ANL/MCS-TM-00

MPICH Abstract Device Interface Version 3.3 Reference Manual

Draft of November 18, 2002

by

William Gropp
Ewing Lusk
Your Name Here
Mathematics and Computer Science Division
Argonne National Laboratory

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Contents

1 Introduction

The goal of the MPICH-2 ADI-3 is to provide routines to support MPI-1 and MPI-2 operations. The target systems include clusters connected with either conventional networks or networks that support remote memory access such as Infiniband, large symmetric multiprocessors (SMPs), and experimental systems (particularly in support of research into MPI implementations).

1.1 Other Work

This section will briefly review other MPI implementation designs.

An example of the use of both lock-free shared-memory operations and threads as MPI "processes" is presented in [**?**].

1.2 MPI Overview

In order to better understand the design of the ADI, we first review MPI communication. MPI-1 defined both point-to-point and collective message-passing. In both of these, the sender and the receiver actively participates in the communication in the sense that communication does not occur until an MPI call is made to initiate it and the data is not guaranteed to be delivered until an MPI call is made by the destination process. In fact, because the destination in memory of the data is described by an MPI call made by the destination process, the data cannot be delivered to its final (user-specified) destination until the matching MPI call is made by the destination process. This feature has led many (but not all) MPI implementations to adopt a *polling* mode of communication where communication for MPI happens only within MPI calls, not asynchronously. An alternative approach uses either one more more separate threads or an interrupt to cause communication to happen outside of MPI calls made by the user[1].

MPI-2 introduced additional operations including remote memory access (RMA), dynamic process management, and parallel I/O. Remote memory access (also called one-sided communication) provides a way to express put, get, and accumulate operations into memory in a remote process. In MPI, these operations are all nonblocking; to ensure that these operations are locally complete, an additional MPI routine must be called. MPI provides two "flavors" of RMA completion: *active target* and *passive target*. In active target, the target process of an RMA operation (that is, the process that did *not* initiate the RMA operation) must call an MPI routine before the RMA completes. The simplest such routine is `MPI_Win_fence`; this is a collective call over all processes associated with the RMA window and is similar to a barrier. The other is the more esoteric "scalable completion" routines, which are called by a group of processes. In both of these cases, an MPI implementation may rely on the target processes calling an MPI routine, and thus these may (but are not required to) use a polling implementation. Unlike some other RMA APIs, MPI active target RMA operations may be applied to any memory belonging to the process.

The other form of RMA completion is handled by calls made only by the originating process. This is called passive target RMA. Because the target process is not required to make any MPI calls, this kind of RMA requires either very capable hardware that can handle all MPI RMA operations or the use of a non-polling agent at the target process, or a combination of these. Because these operations can be more difficult to implement efficiently, MPI allows an MPI implementation to require that passive target RMA operations be allowed only on memory allocated by `MPI_Mem_alloc`.

MPI-2 dynamic process management allows an MPI application both to create new processes and communicate with them and to connect two already running MPI programs together. The number of processes in an MPI program can thus change over the lifetime of the program, though the MPI routines to create or connect to processes are collective over a communicator, allowing an MPI implementation to ensure that these operations are handled in a scalable fashion.

[1] MPICH-1 supported both modes but most implementations chose the polling mode. All ADI-2 implementation distributed with MPICH-1 used polling mode; however, some built by outside groups, such as the version for the Intel TFLOPS system, did *not* use polling mode.

Section 3 describes some of the design choices that these operations suggest. These choices are not the only ones possible, but we believe that they provide a consistent and efficient way to realize the communication defined by MPI.

1.3 A Layered Approach

The ADI described here is full-featured. This allows an implementor to take advantage of the opportunities for more efficient communication. However, to keep this flexibility from becoming a burden, the design of the ADI is also *layered*: the more advanced features can be emulated by the more basic features. The implementation of the ADI distributed with MPICH will include code to provide these more advanced features in terms of the more basic operations, allowing an implementor to quickly create a working implementation of the ADI and providing the opportunity to later enhance the performance by selectively replacing some of these emulations. Section 4 describes this in more detail.

The ADI described here resembles the communication routines of the MPI standard. The differences are

1. The MPI objects such as requests and communicators are not opaque objects; instead, the ADI uses pointers to structures with defined fields.

2. Checking for correct parameter values is not performed by the ADI routines; the implementation of the MPI routines can make these tests before calling the ADI routines.

3. Completion of MPI operations (e.g., `MPI_Wait`) is handled by a combination of busy flags in requests and ADI calls to make progress, rather than through calls similar to the MPI wait and test calls. There is no `MPID_Wait` or `MPID_Test` operation.

4. RMA (remote memory access) operations are complemented by a special interface for low-latency operations, particularly in the passive target case.

5. Collective operations are built out of point to point operations (though provision is made to replace each collective operation with an optimized function). An enhancement is planned that allows the use of pipelined store and forward and scatter/gather (to collections of processes) communication.

6. Dynamic process operations use a similar interface to MPI, but the process of building the new intercommunicator is made more explicit through the use of *virtual connections*.

In addition, the ADI is *not* responsible for the construction and management of other MPI objects such as datatypes, attributes, error handlers, and reduction operations. However, hooks are provided to allow the MPI level to notify the ADI of changes in the other MPI objects.

Most implementors will choose to use a simpler interface that is (will be) documented separately—the method interface. This defines a set of operations that are needed to implement communication on a single connection. This interface allows multiple communication methods to be used in a single MPI program, such as TCP, VIA, and shared memory.

An even simpler interface based on only nonblocking `readv` and `writev`-like operations will also be defined. This will replace the "channel" interface defined in ADI-2. An implementation of ADI-3 will be provided with MPICH that is built on this simple interface; the channel interface and this implementation are described in [**?**].

2 High-level Overview

The ADI is relatively rich in functionality. Before diving into some of the details, we provide a brief overview of the ADI. Communication in the ADI takes place between processes. We call the object that describes the communication between two processes a *connection*, and show a simplified block diagram of the objects used by the ADI in Figure 1. On each connection, we assume that the low-level

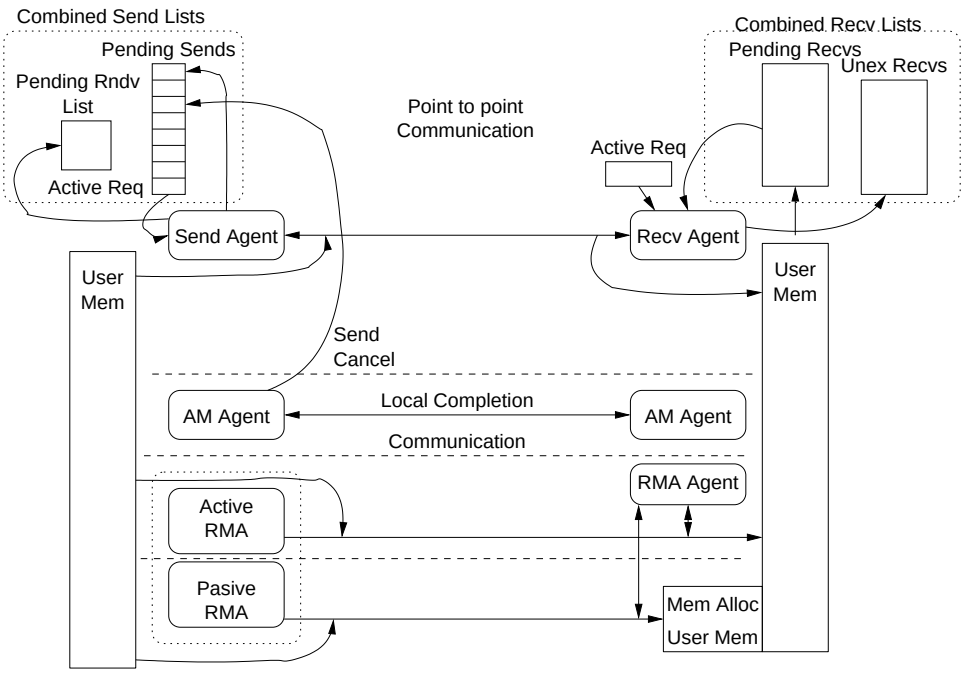

Figure 1: Simplified block diagram of the communication paths on a connection between two processes. The dashed lines separate the four communication types described below.

communication is *nonblocking*, thus there are queues of pending send operations and ordered lists of receive operations. The lines show communication paths; note that data may be moved from user memory to user memory without passing through the send or receive agents. This allows the ADI to support so-called zero-copy methods.

Because the ADI implementation is connection oriented, all details of the data transfer are hidden within the connection. This makes it relatively easy to support multiple communication methods between different processes; for example, the message format used with TCP communication need not look anything like the message format used by VIA communication within the same device. Similarly, flow control is handled on a connection basis, allowing the use of whatever method is appropriate for each communication method.

While the term "connection" is used, there is no requirement that an MPI program create all possible connections or use a connection-oriented low-level protocol. These are really *virtual connections*, with meaning only for the ADI and MPI layers.

It is important to bear in mind the difference between the operations that are sufficient to provide the MPI semantics and the operations that may be necessary to provide a higher-performance match between facilities provided by the hardware and software. The ADI endeavors to provide an effective compromise between a minimal set that can provide the full functionality and a richer (and larger) set that can exploit the capabilities of a wide range of systems. To bridge the gap between these, the MPICH ADI implementation provides some implementations of these higher-performance interfaces in terms of a smaller set of operations. (In ADI-2 used in MPICH-1, this smaller set was called the "channel interface".) A more complete discussion of this is presented in Section 4.

3 Basic Design Rationale

In this section, the basic operations that MPI requires of an ADI are described, as well as the design decisions that we have made based on these operations.

3.1 Communication Types

While all MPI interprocess communication (including MPI-1 and MPI-2) can be supported by a single, suitably powerful mechanism such as active messages, the MPI communication semantics described above suggest four separate types of communication operations. These are:

1. Two-party, point-to-point communication. This is the classic send-receive operation. This typically involves coordination between the sender and the receiver, handling such items as flow control, rendezvous messaging, and eager message delivery. Many low-latency implementations of this kind of communication rely on *polling* to advance the communication (make *progress* in MPI terms). Others may use a separate thread or an interrupt-driven mechanism (or a hybrid of both polling and non-polling). Because MPI semantics support nonblocking communication of arbitrary message sizes, any low-level support for communication in MPI must provide nonblocking, point-to-point communication. This is most easily accomplished if the low-level communications is also nonblocking.

2. Communication that has the property of local-completion. That is, a communication operation that must complete independently of any explicit action by the target (destination) process. This is required in MPI-1 for the implementation of `MPI_Cancel` (in the send case in most implementations) and is useful for `MPI_Abort`. In MPI-2, local completion is also needed for *passive target* remote memory access (RMA) operations. This kind of communication is typically implemented through the use of active messages or remote service requests (without polling).

3. Communication for active-target RMA. In MPI, active target RMA operations include remote put, get, and accumulate. These operations are completed by either an `MPI_Win_fence` made call by all processes in the MPI window object or by the combination of `MPI_Win_complete` and `MPI_Win_wait` at the origin (process that initiating a put, get, or accumulate) and target (process containing the memory accessed by a put, get, or accumulate) processes. This form is similar to the two-party, point-to-point communication because it can be implemented by using a pure polling interface. This communication form is separated from the point-to-point mode because the hardware in some systems allows some of the active-target RMA operations to be implemented directly by hardware or low-level software.

4. Communication for passive-target RMA. These operations must complete locally. If these operations must be implemented by communicating with an agent at the remote process, then some form of non-polling agent is required, such as an interrupt-driven active message or a separate communication thread. As in case three, this communication form is separated from case 2 (local completion for MPI-1) because the hardware in some systems allows passive target RMA operations to be implemented directly. We expect some systems to provide an extensive set of operations (e.g., direct access to memory on an SMP through a shared memory segment), others to provide more limited access (e.g., remote DMA through special network support), and others to be implemented on top of a non-polling communication layer such as active messages. The ADI design is intended to provide a common set of entry points independent of the capabilities of the underlying system.

The ADI design makes no explicit choice between polling and non-polling implementations. Instead, it defines several kinds of polling *points* but allows a purely non-polling (interrupt-driven or separate communication thread) implementation as well.

The four types of communication, of course, can be implemented by a single, suitably powerful abstraction. However, achieving high performance (particularly low latency) requires abstractions that are close to the operations that are efficiently implemented in hardware. The emergence of remote access-style operations in networks [**?**, **?**, **?**] encourages their use as primitives (communication types 3 and 4). Efficient handling of message-passing, particularly the need to manage the flow of point-to-point communications and scalable collective communications on top of more conventional two-sided communications such as TCP suggests communication type 1. Finally, the need to handle some MPI-1 operations that are infrequent and not performance sensitive, but must be handled

```
(sketch of figure to be completed later)
1 can implement 3
2 can implement 4
1 can provide emulation of 2, but only approximately
```

Figure 2: Sample dependencies between communication approaches

reliably, suggests communication type 2. The top level ADI interface provides direct access to these four types of communication. Of course, the implementation of the ADI may implement some of these communication types in terms of others, such as reducing all four types to type two (active messages).

3.2 Additional Goals

To ensure that short messages have the lowest possible latency, the common cases should have direct paths to the low-level data transfer operations. In particular, the MPI and API layers should allow operations to to complete without requiring the creation of intermediate data structures. For example, sending a single word (particularly from a blocking `MPI_Send` operation) should not require creating and initializing internal data structures. However, to maintain a simple code base, we will strive to use common code. This suggests that a basic operation is a simple "attempt to send;" this allows the ADI to attempt to send a short data message and return success without creating, for example, a queue element that holds a pending communication operation. Only if the data cannot be communicated immediately will the ADI create an intermediate data structure to hold the state of the incomplete communication.

3.3 Other Relevant MPI Issues

MPI defines a number of objects such as requests, windows, and communicators. In many cases, these objects are natural choices for use within the ADI. Using these objects directly, rather than defining different objects for use by the ADI and translating between the MPI and ADI versions, avoids unnecessary overhead within the device. Let us look at several of these objects.

3.3.1 Structures Involved in Communication

MPI Requests. The use of nonblocking operations to implement the low-level communication requires an object to hold the current state of the communication and to record completion of the operation. The natural place to store this within the MPI request. Pending send operations must also be saved in a first-in-first-out queue (to maintain the message ordering guaranteed by MPI); this suggests that the queue contain MPI requests. On the receive side, the message-matching defined by MPI suggests saving the requests in an ordered list. Note the asymmetry between sends and receives. On the send side, requests are placed in a strict FIFO queue for each communication path (to maintain ordering of messages). On the receive side, requests for unmatched receives are kept in an ordered list, and this list, because of the wildcard source receive (`MPI_ANY_SOURCE`) is (logically at least) shared by all communication paths. Similarly, requests for unexpected messages (messages sent but for which no matching receive has yet been issued) are kept in an ordered list. Requests are also the logical place for the data structures relevant to packing and unpacking from complex datatypes to simpler layouts such as contiguous data buffers. This is discussed in more detail under MPI datatypes.

MPI Datatypes. MPI communication can be specified using datatypes that describe complex layouts in memory. An MPI implementation must convert these descriptions into data layouts that can be conveniently moved by the low-level communication layers. Such layers typically support only contiguous memory regions or Unix "io vectors" (`struct iovec`); MPI provides more general forms of data layouts. However, while the MPI datatypes are sufficient to express most forms of communication, there are no MPI routines to pack or unpack only a fraction of a datatype. For

example, there is no MPI-defined method to pack as much as fits into a fixed sized buffer and return enough state so that a subsequent pack can pick up where the last left off. Such an operation is needed for any algorithm that packetizes data or that pipelines data transfers. Because this operation is needed both to handle packing noncontiguous data into temporary buffers needed by low-level communication routines (such as TCP `write` or `writev`) and by high-performance algorithms for collective communication, we have introduced a new data structure that is used to pack and unpack buffers described by MPI datatypes. This new structure is called a *segment* and is stored in a structure type named `MPID_Segment`. Segments are discussed in more detail in Section 3.6.

Thus, to handle the need to pack and unpack data, MPI requests also contain a segment. Combined with the datatype and the user-buffer, this gives enough information to move the data to and from the user buffer, even if an intermediate buffer is needed. Note that where possible, no intermediate buffer is used. For contiguous data, and for more general data formats that the underlying communication level supports, data can be moved without using any extra buffers. Segments are required to handle the general case.

Consequences. These considerations suggest that the MPI request (more specifically, the internally-defined structure to which an MPI request refers, which is `MPID_Request`) is the key object. The `MPID_Request` is used to store the progress of communication and order communication between processes. The ADI will use the request as its basic object.

The consequence of this is that the ADI point-to-point communication routines should usually return a request. The only exception is that blocking communication routines should not return a request if the communication is already complete. This allows the blocking communication routines to return completion without ever creating and managing a request. This suggests that the ADI interface for point-to-point communication look something like the following:

```
MPID_Send( buf, count, datatype, tag, <communicator info>, &request )
MPID_Isend( buf, count, datatype, tag, <communicator info>, &request )
MPID_Issend(...)
... similarly for other point-to-point functions
```

The exact form of the arguments that specify the communicator information will be discussed later. This allows a "blocking" send to return a request if the operation has not completed, giving the calling routine more control over what steps to take to complete the request. It sets the request pointer to NULL if it was able to complete without creating a request.

The blocking receive case is similar to the blocking send case. If, when the receive routine is called, the data is available, no request should be created (the cost of creating a request isn't the real issue, it is the cost of initializing and managing the request).

Recall that in the receive case, there are two kinds of receives: posted (but unmatched by an incoming send) and unexpected (sent but unmatched by a receive). For thread-safety, operations on these two lists must be made atomically. These operations include

- check posted and return if found; else insert into unexpected
- check unexpected and return if found; else insert into posted

The MPID request is the appropriate list element to use in constructing these structures.

While the request is allocated by the ADI, `MPID_Datatype`s are allocated by the MPI implementation. In fact, most of the MPI objects, except for requests, will be allocated by the MPI implementation rather than within the ADI. This is an arbitrary choice; for greatest generality, the ADI could be responsible for allocating all MPI objects. However, we believe that most users of MPICH and ADI-3 will not need that flexibility, and managing the objects within the MPI layer instead of the ADI layer simplifies the implementation of the ADI. However, to make provision for any ADI-specific features that must be associated with an MPI object, the definition of the structure associated with each object includes

```
MPID_DEV_xxxx_DECL
```

where xxxx may be COMM, DATATYPE, etc. This provides a simple way to extend the objects defined by the MPI layer without forcing the ADI to provide a complete implementation. Whenever an object is created or destroyed, the MPI layer can call a *hook* routine with a name of the form

```
MPID_Dev_xxxx_create_hook( pointer to object, ... )
MPID_Dev_xxxx_destroy_hook( pointer to object, ... )
```

The other parameters will be defined as it becomes clear what is needed. For example, in the case of communicator creation, a pointer to the old communicator may be needed.

These routines are called after all other creation operations take place and before any of the destroy operations take place.

3.3.2 Communication Contexts and Groups

Communicators. MPI Communicators describe both a collection of processes (a MPI_Group) and a unique communication context. As described later, in MPICH and ADI-3, the communication context is encoded as an integer. The target process of communication in MPI is described by a rank in a group associated with a communicator. While this suggests that MPI groups are fundamental data structures, in MPICH-2, groups are not used in the ADI at all. Instead, each communicator maintains an array of *connections* that are indexed by the rank. The MPI implementation provides complete support for the MPI group operations (e.g., MPI_Group_union), but the ADI does not use groups at all.

3.4 Completing Point-to-Point Operations

To complete some particular MPI communication (described by a request, such as in a call to MPI_Wait, it is necessary to have the ADI respond to *any* pending communication. Thus, it is not necessary to provide the ADI with a collection of request to test or wait on. Instead, we merely need to ask the ADI to try to make progress on communication and they check (using the busy flag in each request) whether any particular MPI requests have completed. In the absence of threads, a simple interface would look like

```
MPI_Waitsome( ... )
{
while( no completed requests found )
    for (i=0; i<count; i++) {
        if (any requests done, return those as complete)
    }
    MPID_Make_progress( TRUE );
}
```

However, if there are multiple threads, particularly if there are separate threads that can complete communication, then this API is incorrect. Instead, a slightly more complex interface is needed to eliminate any race conditions. For example,

```
MPI_Waitsome( ... )
{
  while (1) {
    MPID_Progress_start( );   // Notes that we are about to
                              // check ready flags.  No busy
                              // flags will be cleared
    for (i=0; i<count; i++) {
        if (any request done) { save info on request }
    }
    if (no requests done)
        MPID_Progress_wait();
```

```
        else {
            MPID_Progress_end();
            break;
        }
      }
    }
```

The interface for test operations is similar, except that `MPID_Progress_wait` is replaced with `MPID_Progress_test`, and no outer loop is needed.

In a polling implementation, the "start" and "end" calls are no-ops and the "wait" and "test" calls are blocking and nonblocking polling calls respectively. In a nonpolling implementation, the "start" and "end" calls may set and clear a thread lock or access lock, and the "wait" and "test" calls may yield to a communication thread (in addition, the wait version could wait to be signaled through a condition variable). This interface allows us to use the same code for completing MPI nonblocking operations independent of the choice of polling, nonpolling, threaded, or nonthreaded implementations of the ADI.

3.5 Supporting Collective Operations

When implementing collective communication algorithms, the ability to both store and forward data is important for performance. For example, when complex MPI datatypes are used, it may be necessary when receiving data to first receive into a temporary buffer and then unpack that data into the user's buffer. Forwarding this same data on (for example, within a broadcast) in a separate MPI send operation then requires repacking the data from the user into a temporary buffer. To enable an ADI implementation to avoid this cost, and to make it easier to efficiently write the collective communication algorithms, the ADI provides a variety of store and forward, scatter, and gather operations. Note that these operations can be emulated using only point-to-point; as described above, these can be built on top of simpler, point-to-point communication.

Consequences. Determining completion in the store and forward or multisend cases may involve more than one communication operation and possibly multiple communication methods. This argues that the status of a communication be tracked with a completion counter rather than a simple flag, since multiple communication operations may be working with the same data buffer.

To best exploit the fact that the same data is both being received and sent, the ADI should be able to provide pointers to "good" memory for these operations. For example, such memory may be in a special, pinned page or within a sophisticated NIC.

The algorithms for efficient collective communication provide some information on the kinds of multi-party operations that are required. The ADI does not support the most general of these operations; the goal is to allow an MPI implementation to efficiently and correctly support the more common collective communication operations such as `MPI_Bcast`, `MPI_Scatter`, and `MPI_Allgather`.

The very first version of ADI-3 may not include these more general multi-party communication operations. It is the intent of ADI-3, however, to develop an efficient method for describing and implementing the operations needed for the MPI collective operations.

3.6 Data Segments

MPI datatypes are very general; a single instance of a datatype can describe an arbitrarily large amount of data that is not necessarily contiguous in memory. Further, MPI datatypes can be very concise; a vector datatype of a structure (that itself is not contiguous) describes a very complex memory layout with just a few words of memory. Because few if any low-level communication layers support the full generality of MPI datatypes, it is sometimes necessary to pack and unpack data to intermediate buffers. In addition, it may be necessary to pack or unpack a single MPI datatype with multiple calls; this operation is not supported by the `MPI_Pack` and `MPI_Unpack` routines. For example, the code

```
MPI_Type_vector( 1000000, 1, 237, MPI_DOUBLE, &newtype );
MPI_Type_commit( &newtype );
MPI_Send( buf, 1, newtype, ... );
```

passes a single instance of an MPI datatype that describes 8 MB of data. Further, because this is a vector type, it cannot be represented efficiently with a `struct iovec`. If the underlying communication layer can only send a maximum of 64K at a time (for example, using a shared memory pool or a remote-memory communication area), it is necessary to incrementally pack this datatype, 64K at a time, into a temporary buffer. The segment routines provide this capability.

These routines are also needed for some implementations of the collective communication routines. Many of the better (and in some cases, the best) algorithms for the collective operations in MPI can be cast, at least for systems that are homogeneous is data representation, in terms of operations that view the data to be communicated as a contiguous range of bytes and that send different parts to different processes. To implement these algorithms for MPI requires handling MPI datatypes; even in a single collective operation, some MPI processes may provide a simple, contiguous data buffer while others specify a complex datatype. To implement these algorithms requires a method to extract parts of a data buffer, viewed as a range of bytes. This is a simple variation on the routines that can incrementally pack (and unpack) a data buffer described by an MPI datatype.

Question: We need to tie down the details of the segment creation and pack/unpack operations.

Issues with the segment routines:

1. Who allocates storage for the segment? That is, where does the contiguous buffer come from? In many cases, it would be nice if it was memory that was convenient for the ADI-3 device. In the case of shared memory or RMA-networks, using special memory saves a memory copy.

2. Who sets the amount of data to pack or unpack in each call? The specific concern here is to not stop in the middle of a natural data item, e.g., 3/8ths of the way through a double. The design here allows the routines to make slight adjustments in the amount of data packed or unpacked in order to stay on "natural" boundaries.

3. How much do these routines need to know about MPI Datatypes? We'd like them to be generic so that they could be shared with other projects such as PVFS and HDF5.

3.7 Remote Memory Access

The MPI remote memory access model was deliberately designed to have very loose synchronization requirements and to make minimal *demands* on the underlying hardware. For example, it is possible within the MPI model to support non-cache-coherent systems (such as the NEC vector supercomputers)[2]. However, the model also *allows* an MPI implementation to exploit special hardware capabilities.

The MPI specification is very careful in describing when a process's memory window is accessible to other processes (the *exposure epoch*) and when a process may be performing RMA operations (the *access epoch*). Understanding these is necessary in developing a correct MPI implementation. However, these concepts are deliberately made as general as possible to allow the greatest flexibility to an MPI implementation. In the discussion below, we will usually not refer to the access or exposure epochs. However, if there are questions as to what the terms "synchronization" or "completion" mean in the RMA context, consult the discussion of the RMA epochs in the MPI standard.

The MPI specification has a number of "as if" rules, such as "as if only one process accesses a memory window at a time". Naturally, if an implementation can perform an operation more efficiently without violating such "as if" rules, the implementation is free to do so. An example of this is passive target updates to disjoint regions in a memory window; the "as if" rule says that these must appear "as if executed sequentially," but an implementation can allow concurrent updates if they are known a priori to be disjoint. This suggests that operations be aggregated so that the range of affected bytes within the target window is known. Fortunately, the MPI specification allows aggregation.

[2]The ADI-3 design, however, does assume cache coherence.

3.7.1 RMA Aggregation

One of the most misunderstood parts of the MPI RMA specification is the issue of when operations take place, particularly with the poorly named `MPI_Win_lock` and `MPI_Win_unlock` routines. MPI RMA allows the implementation considerable latitude in the timing of operations. An important case in point is aggregation (combining) of RMA operations. The approach of aggregating RMA operations has been developed in the BSP approach, and the MPI specification was designed to support this technique. For example, the following sequence of MPI calls

```
MPI_Win_lock( MPI_LOCK_SHARED, rank, 0, win );
MPI_Accumulate( &one, 1, MPI_INT, rank, 0, 1, MPI_INT, MPI_SUM, win );
MPI_Win_unlock( rank, win );
```

can be converted into a single, atomic update operation on some systems (particularly on those that have no direct access to remote shared memory, such as a system that only supports TCP communication).

To allow low-latency implementation of single-element remote updates (e.g., put or accumulate), the ADI design allows the MPI implementation to perform aggregation of RMA operations without calling the ADI. This eliminates a layer of function calls in these simple cases[3]. The ADI provides a set of definitions that are used to decide the aggregation threshold, in terms of number of bytes and operations. The device can also specify that no aggregation is done at the MPI level, giving the ADI greater control at the cost of additional function calls.

3.7.2 Nonblocking RMA and Remote Completion

The MPI RMA operations are nonblocking. Thus, there must be some way to indicate both local and remote completion. MPI provides users with three different mechanisms for marking completion in their code:

Fence. This is essentially a barrier synchronization, similar to the `shmembarrier`. To allow the ADI to exploit hardware and software features similar to those used by Cray `shmembarrier` in the Cray T3D and T3E, the ADI provides a similar routine, with the difference that it applies to MPI window objects on arbitrary groups of processes. An ADI implementation for a system that provides an efficient fence operation only on all processes can, of course, test for that case and execute different code when not all processes are involved in the MPI window object.

Passive. This is a kind of two-party synchronization since only the origin and target processes are involved, rather than all members of the group of the MPI window object. The ADI provides a simple "busy" variable that is zero on completion; this allows the flag to be a counter that contains the number of uncompleted operations or a simple boolean that indicates whether the operation is busy. The MPI calls that express this kind of synchronization are the misnamed `MPI_Win_lock` and `MPI_Win_unlock`. Note that calls by a process to `MPI_Win_lock` for its own rank (i.e., "lock my window") are different in behavior from calls to a remote process because the local process may use non-MPI operations to access the memory window (e.g., through simple references or assignments). ADI-3 assumes a cache-coherent memory system, which allows some important simplifications in handling these operations.

Scalable Multiparty. In this mode, not all processes in the window are involved (unlike the fence case), but both origin and target processes make calls to indicate when operations must complete. The MPI calls used to express this kind of synchronization are `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete`, and `MPI_Win_wait`. This form can also be handled with counters that are updated by each partner process. Efficient handling of this approach remains a research issue, however.

[3] In addition, the MPI functions could be inlined by a suitably sophisticated compiler, removing all function calls

3.8 Contexts for Collective, File, and Window Operations

MPI requires that different kinds of communication be non-interfering. That is, communication for collective operations such as `MPI_Bcast` and point-to-point operations such as `MPI_Send`, even on the same communicator, must not interfere with each other. With MPI-2, this is extended to operations on MPI Files and Window objects, both of which may involve some communication by the implementation.

MPI also requires that communication within different communicators be noninterfering. Most implementations achieve this by using a hidden (to the MPI user) *context id*, which is simply an integer that is communicated along with the tag, source, etc. One approach that can be used to ensure that communication of different types be non-interfering is to use different communicators; in MPICH-1, each communicator was created with a second, hidden (from the user) communicator that was used for communication implementing collective operations.

However, this approach has a number of drawbacks, particularly in organizing the code. In addition, with files and windows as well, three hidden communicators (in addition to the one used for point-to-point communication) might be needed[4]. Finally, the implementation of intercommunicator collective operations, also introduced in MPI-2, adds additional complexity. In MPICH-2, we take a different route. Instead of creating hidden communicators, we allocate context ids in groups of four. The API routines for communication take an explicit context id offset as a parameter. These offsets have the following values. For intracommunicators: 0–point to point communication, 1–collective communication, 2–communication for files, 3–communication for window objects. For intercommunicators: 0–point to point, 1–collective in group A, 2–collective in group B, 3–collective over both groups (Note that a single context value for "collective in local group" is adequate; however, having separate context values provides stronger separation and aids in debugging.)

3.8.1 Context Id Generation

The operation `MPI_Comm_dup` should be efficient. In particular, where possible, it should be a local operation (involving no communication). This is possible, at least for the first few dup's of a communicator, if each communicator caches some extra context ids when it is created. The MPI implementation will maintain a cache of context values in a communicator; communicators created by dup'ing that communicator will take a value from the cache. If the cache is empty, the MPI level will make a collective call to the ADI-3 routine that returns context ids. To simplify this, the ADI-3 routine returns a single value; the MPI level will multiply this by a fixed constant size to create a sequence of consecutive context ids. In addition, the MPI level will keep track of returned context ids (made available by freeing a communicator) so that the ADI can be told when a context id is available again.

3.9 Dynamic Processes

There are two major goals here for the ADI: scale to large but not necessarily enormous numbers of processes and maintain modularity so that it is relatively easy to maintain and extend the implementation. A secondary goal is to structure the code so that initialization and rundown are efficient and do not spend a great deal of time setting up facilities that a program never uses.

MPI-2 adds dynamic process management. A consequence of this is that there are no absolute and global process ids. This observation suggests that all communication be considered locally in terms of (possibly virtual) connections to processes. Unlike ADI-2, there are no data structures that map a rank in a communicator into a "global rank" (i.e., rank in `MPI_COMM_WORLD`). Instead, communicators have arrays of virtual connections that are indexed by the rank.

There is a kind of local process id that is used by the MPI group operations. However, it is not critical item and, since the MPI group operations are implemented above the ADI, we do not discuss them here.

[4] An alternative is to perform a shallow duplicate (without invoking attribute copy functions) of the communicators passed into the file and window creation routines.

4 ADI Layers

It is expected that implementations of the ADI will be *layered*, building the four types of communication described in Section 3.1 in terms of simpler communication methods. This section outlines several possible implementations.

4.1 Socket (TCP) communication

Socket-based communication provides a two-party communication similar to the type 1 communication in Section 3.1. A simple implementation can map all four kinds of communication into simple `read` and `write` calls as follows (with some caveats):

1. Point to point communication is a close match. Some care is needed to handle flow control.
2. Local completion communication is more difficult. A simple (and not entirely correct) approach is to simply convert these operations (e.g., `MPID_Cancel_send`) into a message that is sent on the same socket as the type 1 point to point communication. All that this requires is that messages sent on the socket connecting two processes include a header that describes the type of message (e.g., MPI message envelope, MPI cancel message, MPI cancel acknowledgement, etc.). For those familar with the channel device in ADI-2, these are just the packet types.
3. Active target RMA. This can also be converted into simple messages sent on the same socket. Note that there are some complications in handling complex MPI datatypes.
4. Passive target RMA. Just like active target RMA. Note, however, that for correct behavior, the receive agent must be called at least occasionally, even if the target process makes no MPI calls.

The progress engine is implemented by using `select` or `poll`, and is a pure polling approach.

To correctly handle the local completion (type 2) and passive target (type 4) communication when using sockets, there must be an asynchronous communication agent. One easy way to do this is to take the progress engine and place it in a separate thread. The operating system then guarantees that the progress engine is called often enough to ensure that the locally completing communication operations make the necessary progress. However, to illustrate the advantages of the ADI design (in terms of four separate communication types), consider a different version where there are two sockets between communicating pairs of processes. The first socket is used in a polling mode for point-to-point and active target communication (types 1 and 3). This provides the low-latency associated with polling models. The second socket is used for the locally-completing communication (types 2 and 4), and uses a separate thread, process, or `SIGIO` to ensure local completion. This approach provides correctness with the MPI progress model without sacrificing the low latency of the polling approach for the more common operations.

Of course, other approaches are possible, including one that uses a mix of polling and nonpolling even on type 1 and type 3 communication.

4.2 Remote Put

Not yet written

4.3 Shared Memory

Not yet written

5 Summary

This section provides a short summary of the ADI routines. Complete details are provided in the ADI-3 reference manual.

5.1 Point to point communication

Each simple MPI communication operation has its counterpart here:

```
MPID_Send
MPID_Ssend
MPID_Rsend
MPID_Isend
MPID_Issend
MPID_Irsend
MPID_Recv
MPID_Irecv
MPID_Request_free
MPID_Iprobe
MPID_Probe
```

Note that these do not include the buffered send nor the two send-receive operations (`MPI_Sendrecv` and `MPI_Sendrecv_replace`). Send-receive operations are described in terms of separate send and receive operations; buffered send may use the optional `MPID_tBsend` or may simply use `MPID_Isend` with the sample implementation described in the MPI-1 standard.

Several MPI routines are represented by slightly different routines. These include

```
MPID_tBsend        - Used to optimize buffered sends (MPI_Bsend etc.)
MPID_Cancel_send   - Separate routines for cancelling sends and receives
MPID_Cancel_recv
```

In addition, the persistent communication routines have MPID equivalents:

```
MPID_Send_init
MPID_Ssend_init
MPID_Rsend_init
MPID_Recv_init
MPID_Startall
```

These are provided so that the device can effectively manage the persistent request, which may require allocating a device-specific request either when an init routine (e.g., `MPID_Send_init`) is called or when the request is started with `MPID_Startall`.

Note that there is no direct access to the message queues by the MPI layer. We recommend that the device implement the message queue interface defined for external tools and used by Totalview. This interface is described in [**?**] and the files `mpich/src/infoexport/` in MPICH-1.

5.2 Completion of Point to point communication

Completion of nonblocking (or incomplete in the case of a request returned by `MPID_Send`, `MPID_Ssend`, `MPID_Rsend`, or `MPID_Recv`) is handled by calling the progress engine routines and checking the `busy` value in a request. The progress engine routines are

```
MPID_Progress_start
MPID_Progress_end
MPID_Progress_test
MPID_Progress_wait
```

This set of routines is required in order to provide an interface to the `busy` flag that is thread-safe. In addition, there is a routine that indicates *polling points*; places in the MPI code where a polling implementation should check for incoming messages. This routine is nonblocking and is

```
MPID_Progress_poke
```

5.3 Data Segments

The routines to incrementally pack and unpack a data buffer given a user buffer and an MPI datatype are:

```
MPID_Segment_pack_init
MPID_Segment_pack
MPID_Segment_unpack_init
MPID_Segment_unpack
MPID_Segment_free
```

These are used to implement both the `MPI_Pack` and `MPI_Unpack` routines as well as in the default implementations of the MPI collective communications routines. Most implementations of the ADI-3 will also use these routines to handle packing and unpacking messages defined by MPI datatypes.

Comments: For some MPI datatypes, it is sometimes possible to describe the buffer with a Unix-style `struct iovec`. Many devices provide low-level and sometimes even efficient for `iovec` descriptions. One unanswered question is whether we should have a `MPID_Segment_pack_iovec` or whether there should be a way to check whether the Segment routines are needed at all (i.e., they aren't needed for datatypes that are contiguous or that are described efficiently by a `struct iovec`).

5.4 Starting and stopping

The routines to start and stop the ADI match the MPI counterparts. Note however that there are no calls to implement "is initialized"; this is managed entirely at the MPI level.

```
MPID_Init
MPID_Finalize
```

5.5 RMA

The RMA routines haven't been set yet. However, they will include MPID versions of put, get, and accumulate, along with a few calls to handle the aggregated operations (e.g., lock/accumulate/unlock). In addition, we expect to separate the `MPI_Win_lock` and `MPI_Win_unlock` into two kinds of operations: non-local, where the operation is really a start/end RMA operation, and local, where it really is lock/unlock.

One question is whether there should be support at the MPI layer for caching data type descriptions (e.g., for complex MPI datatypes to be applied at the target process) and the corresponding ADI routines, or whether this should be handled entirely by the ADI.

The ADI routines that support RMA include

```
MPID_Win_put
MPID_Win_get
MPID_Win_accumulate
MPID_Win_do
MPID_Win_fence
MPID_Win_start
MPID_Win_end
MPID_Win_local_lock
MPID_Win_local_unlock
```

`MPID_Win_do` is used to send aggregated operations to the ADI. This provides a pointer to a description and an indication of whether this starts, ends, or continues a sequence of RMA operations. `MPID_Win_start` and `MPID_Win_end` are used for nonlocal uses of `MPI_Win_lock` and `MPI_Win_unlock`.

We may want `MPID_Win_do_put`, `MPID_Win_do_get`, and `MPID_Win_do_accumulate` instead of a single `MPID_Win_do`.

5.6 Dynamic Processes

The MPID routines are similar to the MPI routines. Undecided: does the MPI layer or the ADI layer setup the communicator? The ADI layer needs only provide the pointers to the connection structure; the MPI layer could build the communicators and access the context id values.

5.7 Device Hooks

To allow the device to track the creation and destruction of MPI objects (other than requests), the device may define hook routines. These have the form

```
MPID_Dev_xxx_create_hook
MPID_Dev_xxx_destroy_hook
```

where `xxx` is the object type, such as `comm`, `datatype`, `group`, etc. Many devices will define these as C preprocessor macros that expand to nothing, thus eliminating any function calls.

6 Integrating a New Device into the MPICH Build Tree

This section describes how to add a device or method into the MPICH build tree. This section is intended both to describe the process of adding a device and the rationale for the design of the device-dependent modules.

6.1 Steps Needed in Implementing a Device

1. `mpiexec`. Define how `mpich` creates the correct `mpiexec`. There must be an `mpiexec` target in the device's `Makefile`.

2. Installation. Define how `mpich` gets the proper device-specific files installed; e.g., if the device uses the multi-purpose demon (`mpd`), ensure that the `mpd` is installed. Provide an `install` target in the device's `Makefile`. For example, in the `Makefile.sm`, use the line

   ```
   install_BIN = mpd
   ```

   to install the program `mpd` into the `bin` directory.

3. Device-specific documentation, such as environment variables and command-line arguments used only by a particular device. Place this information into the file `devdoc.txt`. The `mpich` documentation generators will look for this file.

4. Testing codes for device-specific functions. Place these in a `test` subdirectory of the device. These tests should be performed through a `test` target in the device's `Makefile`.

5. Include files. Any include files that are needed by `mpiimpl.h` (used to build the implementation of the MPI routines) should be *copied* into `mpich/src/include` as part of the device's configure step. Provide the files `mpidpre.h` and `mpidpost.h`. The implementation of all MPI routines include files in this order:

   mpi.h The standard `mpi.h` that all MPI users include

   mpidpre.h Any definitions needed *before* the provided definitions of the contents of the internal structures. This can included definitions that override parts of `mpiimpl.h`

   contents of mpiimpl.h The bulk of the internal definitions. This also includes information on the timers.

   mpidpost.h Any definitions needed by the device after the rest of the definitions in `mpiimpl.h`. In many cases, this file may be empty.

   All of these are included by the file `mpiimpl.h`.

6.2 Directory Structure

A device should be placed in a subdirectory of `mpich/src/mpid/`; for example, `mpich/src/mpid/mm` is the multi-method ADI delivered with `mpich`. The directory name is the same as the device name specified to the `mpich configure` with the `--with-device` option.

6.3 Device Configuration and Setup

Each device must have a `configure` script. This will be run by the `mpich configure` as part of the top-level configuration. Any other commands that a device needs for setup should be run using the `AC_OUTPUT_COMMANDS autoconf` macro. Autoconf version 2.13 or later, but before 2.50, should be used; we recommend using the macros defined in `mpich/confdb/aclocal.m4`, as they include fixes to `autoconf`[5] Do not modify the `mpich configure` to support a device.

There must be a `echomaxprocname` target in the `Makefile` in the device's directory. This should look something like

```
echomaxprocname:
       @echo 128
```

This value will be used as the value for `MPI_MAX_PROCESSOR_NAME`, and must be an integer value.

A Data Structures

MPID_Request_kind — Kinds of MPI Requests

Synopsis

```
typedef enum { MPID_REQ_SEND, MPID_REQ_RECV, MPID_REQ_PERSISTENT_SEND,
               MPID_REQ_PERSISTENT_RECV, MPID_REQ_USER } MPID_Request_kind;
```

Module

Request-DS

MPID_Request — Description of the Request data structure

Synopsis

```
typedef struct {
    int          handle;  /* Value of MPI_Request for this structure */
    volatile int ref_count;
    MPID_Request_kind kind; /* Kind of request */
    /* The various types of requests may define subclasses for each
       kind.  In particular, the user and persistent requests need
       special information */
    volatile int cc;      /* Completion counter */
    int volatile *cc_ptr;
    /* Pointer to the completion counter associated with a group of requests.
```

[5] There are serious bugs in the `AC_CHECK_HEADER` macro that are still present in `autoconf` 2.52.

```
       When the counter reaches zero, the group of requests is complete. */
    MPI_Status status;
    MPID_Segment segment;
    MPID_Comm    *comm;   /* Originating Comm; needed to find error
                             handler if necessary */
    /* other, device-specific information */
} MPID_Request;
```

Module

Request-DS

Notes

If it is necessary to remember the MPI datatype, this information is saved within the `segment`, not as part of a separate `MPID_Datatype` entry.
Requests come in many flavors, as stored in the `kind` field. It is expected that each kind of request will have its own structure type (e.g., `MPID_Request_send_t`) that extends the `MPID_Request`.

MPID_Comm — Description of the Communicator data structure

Synopsis

```
typedef struct {
    int           handle;        /* value of MPI_Comm for this structure */
    volatile int ref_count;
    int16_t       context_id;    /* Assigned context id */
    int           remote_size;   /* Value of MPI_Comm_(remote)_size */
    int           local_size;    /* Value of MPI_Comm_size */
    int           rank;          /* Value of MPI_Comm_rank */
    MPID_VC *(*virtural connection)[]; /* Virtual connection table */
    MPID_Comm_kind_t comm_kind;  /* MPID_INTRACOMM or MPID_INTERCOMM */
    MPID_List     attributes;    /* List of attributes */
    MPID_Group    *local_group,  /* Groups in communicator. */
                  *remote_group; /* The local and remote groups are the
                                  same for intra communicators */
    char          name[MPI_MAX_OBJECT_NAME];  /* Required for MPI-2 */
    MPID_Errhandler *errhandler;  /* Pointer to the error handler structure */
    struct MPID_Collops_struct  *coll_fns;    /* Pointer to a table of
                                              functions implementing the
                                              collective routines */
  /* other, device-specific information */
#ifdef MPID_DEV_COMM_DECL
    MPID_DEV_COMM_DECL
#endif
} MPID_Comm;
```

Notes

Note that the size and rank duplicate data in the groups that make up this communicator. These are used often enough that this optimization is valuable.
The virtual connection table is an explicit member of this structure. This contains the information used to contact a particular process, indexed by the rank relative to this communicator.
Groups are allocated lazily. That is, the group pointers may be null, created only when needed by a routine such as `MPI_Comm_group`. The local process ids needed to form the group are available within the virtual connection table. For intercommunicators, we may want to always have the groups. If not, we either need the `local_group` or we need a virtual connection table corresponding to the `local_group` (we may want this anyway to simplify the implementation of the intercommunicator collective routines).
The pointer to the structure containing pointers to the collective routines allows an implementation to replace each routine on a routine-by-routine basis. By default, this pointer is null, as are the pointers within the structure. If either pointer is null, the implementation uses the generic provided implementation. This choice, rather than initializing the table with pointers to all of the collective routines, is made to reduce the space used in the communicators and to eliminate the need to include the implementation of all collective routines in all MPI executables, even if the routines are not used.

Module

Communicator-DS

Question

For fault tolerance, do we want to have a standard field for communicator health? For example, ok, failure detected, all (live) members of failed communicator have acked.

MPID_Segment — Description of the Segment data structure

Synopsis

```
typedef struct {
    void *ptr;
    int  bytes;
    int  alloc_bytes;    /* For a receive buffer, this may > bytes */

    /* stuff to manage pack/unpack */
    MPID_Dataloop_stackelm loopstack[MPID_MAX_DATATYPE_DEPTH];
    int  cur_sp;   /* Current stack pointer when using loopinfo */
    int  valid_sp; /* maximum valid stack pointer.  This is used to
                      maintain information on the stack after it has
                      been placed there by following the datatype field
                      in a MPID_Dataloop for any type except struct */

    /* other, device-specific information */
} MPID_Segment;
```

Notes

This has no corresponding MPI object.
The dataloop stack works as follows (the actual code will of course optimize the access to the stack elements and eliminate, for example, the various array references):

```
cur_sp=valid_sp=0;
stackelm[cur_sp].loopinfo  = datatype->loopinfo;
stackelm[cur_sp].loopinfo.curcount = 0;
while (cur_sp >= 0) {
   if stackelm[cur_sp].loopinfo.kind is final then
      // final means simple, consisting of basic datatypes, such
      // as a vector datatype made up of bytes or doubles)
      process datatype (this uses loopinfo.kind to pick the correct
          code fragments; we may also include some alignment tests
          so that longer word moves may be used for short (e.g.,
          one or two word) blocks).
          We can also choose to stop and return here when, for example,
          we have filled an output buffer.
      cur_sp--;
   else if stackelm[cur_sp].curcount == stackelm[cur_sp].loopinfo.cm_t.count
       then {
       // We are done with the datatype.
       cur_sp--;
       }
   else {
      // need to push a datatype.  Two cases: struct or other
      if (stackelm[cur_sp].loopinfo.kind == struct_type) {
         stackelm[cur_sp+1].loopinfo =
         stackelm[cur_sp].loopinfo.s_t.dataloop[stackelm[cur_sp].curcount];
         }
      else {
         if (valid_sp <= cur_sp) {
             stackelm[cur_sp+1].loopinfo =
             stackelm[cur_sp].loopinfo.cm_t.dataloop;
             valid_sp = cur_sp + 1;
         }
      }
      stackelm[cur_sp].curcount++;
      cur_sp ++;
      stackelm[cur_sp].curcount = 0;
   }
}
```

This may look a little bit messy (and not all of the code is here), but it shows how `valid_sp` is used to avoid recopying the information contained in a dataloop in the non-struct case. For example, a vector of vector has the dataloop description read only once, not once for each count of the outer vector.
Note that a relatively small value of `MPID_MAX_DATATYPE_DEPTH`, such as `16`, will allow the processing of extremely complex datatypes that describe huge amounts of memory. Thus, an `MPID_Segment` is a small object.

Module

Segment-DS

Questions

Should this have an id for allocation and similarity purposes?
Do we really need to specify the contents of the `MPID_Segment` structure?
Should we allow the `MPID_Dataloop_stackelm` to be a pointer, so that we can allocate a larger stack?

MPID_Dataloop — Description of the structure used to hold a datatype description

Synopsis

```
typedef struct dataloop_ {
    int kind;                   /* Contains both the loop type
                                (contig, vector, blockindexed, indexed,
                                or struct), a bit that indicates
                                whether the datatype is a leaf type, and
                                the natural element size */
    union {
        int                         count;
        MPID_Dataloop_contig        c_t;
        MPID_Dataloop_vector        v_t;
        MPID_Dataloop_blockindexed  bi_t;
        MPID_Dataloop_indexed       i_t;
        MPID_Dataloop_struct        s_t;
    } loop_params;
    MPI_Aint extent;
    int handle;                   /* Having the id here allows us to find the
                                  full datatype structure from the
                                  Dataloop description */
} MPID_Dataloop;
```

Fields

kind Indicates what type of datatype. May have the value `MPID_CONTIG`, `MPID_VECTOR`, `MPID_BLOCKINDEXED`, `MPID_INDEXED`, or `MPID_STRUCT`.

loop_parms A union containing the 5 dataloop structures, e.g., `MPID_Dataloop_contig`, `MPID_Dataloop_vector`, etc. A sixth element in this union, `count`, allows quick access to the shared `count` field in the five dataloop structure.

extent The extent of the datatype

id id for the corresponding `MPI_Datatype`.

Module

Datatype-DS

MPID_Dataloop_contig — Description of a contiguous datatype

Synopsis

```
typedef struct {
    int count;
    struct dataloop_ *dataloop;
} MPID_Dataloop_contig;
```

Fields

count Number of elements
datatype datatype that this datatype consists of

Notes

`Count` may be in terms of the number of elements stored in the dataloop `kind` field, particularly for leaf dataloops.

Module

Datatype-DS

MPID_Dataloop_vector — Description of a vector or strided datatype

Synopsis

```
typedef struct {
    int      count;
    struct dataloop_ *dataloop;
    int      blocksize;
    MPI_Aint stride;
} MPID_Dataloop_vector;
```

Fields

count Number of blocks
blocksize Number of datatypes in each block
stride Stride (in elements) between each block
datatype Datatype of each element

Notes

`Stride` may be in terms of the number of elements stored in the dataloop `kind` field, particularly for leaf dataloops.

Module

Datatype-DS

MPID_Dataloop_blockindexed — Description of a block-indexed datatype

Synopsis

```
typedef struct {
    int      count;
    struct dataloop_ *dataloop;
    int      blocksize;
    MPI_Aint *offset;
} MPID_Dataloop_blockindexed;
```

Fields

count	Number of blocks
blocksize	Number of elements in each block
offset	Array of offsets (in elements) to each block
datatype	Datatype of each element

Notes

`Offset` and `blocksize` may be in terms of the number of elements stored in the dataloop `kind` field, particularly for leaf dataloops.

Module

Datatype-DS

MPID_Dataloop_indexed — Description of an indexed datatype

Synopsis

```
typedef struct {
    int      count;
    struct dataloop_ *dataloop;
    int      *blocksize;
    MPI_Aint *offset;
} MPID_Dataloop_indexed;
```

Fields

count	Number of blocks
blocksize	Array giving the number of elements in each block
offset	Array of offsets (in elements) to each block

datatype Datatype of each element

Notes

`Offset` and `blocksize` may be in terms of the number of elements stored in the dataloop `kind` field, particularly for leaf dataloops.

Module

Datatype-DS

MPID_Dataloop_struct — Description of a structure datatype

Synopsis

```
typedef struct {
    int      count;
    struct dataloop_ *dataloop;
    int      *blocksize;
    MPI_Aint *offset;
} MPID_Dataloop_struct;
```

Fields

count Number of blocks
blocksize Array giving the number of elements in each block
offset Array of offsets (in elements) to each block
datatype Array of datatypes describing the elements of each block

Notes

`Blocksize` and `offset` may be in terms of the number of elements stored in the dataloop `kind` field, particularly for leaf dataloops.

Module

Datatype-DS

MPID_Datatype — Description of the MPID Datatype structure

Synopsis

```
typedef struct {
    int           handle;        /* value of MPI_Datatype for this structure */
    volatile int  ref_count;
    int           is_contig;     /* True if data is contiguous (even with
                                    a (count,datatype) pair) */
    MPID_Dataloop loopinfo;      /* Describes the arguments that the
```

```
                                   user provided for creating the datatype;
                                these are used to implement the
                                MPI-2 MPI_Type_get_contents functions */
    int           size;
    MPI_Aint      extent;        /* MPI-2 allows a type to created by
                                resizing (the extent of) an existing
                                type.  Note that the extent of the
                                datatype may be different from the
                                extent information in loopinfo */

    /* The remaining fields are required but less frequently used, and
       are placed after the more commonly used fields */
    int           has_sticky_ub;   /* The MPI_UB and MPI_LB are sticky */
    int           has_sticky_lb;
    int           is_permanent;  /* e.g., MPI_DOUBLE*/
    int           is_committed;  /* See MPID_Datatype_commit */

    int           loopinfo_depth; /* Depth of dataloop stack needed
                                    to process this datatype.  This
                                    information is used to ensure that
                                    no datatype is constructed that
                                    cannot be processed (see MPID_Segment) */

    MPI_Aint      ub, lb,        /* MPI-1 upper and lower bounds */
                  true_ub, true_lb; /* MPI-2 true upper and lower bounds */
    MPID_List     attributes;    /* MPI-2 adds datatype attributes */

    int32_t       cache_id;      /* These are used to track which processes */
    MPID_Lpidmask mask;          /* have cached values of this datatype */

    char          name[MPI_MAX_OBJECT_NAME];  /* Required for MPI-2 */

  /* other, device-specific information */
} MPID_Datatype;
```

Notes

The `ref_count` is needed for nonblocking operations such as

```
MPI_Type_struct( ... , &newtype );
MPI_Irecv( buf, 1000, newtype, ..., &request );
MPI_Type_free( &newtype );
...
MPI_Wait( &request, &status );
```

Module

Datatype-DS

Notes

Alternatives

The following alternatives for the layout of this structure were considered. Most were not chosen because any benefit in performance or memory efficiency was outweighed by the added complexity of the implementation.
A number of fields contain only boolean inforation (`is_contig`, `has_sticky_ub`, `has_sticky_lb`, `is_permanent`, `is_committed`). These could be combined and stored in a single bit vector.
`MPI_Type_dup` could be implemented with a shallow copy, where most of the data fields, particularly the `opt_dataloop` field, would not be copied into the new object created by `MPI_Type_dup`; instead, the new object could point to the data fields in the old object. However, this requires more code to make sure that fields are found in the correct objects and that deleting the old object doesn`t invalidate the dup`ed datatype.
A related optimization would point to the `opt_dataloop` and `dataloop` fields in other datatypes. This has the same problems as the shallow copy implementation.
In addition to the separate `dataloop` and `opt_dataloop` fields, we could in addition have a separate `hetero_dataloop` optimized for heterogeneous communication for systems with different data representations.
Earlier versions of the ADI used a single API to change the `ref_count`, with each MPI object type having a separate routine. Since reference count changes are always up or down one, and since all MPI objects are defined to have the `ref_count` field in the same place, the current ADI3 API uses two routines, `MPIU_Object_add_ref` and `MPIU_Object_release_ref`, to increment and decrement the reference count.

B Basic Point-to-Point

MPID_Send — MPID entry point for MPI_Send

Synopsis

```
int MPID_Send( const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPID_Comm *comm, int context_offset,
               MPID_Request **request )
```

Notes

The only difference between this and `MPI_Send` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, a context id offset is provided in addition to the communicator, and a request may be returned. The context offset is added to the context of the communicator to get the context it used by the message. A request is returned only if the ADI implementation was unable to complete the send of the message. In that case, the usual `MPI_Wait` logic should be used to complete the request. This approach is used to allow a simple implementation of the ADI. The ADI is free to always complete the message and never return a request.

Module

Communication

MPID_Ssend — MPID entry point for MPI_Ssend

Synopsis

```
int MPID_Ssend( const void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPID_Comm *comm, int context_offset,
                MPID_Request **request )
```

Notes

The only difference between this and `MPI_Ssend` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, a context id offset is provided in addition to the communicator, and a request may be returned. The context offset is added to the context of the communicator to get the context it used by the message. A request is returned only if the ADI implementation was unable to complete the send of the message. In that case, the usual `MPI_Wait` logic should be used to complete the request. This approach is used to allow a simple implementation of the ADI. The ADI is free to always complete the message and never return a request.

Module

Communication

MPID_Rsend — MPID entry point for MPI_Rsend

Synopsis

```
int MPID_Rsend( const void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPID_Comm *comm, int context_offset,
                MPID_Request **request )
```

Notes

The only difference between this and `MPI_Rsend` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, a context id offset is provided in addition to the communicator, and a request may be returned. The context offset is added to the context of the communicator to get the context it used by the message. A request is returned only if the ADI implementation was unable to complete the send of the message. In that case, the usual `MPI_Wait` logic should be used to complete the request. This approach is used to allow a simple implementation of the ADI. The ADI is free to always complete the message and never return a request.

Module

Communication

MPID_Isend — MPID entry point for MPI_Isend

Synopsis

```
int MPID_Isend( const void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPID_Comm *comm, int context_offset,
                MPID_Request **request )
```

Notes

The only difference between this and `MPI_Isend` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Issend — MPID entry point for MPI_Issend

Synopsis

```
int MPID_Issend( const void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPID_Comm *comm, int context_offset,
                MPID_Request **request )
```

Notes

The only difference between this and `MPI_Issend` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Irsend — MPID entry point for MPI_Irsend

Synopsis

```
int MPID_Irsend( const void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPID_Comm *comm, int context_offset,
                MPID_Request **request )
```

Notes

The only difference between this and `MPI_Irsend` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_tBsend — Attempt a send and return if it would block

Synopsis

```
int MPID_tBsend( const void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPID_Comm *comm, int context_offset )
```

Notes

This has the semantics of `MPI_Bsend`, except that it returns the internal error code `MPID_WOULD_BLOCK` if the message can't be sent immediately (t is for "try").
The reason that this interface is chosen over a query to check whether a message *can* be sent is that the query approach is not thread-safe. Since the decision on whether a message can be sent without blocking depends (among other things) on the state of flow control managed by the device, this approach also gives the device the greatest freedom in implementing flow control. In particular, if another MPI process can change the flow control parameters, then even in a single-threaded implementation, it would not be safe to return, for example, a message size that could be sent with `MPI_Bsend`.
This routine allows an MPI implementation to optimize `MPI_Bsend` for the case when the message can be delivered without blocking the calling process. An ADI implementation is free to have this routine always return `MPID_WOULD_BLOCK`, but is encouraged not to.
To allow the MPI implementation to avoid trying this routine when it is not implemented by the ADI, the C preprocessor constant `MPID_HAS_TBSEND` should be defined if this routine has a nontrivial implementation.

Module

Communication

MPID_Recv — MPID entry point for MPI_Recv

Synopsis

```
int MPID_Recv( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPID_Comm *comm, int context_offset,
               MPI_Status *status, MPID_Request **request )
```

Notes

The only difference between this and `MPI_Recv` is that the basic error checks (e.g., valid communicator, datatype, source, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, a context id offset is provided in addition to the communicator, and a request may be returned. The context offset is added to the context of the communicator to get the context it used by the message. As in `MPID_Send`, the request is returned only if the operation did not complete. Conversely, the status object is populated with valid information only if the operation completed.

Module

Communication

MPID_Irecv — MPID entry point for MPI_Irecv

Synopsis

```
int MPID_Irecv( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPID_Comm *comm, int context_offset,
               MPID_Request **request )
```

Notes

The only difference between this and `MPI_Irecv` is that the basic error checks (e.g., valid communicator, datatype, source, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Request_release — Release a request

Synopsis

```
void MPID_Request_release( MPID_Request *request )
```

Input Parameter

request request to release

Notes

This routine is called to release a reference to request object. If the reference count of the request object has reached zero, the object will be deallocated.

Module

Request

MPID_Cancel_send — Cancel the indicated send request

Synopsis

```
void MPID_Cancel_send( MPID_Request *request )
```

Input Parameter

request Send request to cancel

Notes

Cancel is a tricky operation, particularly for sends. Read the discussion in the MPI-1 and MPI-2 documents carefully. This call only requests that the request be cancelled; a subsequent wait or test must first succeed (i.e., the request completion counter must be zeroed).

Module

Request

MPID_Cancel_recv — Cancel the indicated recv request

Synopsis

```
void MPID_Cancel_recv( MPID_Request *request )
```

Input Parameter

request Receive request to cancel

Notes

This cancels a pending receive request. In many cases, this is implemented by simply removing the request from a pending receive request queue. However, some ADI implementations may maintain these queues in special places, such as within a NIC (Network Interface Card). This call only requests that the request be cancelled; a subsequent wait or test must first succeed (i.e., the request completion counter must be zeroed).

Module

Request

MPID_Iprobe — Look for a matching request in the receive queue but do not remove or return it

Synopsis

```
int MPID_Iprobe( int source, int tag, MPID_Comm *comm, int context_offset,
                 int * flag, MPI_Status *status )
```

Input Parameters

source	rank to match (or `MPI_ANY_SOURCE`)
tag	Tag to match (or `MPI_ANY_TAG`)
comm	communicator to match.
context_offset	context id offset of communicator to match

Output Parameter

flag	true if a matching request was found, false otherwise.
status	`MPI_Status` set as defined by `MPI_Iprobe` (only valid when return flag is true).

Return Value

Error Code.

Notes

Note that the values returned in `status` will be valid for a subsequent MPI receive operation only if no other thread attempts to receive the same message. (See the discussion of probe in Section 8.7.2 (Clarifications) of the MPI-2 standard.)
Providing the `context_offset` is necessary at this level to support the way in which the MPICH implementation uses context ids in the implementation of other operations. The communicator is present to allow the device to use message-queues attached to particular communicators or connections between processes.
Devices that rely solely on polling to make progress should call MPID_Progress_poke() (or some equivalent function) if a matching request could not be found. This insures that progress continues to be made even if the application is calling MPI_Iprobe() from within a loop not containing calls to any other MPI functions.

Module

Request

MPID_Probe — Block until a matching request is found and return information about it

Synopsis

```
int MPID_probe( int source, int tag, MPID_Comm *comm,
                int context_offset, MPI_Status *status )
```

Input Parameters

source	rank to match (or `MPI_ANY_SOURCE`)
tag	Tag to match (or `MPI_ANY_TAG`)
comm	communicator to match.
context_offset	context id offset of communicator to match

Output Parameter

status	`MPI_Status` set as defined by `MPI_Probe`

Return Value

Error code.

Notes

Note that the values returned in `status` will be valid for a subsequent MPI receive operation only if no other thread attempts to receive the same message. (See the discussion of probe in Section 8.7.2 Clarifications of the MPI-2 standard.)
Providing the `context_offset` is necessary at this level to support the way in which the MPICH implementation uses context ids in the implementation of other operations. The communicator is present to allow the device to use message-queues attached to particular communicators or connections between processes.

Module

Request

C Persistent Point-to-Point

MPID_Send_init — MPID entry point for MPI_Send_init

Synopsis

```
int MPID_Send_init( const void *buf, int count, MPI_Datatype datatype,
                    int dest, int tag, MPID_Comm *comm, int context_offset,
                    MPID_Request **request )
```

Notes

The only difference between this and `MPI_Send_init` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Ssend_init — MPID entry point for MPI_Ssend_init

Synopsis

```
int MPID_Ssend_init( const void *buf, int count, MPI_Datatype datatype,
                     int dest, int tag, MPID_Comm *comm, int context_offset,
                     MPID_Request **request )
```

Notes

The only difference between this and `MPI_Ssend_init` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Rsend_init — MPID entry point for MPI_Rsend_init

Synopsis

```
int MPID_Rsend_init( const void *buf, int count, MPI_Datatype datatype,
                     int dest, int tag, MPID_Comm *comm, int context_offset,
                     MPID_Request **request )
```

Notes

The only difference between this and `MPI_Rsend_init` is that the basic error checks (e.g., valid communicator, datatype, dest, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Recv_init — MPID entry point for MPI_Recv_init

Synopsis

```
int MPID_Recv_init( void *buf, int count, MPI_Datatype datatype,
                    int source, int tag, MPID_Comm *comm, int context_offset,
                    MPID_Request **request )
```

Notes

The only difference between this and `MPI_Recv_init` is that the basic error checks (e.g., valid communicator, datatype, source, and tag) have been made, the MPI opaque objects have been replaced by MPID objects, and a context id offset is provided in addition to the communicator. This offset is added to the context of the communicator to get the context it used by the message.

Module

Communication

MPID_Startall — MPID entry point for MPI_Startall

Synopsis

```
int MPID_Startall( int count, MPID_Request requests[] )
```

Notes

The only difference between this and `MPI_Startall` is that the basic error checks (e.g., count) have been made, and the MPI opaque objects have been replaced by pointers to MPID objects.

Rationale

This allows the device to schedule communication involving multiple requests, whereas an implementation built on just `MPID_Start` would force the ADI to initiate the communication in the order encountered.

D Generalized Requests

MPID_Request_create — Create and return a bare request

Synopsis

```
MPID_Request *MPID_Request_create( void )
```

Return value

A pointer to a new request object.

Notes

This routine is intended for use by `MPI_Grequest_start` only. Note that once a request is created with this routine, any progress engine must assume that an outside function can complete a request with `MPID_Request_set_completed`.
The request object returned by this routine should be initialized such that ref_count is one and handle contains a valid handle referring to the object.

MPID_Request_set_completed — Mark a request as completed

Synopsis

```
void MPID_Request_set_completed( MPID_Request *request )
```

Input Parameter

request_ptr Pointer to request to mark as completed.

Notes

This routine is intended for use by `MPI_Grequest_complete` only. It is provided to ensure that any MPI routine that is blocked waiting for a request to complete will be unblocked.

E Data Segment

MPID_Segment_init_pack — Get a segment ready for packing and sending (put, Rhcv, and/or stream) data

Synopsis

```
int MPID_Segment_init_pack( const void *buf, int count, MPID_Datatype *dtype,
                            MPID_Comm *comm, int rank,
                            MPID_Segment *segment )
```

Input Parameters

buf	Buffer to setup for sending
count	Number of items
dtype	Datatype of items
comm	Communicator for communication
rank	Rank of target. Use `MPI_ANY_SOURCE` for any member of the communicator

Output Parameter

segment	Segment descriptor.

Notes

Initializes a `MPID_Segment` from a specified user buffer which is described by `(buf,count,datatype)` and hence may represent noncontiguous data or contiguous data in a heterogeneous system.

Module

Segment

Questions

Add an *intent* so that sending all of the data can use device knowledge about complex data layouts? E.g., `intent == MPID_BUFFER_ALL` or `intent == MPID_BUFFER_SEGMENT`?
The original description contained this text:
If null, the original buffer is contiguous and can be used as is.
If `buf_desc` is non-null, the routine `MPID_Segment_pack` must be used to fill a memory location or return a pointer to a contiguous buffer.
However, the code is somewhat simplified if a segment is always used (in an case where a segment *may* be used).

MPID_Segment_pack — Pack up the designated buffer with a range of bytes.

Synopsis

```
void *MPID_Segment_pack( MPID_Segment *segment, int *first, int *last,
                         void *send_buffer )
```

Input Parameters

segment	Segment descriptor (initialized with `MPID_Segment_init_pack`)
send_buffer	Pointer to buffer to place data in. May be `NULL`, in which case `MPID_Segment_pack` uses an internal buffer (and returns it).

Inout Parameters

first	Pointer to first byte index to be packed (on input) or actually packed (on output)
last	Pointer to last byte index to be packed (on input) or actually packed (on output)

Return value

Pointer to memory containing packed data.

Notes

Providing a send buffer allows implementations to make use of double buffering without requiring that the segment code keep track of multiple buffers.

The values of `first` and `last` that are returned must satisfy the following restrictions:

1	(last-first) on output must be less than or equal to (last-first) on input.
2	(last+1) on output must have the property that when used as first on a subsequent input, the value is unchanged.
3	If first is zero on input, then first must be 0 on output.

In addition, the *change* in first should be as small as possible. For example, moving first so as to align the first byte on a word or cache-line boundary (subject to the 3 requirements above) is allowed (and is the reason to make first an in/out parameter). However, as rule three requires, if first is zero on input, no change to first is allowed even for alignment purposes. However, it is permissible to require that the buffers satisfy language- or processor-specified alignment rules.

Module

Segment

Questions

If `MPID_Segment_pack` returns an internal buffer, who is responsible for freeing it? It can't be a single static buffer because that isn't thread-safe. Is it freed as part of `MPID_Segment_free`?

If a send buffer is going to be provided, do we want to tell `MPID_Segment_init_pack` so that no internal buffer is allocated?

Should we return the number of bytes packed in a variable, just to simplify the use of the result (which, after all, is a pointer *and* a length)?

MPID_Segment_init_unpack — Get a segment ready for receiving and unpacking data

Synopsis

```
void * MPID_Segment_init_unpack( const void *buf, int maxcount,
                                 MPID_Datatype *dtype,
                                    MPID_Comm *comm, int rank,
                                    MPID_Segment *segment )
```

Input Parameters

buf	Buffer to setup for receiving
maxcount	Number of items
dtype	Datatype of items
comm	Communicator for communication
rank	Rank of source. Use `MPI_ANY_SOURCE` for any member of the communicator

Output Parameter

segment	Segment descriptor.

Module

Segment

Questions

Should there be a flag indicating that the buffer may be used in a put operation (as part of a receive and forward operation)?
Do we need both an init pack and init unpack?

MPID_Segment_unpack — Unpack the designated buffer with a range of bytes

Synopsis

```
void *MPID_Segment_unpack( MPID_Segment *segment, int *first, int *last,
                           void *recv_buffer )
```

Input Parameters

segment	Segment descriptor (initialized with `MPID_Segment_init_unpack`)
recv_buffer	Pointer to buffer to place data in. May be `NULL`, in which case `MPID_Segment_unpack` uses an internal buffer (and returns it).

Inout Parameters

first	Pointer to first byte index to be unpacked (on input) or actually unpacked (on output)
last	Pointer to last byte index to be unpacked (on input) or actually unpacked (on output)

Return value

Pointer to memory containing unpacked data.

Module

Segment

MPID_Segment_free — Free a segment

Synopsis

```
int MPID_Segment_free( MPID_Segment *segment )
```

Input Parameter

segment Segment to free

Notes

Unlike the MPI routines to free objects, the MPID routines typically do not also set the object handle to `NULL`.

Question

Should we just use the generic object allocator and deallocator for segments (`MPIU_Handle_obj_destroy`)? Note that segments, like requests, are performance-critical.

Module

Segment

F Progress Engine

The progress engine provides a way for the MPI implementation and the ADI implementation to coordinate progress on communication. The design makes no assumptions about whether the ADI uses polling or a separate communication thread or interrupts or any other mechanism to make progress.

Here are several implementation sketches. First, a polling implementation for a single-threaded implementation:

```
MPID_Progress_start - no-op
MPID_Progress_end - no-op
MPID_Progress_test - select with no wait
MPID_Progress_wait - select with infinite wait
MPID_Progress_poke - select with no wait
```

A non-polling, single-threaded implementation that used a separate thread for communication could use (this isn't correct yet)

```
MPID_Progress_start - Set flag indicating checks in progress
MPID_Progress_end - Clear flag
MPID_Progress_test - yield to communication thread
```

```
MPID_Progress_wait - if no completions since flag set,
                     wait on condition variable.  Otherwise, return.
MPID_Progress_poke - either no-op or yield to communication thread
```

See the generalized request functions in Section D for some additional requirements on the progress engine.

MPID_Progress_start — Begin a block of operations that check the completion counters in requests.

Synopsis

```
void MPID_Progress_start( void )
```

Notes

This routine is used to inform the progress engine that a block of code will examine the completion counter of some `MPID_Request` objects and then call one of `MPID_Progress_end`, `MPID_Progress_wait`, or `MPID_Progress_test`.
This routine is needed to properly implement blocking tests when multithreaded progress engines are used. In a single-threaded implementation of the ADI, this may be defined as an empty macro.

Module

Communication

MPID_Progress_end — End a block of operations begun with MPID_Progress_start

Synopsis

```
void MPID_Progress_end( void )
```

Notes

This instructs the progress engine to end the block begun with `MPID_Progress_start`. The progress engine is not required to check for any pending communication.
The purpose of this call is to release any locks initiated by `MPID_Progess_start`. It is typically used when checks of the relevant request completion counters found a completed request. In a single threaded ADI implementation, this may be defined as an empty macro.

MPID_Progress_test — Check for communication since `MPID_Progress_start`

Synopsis

```
int MPID_Progress_test( void )
```

Return value

The number of communication actions since `MPID_Progress_start`.

Notes

Like `MPID_Progress_end` and `MPID_Progress_wait`, this completes the block begun with `MPID_Progress_start`. Unlike `MPID_Progress_wait`, it is a nonblocking call. It returns the number of communication events, which is only indicates the maximum number of separate requests that were completed. The only restriction is that if the completion status of any request changed between `MPID_Progress_start` and `MPID_Progress_test`, the return value must be at least one.

MPID_Progress_wait — Wait for some communication since `MPID_Progress_start`

Synopsis

```
void MPID_Progress_wait( void )
```

Notes

This instructs the progress engine to wait until some communication event happens since `MPID_Progress_start` was called. This call blocks the calling thread (only, not the process). Before returning, it releases the block begun with `MPID_Progress_start`.

MPID_Progress_poke — Allow a progress engine to check for pending communication

Synopsis

```
void MPID_Progress_poke( void )
```

Notes

This routine provides a way to invoke the progress engine in a polling implementation of the ADI. This routine must be nonblocking.
A multithreaded implementation is free to define this as an empty macro.

G Starting and stopping

MPID_Init — Initialize the device

Synopsis

```
int MPID_Init( int *argc_p, char *(*argv_p)[],
               int requested, int *provided,
               MPID_Comm **parent_comm, int *has_args, int *has_env )
```

Input Parameters

argc_p Pointer to the argument count
argv_p Pointer to the argument list
requested Requested level of thread support. Values are the same as for the `required` argument to `MPI_Init_thread`, except that we define an enum for these values.

Output Parameter

provided Provided level of thread support. May be less than the requested level of support.
parent_comm `MPID_Comm` of parent. This is null for all MPI-1 uses and for processes that are *not* started with `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`.
has_args Set to true if `argc_p` and `argv_p` contain the command line arguments. See below.
has_env Set to true if the environment of the process has been set as the user expects. See below.

Return value

Returns `0` on success and an MPI error code on failure. Failure can happen when, for example, the device is unable to start or contact the number of processes specified by the `mpiexec` command.

Notes

Null arguments for `argc_p` and `argv_p` *must* be valid (see MPI-2, section 4.2)
Multi-method devices should initialize each method within this call. They can use environment variables and/or command-line arguments to decide which methods to initialize (but note that they must not *depend* on using command-line arguments).
This call also initializes all MPID data needed by the device. This includes the `MPID_Request`s and any other data structures used by the device.
The arguments `has_args` and `has_env` indicate whether the process was started with command-line arguments or environment variables. In some cases, only the root process is started with these values; in others, the startup environment ensures that each process receives the command-line arguments and environment variables that the user expects. While the MPI standard makes no requirements that command line arguments or environment variables are provided to all processes, most users expect a common environment. These variables allow an MPI implementation (that is based on ADI-3) to provide both of these by making use of MPI communication after `MPID_Init` is called but before `MPI_Init` returns to the user.
This routine is used to implement both `MPI_Init` and `MPI_Init_thread`.
Setting the environment requires a `setenv` function. Some systems may not have this. In that case, the documentation must make clear that the environment may not be propagated to the generated processes.
The `parent_comm` argument may not be the right interface.

Module

MPID_CORE

Questions

The values for `has_args` and `has_env` are boolean. They could be more specific. For example, the value could indicate the rank in `MPI_COMM_WORLD` of a process that has the values; the value `MPI_ANY_SOURCE` (or a `-1`) could indicate that the value is available on all processes (including this one). We may want this since otherwise the processes may need to determine whether any process needs the command line. Another option would be to use positive values in the same way that the `color` argument is used in `MPI_Comm_split`; a negative value indicates the member of the processes with that color that has the values of the command line arguments (or environment). This allows for non-SPMD programs.

Do we require that the startup environment (e.g., whatever `mpiexec` is using to start processes) is responsible for delivering the command line arguments and environment variables that the user expects? That is, if the user is running an SPMD program, and expects each process to get the same command line argument, who is responsible for this? The `has_args` and `has_env` values are intended to allow the ADI to handle this while taking advantage of any support that the process manager framework may provide.

Can we fix the Fortran command-line arguments? That is, can we arrange for `iargc` and `getarg` (and the POSIX equivalents) to return the correct values? See, for example, the Absoft implementations of `getarg`. We could also contact PGI about the Portland Group compilers, and of course the `g77` source code is available. Does each process have the same values for the environment variables when this routine returns?

If we don't require that all processes get the same argument list, we need to find out if they did anyway so that `MPI_Init_thread` can fixup the list for the user. This argues for another return value that flags how much of the environment the `MPID_Init` routine set up so that the `MPI_Init_thread` call can provide the rest. The reason for this is that, even though the MPI standard does not require it, a user-friendly implementation should, in the SPMD mode, give each process the same environment and argument lists unless the user explicitly directed otherwise.

How does this interface to BNR? Do we need to know anything? Should this call have an info argument to support BNR?

The following questions involve how environment variables and command line arguments are used to control the behavior of the implementation. Many of these values must be determined at the time that `MPID_Init` is called. These all should be considered in the context of the parameter routines described in the MPICH2 Design Document.

Are there recommended environment variable names? For example, in ADI-2, there are many debugging options that are part of the common device. In MPI-2, we can't require command line arguments, so any such options must also have environment variables. E.g., `MPICH_ADI_DEBUG` or `MPICH_ADI_DB`.

Names that are explicitly prohibited? For example, do we want to reserve any names that `MPI_Init_thread` (as opposed to `MPID_Init`) might use?

How does information on command-line arguments and environment variables recognized by the device get added to the documentation?

What about control for other impact on the environment? For example, what signals should the device catch (e.g., `SIGFPE`? `SIGTRAP`?)? Which of these should be optional (e.g., ignore or leave signal alone) or selectable (e.g., port to listen on)? For example, catching `SIGTRAP` causes problems for `gdb`, so we'd like to be able to leave `SIGTRAP` unchanged in some cases.

Another environment variable should control whether fault-tolerance is desired. If fault-tolerance is selected, then some collective operations will need to use different algorithms and most fatal errors detected by the MPI implementation should abort only the affected process, not all processes.

MPID_Finalize — Perform the device-specific termination of an MPI job

Synopsis

```
int MPID_Finalize( void )
```

Return Value

`MPI_SUCCESS` or a valid MPI error code. Normally, this routine will return `MPI_SUCCESS`. Only in extrordinary circumstances can this routine fail; for example, if some process stops responding during the finalize step. In this case, `MPID_Finalize` should return an MPI error code indicating the reason that it failed.

Notes

Module

MPID_CORE

Questions

Need to check the MPI-2 requirements on `MPI_Finalize` with respect to things like which process must remain after `MPID_Finalize` is called.

MPID_Abort — Abort at least the processes in the specified communicator.

Synopsis

```
int MPID_Abort( MPID_Comm *comm, int return_code )
```

Input Parameters

comm	Communicator of processes to abort
return_code	Return code to return to the calling environment. See notes.

Return value

`MPI_SUCCESS` or an MPI error code. Normally, this routine should not return, since the calling process must be a member of the communicator. However, under some circumstances, the `MPID_Abort` might fail; in this case, returning an error indication is appropriate.

Notes

In a fault-tolerant MPI implementation, this operation should abort *only* the processes in the specified communicator. Any communicator that shares processes with the aborted communicator becomes invalid. For more details, see (paper not yet written on fault-tolerant MPI).
In particular, if the communicator is `MPI_COMM_SELF`, only the calling process should be aborted.
The `return_code` is the return code that this particular process will attempt to provide to the `mpiexec` or other program invocation environment. See `mpiexec` for a discussion of how return codes from many processes may be combined.

An external agent that is aborting processes can invoke this with either `MPI_COMM_WORLD` or `MPI_COMM_SELF`. For example, if the process manager wishes to abort a group of processes, it should cause `MPID_Abort` to be invoked with `MPI_COMM_SELF` on each process in the group.

Question

An alternative design is to provide an `MPID_Group` instead of a communicator. This would allow a process manager to ask the ADI to kill an entire group of processes without needing a communicator. However, the implementation of `MPID_Abort` will either do this by communicating with other processes or by requesting the process manager to kill the processes. That brings up this question: should `MPID_Abort` use `BNR` to kill processes? Should it be required to notify the process manager? What about persistent resources (such as SYSV segments or forked processes)?
This suggests that for any persistent resource, an exit handler be defined. These would be executed by `MPID_Abort` or `MPID_Finalize`. See the implementation of `MPI_Finalize` for an example of exit callbacks. In addition, code that registered persistent resources could use persistent storage (i.e., a file) to record that information, allowing cleanup utilities (such as `mpiexec`) to remove any resources left after the process exits.
`MPI_Finalize` requires that attributes on `MPI_COMM_SELF` be deleted before anything else happens; this allows libraries to attach end-of-job actions to `MPI_Finalize`. It is valuable to have a similar capability on `MPI_Abort`, with the caveat that `MPI_Abort` may not guarantee that the run-on-abort routines were called. This provides a consistent way for the MPICH implementation to handle freeing any persistent resources. However, such callbacks must be limited since communication may not be possible once `MPI_Abort` is called. Further, any callbacks must guarantee that they have finite termination.
One possible extension would be to allow *users* to add actions to be run when `MPI_Abort` is called, perhaps through a special attribute value applied to `MPI_COMM_SELF`. Note that is is incorrect to call the delete functions for the normal attributes on `MPI_COMM_SELF` because MPI only specifies that those are run on `MPI_Finalize` (i.e., normal termination).

Module

MPID_CORE

H Information about the device

MPID_Get_processor_name — Return the name of the current processor

Synopsis

```
int MPID__Get_processor_name( char *name, int *resultlen)
```

Output Parameters

name	A unique specifier for the actual (as opposed to virtual) node. This must be an array of size at least `MPI_MAX_PROCESSOR_NAME`.
resultlen	Length (in characters) of the name

Notes

The name returned should identify a particular piece of hardware; the exact format is implementation defined. This name may or may not be the same as might be returned by `gethostname`, `uname`, or `sysinfo`.
This routine is essentially an MPID version of `MPI_Get_processor_name` . It must be part of the device because not all environments support calls to return the processor name.

I RMA

(not yet defined)

J Dynamic Processes

(not yet designed)

K Collective Communication

(not yet designed)

L Connections and Local Process Ids

These routines are used to manage connections. MPI Communicators contain references to these tables; group operations use the local process id (see `MPID_VCR_Get_lpid`) to identify processes.

MPID_VCRT_Create — Create a virtual connection reference table

Synopsis

```
int MPID_VCRT_Create(int size, MPID_VCRT *vcrt_ptr)
```

MPID_VCRT_Add_ref — Add a reference to a VCRT

Synopsis

```
int MPID_VCRT_Add_ref(MPID_VCRT vcrt)
```

MPID_VCRT_Release — Release a reference to a VCRT

Synopsis

```
int MPID_VCRT_Release(MPID_VCRT vcrt)
```

MPID_VCRT_Get_ptr —

Synopsis

```
int MPID_VCRT_Get_ptr(MPID_VCRT vcrt, MPID_VCR **vc_pptr)
```

MPID_VCR_Dup —

Synopsis

```
int MPID_VCR_Dup(MPID_VCR orig_vcr, MPID_VCR * new_vcr)
```

MPID_VCR_Get_lpid — Get the local process id that corresponds to a virtual connection reference.

Synopsis

```
int MPID_VCR_Get_lpid(MPID_VCR vcr, int * lpid_ptr)
```

Notes

The local process ids are described elsewhere. Basically, they are a nonnegative number by which this process can refer to other processes to which it is connected. These are local process ids because different processes may use different ids to identify the same target process

M Device Hooks

MPID_Dev_xxx_create_hook — Inform the device about the creation of xxx

Synopsis

```
void MPID_Dev_xxx_create_hook( MPID_xxx *obj )
```

Input Parameter

obj Pointer to the object that has been created

Notes

This is a pseudo routine. The device may define routines of this for for each MPID object (e.g., `MPID_Comm`, `MPID_Datatype`, `MPID_File`, etc.). The `xxx` is replaced with the name of the object; e.g., `comm`, `datatype`, `file`, etc.
This may be defined as an empty macro.
The exact parameter list may depend on the object. These parameter lists have not yet been defined. If a device defines either the create or destroy hooks, it must define both. In addition, the C preprocessor symbol `HAVE_DEV_xxx_HOOK` must be defined. If this is not defined, then `MPID_Dev_xxx_create_hook` and `MPID_Dev_xxx_destroy_hook` will be defined as empty macros.
See also `MPID_Dev_xxx_destroy_hook`

MPID_Dev_xxx_destroy_hook — Inform the device about the destruction of xxx

Synopsis

```
void MPID_Dev_xxx_create_hook( MPID_xxx *obj )
```

Input Parameter

obj Pointer to the object that has been created

Notes

This is a pseudo routine. The device may define routines of this for for each MPID object (e.g., `MPID_Comm`, `MPID_Datatype`, `MPID_File`, etc.). The `xxx` is replaced with the name of the object; e.g., `comm`, `datatype`, `file`, etc.
This may be defined as an empty macro.
See also `MPID_Dev_xxx_create_hook`

References

[1] James Cownie and William Gropp. A standard interface for debugger access to message queue information in MPI. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 51–58. Springer Verlag, 1999.

[2] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared memory machines. *ACM Transactions on Programming Languages and Systems*, 0(0):999–1025, January 2000.

Index