note that a "failed" cancel still returns `MPI_SUCCESS` (as implied by the MPI standard).

The cancel routines are

```
MPID_SendCancel( handle, &error_code )
MPID_RecvCancel( handle, &error_code )
```

6.5 Error Handling

The MPID routines general make use of the `error_code` field to indicate errors. The MPICH implemenation will check these values and call the appropriate error handler. Note that the ADI routines are only required to set the values when there is an error, though they are allowed to set it to `MPI_SUCCESS` to indicate no error. This saves a potential store into a variable.

6.6 Heterogeneous Support

Most of the burden on support of heterogeneous systems has been moved from the MPI implementation to the MPID implementations that may be heterogeneous (currently, just the `ch_p4` and `ch_tcp` versions). However, there is one place where the MPI implementation needs to understand how heterogeneous data is handled (at least in part), and that is in MPI_Pack and MPI_Unpack. In addition, if the collective routines are implemented in terms of the point-to-point routines, a slightly more powerful form of MPI_Pack and MPI_Unpack are needed; these allow the pack and unpack formats to depend on the destination/source of the message, not just on the communicator. Perhaps the simplest approach is to view the existing MPIR_Pack2 and MPIR_Unpack2 routines as really MPID routines. This is the approach that we recommend.

If the ADI supports heterogeneity, it must define the macro MPID_HAS_HETEROGENEOUS.

7 Examples

This section shows some short test programs. These examples assume that a null communicator may be used instead of MPI_COMM_WORLD; note that they do supply an explicit context variable.

This first test shows a simple exchange of messages with a blocking, contiguous data. No error checking is done.

```
int main(argc,argv)
int argc;
char **argv;
{
char       buf[256];
int        ntest, i, len = 256, err, msgrep = 0;
struct MPIR_COMMUNICATOR *comm = 0;
MPI_Status status;

ntest = 10000;
err   = MPI_SUCCESS;

MPID_Init( &argc, &argv, (void *)0, &err );

for (i=0; i<ntest; i++) {
    if (MPID_MyWorldRank == 0) {
        MPID_SendContig( comm, buf, len, 0, 0, 1, msgrep, &err );
        MPID_RecvContig( comm, buf, len, 0, 0, 0, &status, &err );
        }
    else {
        MPID_RecvContig( comm, buf, len, 0, 0, 0, &status, &err );
        MPID_SendContig( comm, buf, len, 0, 0, 0, msgrep, &err );
        }
    }

MPID_End();
return 0;
}
```

This second test uses nonblocking, contiguous operations.

```
int main(argc,argv)
int argc;
char **argv;
{
char         buf[256];
int          ntest, i, len = 256, err, msgrep = 0;
struct MPIR_COMMUNICATOR *comm = 0;
MPI_Status   status;
MPIR_RHANDLE rhandle;
MPI_Request  request = (MPI_Request)&rhandle;
```

```
ntest = 10000;
err   = MPI_SUCCESS;

MPID_Init( &argc, &argv, (void *)0, &err );

for (i=0; i<ntest; i++) {
    if (MPID_MyWorldRank == 0) {
        MPID_SendContig( comm, buf, len, 0, 0, 0, 1, msgrep, &err );
        MPID_IrecvContig( comm, buf, len, 0, 0, 0, request, &err );
        MPID_RecvComplete( request, &status, &err );
        }
    else {
        MPID_RecvContig( comm, buf, len, 0, 0, 0, &status, &err );
        MPID_SendContig( comm, buf, len, 0, 0, 0, 0, msgrep, &err );
        }
    }

MPID_End();
return 0;
}
```

8 ADI Bindings

Here are the actual bindings for the routines. They can be found in the file 'mpid/ch2/mpid_bind.h'.

```
void MPID_Init ( int *, char ***, void *, int *);
void MPID_End  (void);
void MPID_Abort ( struct MPIR_COMMUNICATOR *, int, char *, char * );
int  MPID_DeviceCheck ( MPID_BLOCKING_TYPE );
void MPID_Node_name ( char *, int );
int  MPID_WaitForCompleteSend (MPIR_SHANDLE *);
int  MPID_WaitForCompleteRecv (MPIR_RHANDLE *);
void MPID_Version_name (char *);

/* SetPktSize is used by util/cmnargs.c */
void MPID_SetPktSize ( int );

void MPID_RecvContig ( struct MPIR_COMMUNICATOR *, void *, int, int,
                                 int, int, MPI_Status *, int * );
void MPID_IrecvContig ( struct MPIR_COMMUNICATOR *, void *, int,
                                  int, int, int, MPI_Request, int * );
void MPID_RecvComplete ( MPI_Request, MPI_Status *, int *);
int  MPID_RecvIcomplete ( MPI_Request, MPI_Status *, int *);

void MPID_SendContig ( struct MPIR_COMMUNICATOR *, void *, int, int,
                                 int, int, int, MPID_Msgrep_t, int * );
void MPID_BsendContig ( struct MPIR_COMMUNICATOR *, void *, int,
                                  int, int, int, int, MPID_Msgrep_t, int * );
void MPID_SsendContig ( struct MPIR_COMMUNICATOR *, void *, int,
                                  int, int, int, int, MPID_Msgrep_t, int * );
void MPID_IsendContig ( struct MPIR_COMMUNICATOR *, void *, int,
                                  int, int, int, int, MPID_Msgrep_t,
                                  MPI_Request, int * );
```

```
void MPID_IssendContig ( struct MPIR_COMMUNICATOR *, void *, int,
                                   int, int, int, int,
                                   MPID_Msgrep_t, MPI_Request, int * );
void MPID_SendComplete ( MPI_Request, int *);
int  MPID_SendIcomplete ( MPI_Request, int *);

void MPID_Probe ( struct MPIR_COMMUNICATOR *, int, int, int, int *,
                            MPI_Status * );
void MPID_Iprobe ( struct MPIR_COMMUNICATOR *, int, int, int, int *,
                             int *, MPI_Status * );

void MPID_SendCancel ( MPI_Request, int * );
void MPID_RecvCancel ( MPI_Request, int * );

/* General MPI Datatype routines  */
void MPID_SendDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                   struct MPIR_DATATYPE *,
                                   int, int, int, int, int * );
void MPID_SsendDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                    struct MPIR_DATATYPE *,
                                    int, int, int, int, int * );
void MPID_IsendDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                    struct MPIR_DATATYPE *,
                                    int, int, int, int, MPI_Request, int * );
void MPID_IssendDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                     struct MPIR_DATATYPE *,
                                     int, int, int, int, MPI_Request, int * );
void MPID_RecvDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                   struct MPIR_DATATYPE *,
                                   int, int, int, MPI_Status *, int * );
void MPID_IrecvDatatype ( struct MPIR_COMMUNICATOR *, void *, int,
                                    struct MPIR_DATATYPE *,
                                   int, int, int, MPI_Request, int * );

/* Pack and unpack support */
void MPID_Msg_rep ( struct MPIR_COMMUNICATOR *, int,
                              struct MPIR_DATATYPE *,
                              MPID_Msgrep_t *,
                              MPID_Msg_pack_t * );
void MPID_Msg_act ( struct MPIR_COMMUNICATOR *, int,
                              struct MPIR_DATATYPE *, MPID_Msgrep_t,
                              MPID_Msg_pack_t * );
void MPID_Pack_size ( int, struct MPIR_DATATYPE *, MPID_Msg_pack_t,
                                int * );
void MPID_Pack ( void *, int, struct MPIR_DATATYPE *,
                           void *, int, int *, struct MPIR_COMMUNICATOR *,
                           int, MPID_Msgrep_t, MPID_Msg_pack_t, int * );
void MPID_Unpack ( void *, int, MPID_Msgrep_t, int *,
                             void *, int, struct MPIR_DATATYPE *, int *,
                             struct MPIR_COMMUNICATOR *, int, int * );

/* Requests */
void MPID_Request_free (MPI_Request);
```

```
/*
 * These are debugging commands; they are exported so that the command-line
 * parser and other routines can control the debugging output
 */
void MPID_SetDebugFile ( char * );
void MPID_Set_tracefile ( char * );
void MPID_SetSpaceDebugFlag ( int );
void MPID_SetDebugFlag ( int );
```

Acknowledgments

We thank Jim Cownie for reminding us of the relationship between probe and receive. We thank Jonathan Geisler and Nathan Doss for their comments.

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html, 1995.