MMPI: Asynchronous Message Management for the Message-Passing Interface

Harold Carter Edwards
MMPI: Asynchronous Message Management for the Message-Passing Interface

Harold Carter Edwards
Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, Texas, 78712

TICAM Report 96-44

Abstract

The Message-Passing Interface (MPI) Standard defines an asynchronous message passing functionality through primitive nonblocking send and receive operations. Applications which use asynchronous message passing within MPI must assume responsibility for managing the MPI nonblocking send and receive primitives. For an application with dynamic interprocessor communication requirements, asynchronous message management is a complex task. The asynchronous message Management for MPI (MMPI) library described in this report has been developed to simplify dynamic asynchronous message passing with MPI.
1 Introduction

A family of parallel applications developed at the Texas Institute for Computational and Applied Mathematics (TICAM) use large irregular distributed data structures. Runtime modification and redistribution of these data structures occurs at various phases during the applications' execution. The message passing patterns associated with accessing these irregular, dynamic, and distributed data structures are complex and unpredictable.

One approach taken at TICAM to address these complex and unpredictable communications is to use asynchronous message passing. Asynchronous message passing has traditionally been used to overlap communications with computations; however, it also provides a flexible mechanism for managing complex and unpredictable message passing requirements. Very briefly, asynchronous message passing is used to perform communications which might occur, where each message is processed if and when it is received.

The Message-Passing Interface (MPI) defines primitive communication operations for asynchronous message passing. Using these primitive operations for overlapping communications and computation is straightforward; however, using these primitive operations to address complex and unpredictable message passing requirements is a formidable task. The objective of the MMPI library is to simplify this task with a set of MPI-like opaque data types and subprograms which are “layered on” MPI.

A very brief review of asynchronous message passing with MPI is given in the following section. However, it is assumed that the reader is already familiar with the Message-Passing Interface Standard, especially those sections regarding asynchronous message passing.

1.1 Asynchronous Messages Passing with MPI

The Message-Passing Interface defines primitive nonblocking send and receive operations (see Section 3.7 of the MPI Standard [3]). A simple MPI nonblocking communication operation is initiated with a start call and is terminated with a complete call. Other computations may occur between these MPI calls, thus overlapping communications and computations. A nonblocking operation which has been started but has not yet been completed is referred to as a “pending” or an “active” operation. The MPI specification does not include a lower or upper bound on the number of pending nonblocking operations; however, it is stated in the MPI specification that “an MPI implementation should be able to support a large number of pending nonblocking operations.”

An application with unpredictable communication requirements may initiate nonblocking receive operations for all messages which might occur within a particular context. These active nonblocking operations must be periodically tested for
completion. When such a pending nonblocking operation completes, the application performs some application specific processing of the received message. If several such nonblocking operations are active, the application must also be concerned with message progression rules, deadlock, and buffer overflow. A final element of complexity in this scheme is that the size and content of the message which might be received may vary. In summary, this approach of initiating nonblocking receives for messages which might occur must deal with the following issues:

- a dynamic set of pending nonblocking operations,
- periodic testing of pending nonblocking operations,
- matching completed messages with appropriate processing,
- message progression rules,
- messages of variable size and content,
- deadlocking, and
- buffer overflow.

Using MPI asynchronous message passing for complex and unpredictable communication can become unwieldy without asynchronous message management.

### 1.2 Management for MPI Asynchronous Messages

Each nonblocking communication operation started by an application may be posted to MMPI for subsequent completion. Each nonblocking operation is posted with an application supplied *handler*. A handler consists of a subprogram and an arbitrary set of data. Upon completion of the nonblocking operation MMPI calls the associated handler subprogram, with the handler data, to process the completed message.

An application may send multiple messages, of varying contents, to the same nonblocking receive operation. In such a scenario an MPI nonblocking receive operation must be restarted after each received message has been processed. Potential problems with this scenario include sending a message which is larger than the receiver's current buffer, or "flooding" the receiver with messages which are too numerous or too large to be buffered by the MPI implementation. The MMPI library addresses this communication scenario and its potential problems by providing an alternative to the MPI nonblocking receive, an MMPI *Consumer*.

An MMPI Consumer is an "open ended" receive operation where an arbitrary number of messages of arbitrary content and length may be received and processed by a single Consumer. Messages received by a Consumer are "consumed" by the application via an application provided handler. The application's consumer handler is responsible for interpreting the contents of each message and processing it accordingly. All other message passing responsibilities are assumed by MMPI so that the MPI implementation is never required to buffer numerous or large Consumer messages.
1.3 Related Message Management Capabilities

Other message passing libraries support handlers for asynchronous message passing. Two such libraries are:

- the NX library [1] and
- active messages [4].

This is by no means intended to be a comprehensive list of communication libraries with asynchronous message management.

The `hrecv` and `hrecvx` routines of the NX library post an asynchronous receive to the run-time environment with an attached handler. When the NX run-time environment matches a message with the asynchronous receive the application’s local thread of control is interrupted and the attached handler is invoked. The NX run-time environment allows an application to post several asynchronous receives via `hrecv` or `hrecvx`, and then “forget” about them until they are matched with a message. In contrast, an MPI based application must explicitly check each asynchronous receive operations to guarantee that a message has actually been received.

Active messages [4] also allow the application to associate routines with received messages. As with the NX run-time environment, the active message run-time environment automatically invokes the handler upon receipt of a message.

A similar interrupt-like “handler” functionality could be implemented with MPI and threads by 1) creating a thread for each asynchronous message and 2) performing a blocking receive within that thread for the message. Such an MPI+threads approach assumes the availability of threads and a thread safe MPI implementation, requires the application developer to be knowledgeable in mixing threads and message passing, and relies upon “fairness” of the threads run-time support. This MPI+threads approach is not used in the MMPI library.

The MPI-2 effort (see http://parallel.nas.nasa.gov/MPI-2/) has debated an interrupt-like “handler” functionality at some length. In the early stages of the MPI-2 effort this functionality appeared in the “mpi-lsided” group. One contribution to this debate was the TICAM report “A Consistent Extension of the Message-Passing Interface (MPI) for Nonblocking Communication Handlers” [2]. The handler discussion was subsequently moved to the “mpi-external” group of the MPI-2 effort where the integration of MPI, handlers, and threads are addressed. Note again that the MMPI library does not use threads, so the integration of handlers and threads is not an issue here.
1.4 MMPI Overview

A brief overview of the MMPI library is given in this section. The five components of the MMPI library are layered on MPI as shown here. The application programming interface for each component is given in the sections noted here.

<table>
<thead>
<tr>
<th>Component</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>5</td>
</tr>
<tr>
<td>Tag Mgt</td>
<td>2</td>
</tr>
<tr>
<td>Request Mgt.</td>
<td>3</td>
</tr>
<tr>
<td>Packed Buffer Mgt.</td>
<td>4</td>
</tr>
<tr>
<td>Log File Mgt.</td>
<td>6</td>
</tr>
<tr>
<td>MPI</td>
<td></td>
</tr>
</tbody>
</table>

The MMPI Consumer provides the “highest level” of asynchronous message management functionality, where the application need not be concerned with allocating buffers, managing MPI requests, or “flooding” an MPI implementation. Integration of an MMPI Consumer within a larger parallel application requires some familiarity with the underlying tag management, request management, and packed buffer management components of MMPI.

1.4.1 Tag Management

Each MPI message has an associated message identification tag. Incoming messages are matched to receive requests, in part, according to the tags of the receive requests. If a message matches two different posted nonblocking MPI receive requests it is unpredictable which receive will be matched with the message. Such tag conflicts can arise when an application integrates several software components which use non-blocking communications. To avoid such conflicts the MMPI library provides a tag management facility.

1.4.2 Request Management

The Message-Passing Interface Standard defines an asynchronous message passing capability with primitives for nonblocking point-to-point sends and receives (Section 3.7 of [3]). Within MPI each asynchronous communication operation is identified with a request, an MPI opaque data object. The MMPI library allows an application to associate a request handler with an MPI request. The MMPI request manager assumes responsibility for completing the MPI request’s send or receive operation and invoking the application’s associated request handler.

1.4.3 Packed Buffer Management

Messages with compound or varying contents are most easily managed through the MPI pack and unpack routines (Section 3.13 of [3]). Multiple calls to these MPI routines are used to pack and unpack compound messages. Multiple calls to the MPI pack and unpack routines require an adequately sized buffer (possibly dynamically
allocated and reallocated) and a consistent set of calling arguments. The MMPI library defines an opaque object type and supporting routines which manage both buffer allocation/reallocation and calling argument consistencies between multiple calls to the MPI pack and unpack routines.

1.5 Consumer

An MMPI Consumer is a high level communication object which:
- asynchronously receives an unlimited number of messages,
- manages its own message tags,
- manages its own message buffers, and
- supports a consumer handler routine.

In using an MMPI Consumer the application is only responsible for packing and unpacking the consumer messages. A consumer message is unpacked on the receiving processor by the application’s consumer handler routine. An application’s consumer handler is similar to the request handler in that it is invoked automatically for each received consumer message. Multiple MMPI Consumer data objects may be created where each consumer may be given its own independent consumer handler.

1.6 Log File Management

MMPI routines output warning and error messages to a processor specific log file. This log file is also available for an application’s text output as a
- C file descriptor (int),
- C file buffer (FILE *), or
- C++ output stream (ostream &).

The name of the log file can be selected by the application immediately after MPI initialization with MPI_Init. The application’s selected name for the log file is appended with the rank of the processor to insure that each processor outputs to a different log file.
2 Tag Management

The MMPI tag management facility manages an application specified range of tags for a given MPI Communicator. Tags may be reserved from a communicator’s set, used to identify a message, and then released. MMPI tag management guarantees a unique tag for given message if 1) the tag is reserved and 2) the reserved tag is used to identify a single send operation.

2.1 Environment

Tag management is enabled for a particular MPI Communicator with a call to MMPI_Enable or MMPI_Enable_tag. A call to MMPI_Enable calls MMPI_Enable_tag with the indicated default values for the tag_min and tag_max arguments.

    int MMPI_Enable( MPI_Comm comm ); /* IN OUT */

    int MMPI_Enable_tag(
        MPI_Comm comm ,      /* IN OUT default: */
        unsigned tag_min ,   /* IN 24576 (0x6000) */
        unsigned tag_max );  /* IN 32767 (0x7fff) */

The call to MMPI_Enable* must be performed on all processors of the given communicator and each processor must have identical values for tag_min and tag_max. The MMPI_Enable* routines also “set up” the MMPI_Barrier routine (see Section 3.3.2). Thus the call MMPI_Enable* must occur before the first call to MMPI_Barrier. It is recommended that MMPI_Enable or MMPI_Enable_tag be called 1) for the MPI_COMM_WORLD communicator immediately after calling MPI_Init and 2) for any other communicator immediately after its creation.

    int MMPI_Disable( MPI_Comm ); /* IN OUT */

MMPI tag management is disabled for a particular communicator with a call to MMPI_Disable or when the communicator is destroyed via MPI_Comm_free. This call also “cleans up” the MMPI_Barrier routine.

During development and debugging of an application the set of global tags may become inconsistent. The application programmer may test for this error through a synchronous call to MMPI_Tag_verify.

    int MMPI_Tag_verify( MPI_Comm comm );

This routine broadcasts (MMPI_Bcast) the global tag table from processor #0 and compares the broadcast global tag table with the local tag table. If the two lists do not compare MPI_ERR_COMM is returned on that processor. Note that MMPI_Tag_verify is called by MMPI_Init_tag to insurne consistent tag management activation.
2.2 Global Tag Management

Reservation of a tag with the same value on all processors is a synchronous operation. The MMPI_Tag_get_global routine, which reserves a globally consistent tag value, and the MMPI_Tag_rel_global routine, which releases a reserved global tag value, must be called synchronously on all processors.

```c
int MMPI_Tag_get_global(
    MPI_Comm comm, /* IN OUT */
    int * tag ); /* OUT */

int MMPI_Tag_rel_global(
    MPI_Comm comm, /* IN OUT */
    int * tag ); /* OUT */
```

2.3 Local Tag Management

In some cases a tag does not need to be globally consistent. For these cases the MMPI_Tag_get_local and MMPI_Tag_rel_local routines have been provided.

```c
int MMPI_Tag_get_local(
    MPI_Comm comm, /* IN OUT */
    int * tag ); /* OUT */

int MMPI_Tag_rel_local(
    MPI_Comm comm, /* IN OUT */
    int * tag ); /* OUT */
```
3 Request Management

MPI nonblocking communications (Section 3.7 of [3]) support asynchronous message passing. Each MPI asynchronous message requires two actions: 1) start the asynchronous message request (send or receive) and 2) complete the request. An asynchronous message request which has been started but has not yet completed is termed an active request. Completion of active request is typically followed by an application specific processing of the message contents.

The MMPI library provides a facility for managing MPI active requests. This request management facility allows an application to start an asynchronous messages and then give the MMPI request manager responsibility for completing the active requests. Upon completion of an application’s active request the MMPI request manager invokes an application specified request handler which processes the message contents.

Each request handler consists of an application supplied routine and data object. The value of the request handler data object is arbitrary. The request handler routine may access the application’s data, call MPI routines, or initiate new asynchronous message. A request handler routine may also reactivate its associated request. When a request handler routine reactivates its own request the request and its handler become “self sustaining”.

3.1 Request Handler Specification

A request handler routine’s interface must conform to the following C type definition.

```c
typedef int MMPI_Request_handler(
    void * handler_data , /* IN OUT */
    MPI_Request * request , /* IN OUT */
    MPI_Status * status );/* IN */
```

The request handler data object is a pointer to an arbitrary data object, which may be NULL.

The `request` is the application’s active request which is completed by the request manager. The `status` is the request completion status generated by MPI. The request handler routine returns either MPI_SUCCESS or an MPI error code. Other interactions between an application’s request handler routine and the request manager are discussed in Section 3.3.4.

3.2 Posting Request Handlers

An active asynchronous message request is posted to the request manager with a request handler.
int MMPLPost_handler(
    MPLRequest request ,    /* IN */
    void * handler_data ,    /* IN */
    MMPIRequest_handler handler_routine );    /* IN */

The handler_routine is called by the request manager when the request completes. The handler_data is given to the handler_routine when it is called by the request manager. The request input to the MMPLPost_handler routine is copied by the request manager, this copy of the request is subsequently given to the handler_routine.

A request may be unposted from the request manager by calling MMPIPost_handler with a handler_routine value of MMPI_REQUEST_HANDLER_NULL.

### 3.3 Request Completion

Requests posted to the request manager may be completed only during calls to any of four MMPI routines:

- MMPI_Test
- MMPI_Wait
- MMPI_Barrier
- MMPI_Serve_handler

If at least one of these routines is not called by the application then posted requests cannot be completed and the application may deadlock.

#### 3.3.1 Test and Wait

The MMPI_Test and MMPI_Wait routines are “replacements” for the standard MPI_Wait and MPI_Test routines.

```c
int MMPI_Wait(
    MPI_Request * request ,    /* IN OUT */
    MPI_Status * status );    /* OUT */

int MMPI_Test(
    MPI_Request * request ,    /* IN OUT */
    int * flag ,    /* OUT */
    MPI_Status * status );    /* OUT */
```

The arguments to these routines are identical to the replaced MPI_Wait and MPI_Test routines. However, if the input request is persistent, has been posted to the request manager, and is “self sustaining” the request may complete more than once before the MMPI_Wait or MMPI_Test returns. In this situation the output status reflects the first completion.
Once posted to the request manager the request should not be tested or waited for with the standard MPI\_Wait and MPI\_Test family of routines. The results of such a call leaves the request manager with an inconsistent set of requests, which is likely to yield erroneous results. If it is absolutely necessary to call one of the MPI\_Wait and MPI\_Test routines for an active request which has been posted with MMPI request manager, the request must first be unposted from the request manager.

3.3.2 Barrier

The MMPI\_Barrier routine is a “replacement” for the MPI\_Barrier routine.

```c
int MMPI\_Barrier( MPI\_Comm comm );
```

A call to MMPI\_Barrier accomplishes the same synchronization as a call to MPI\_Barrier; however, during synchronization the request manager may complete posted requests.

Recall that the MMPI\_Barrier routine is “set up” by a call to MMPI\_Enable or MMPI\_Enable\_tag. Thus one of the MMPI\_Enable* routines must be called for a given communicator before the first call to MMPI\_Barrier. Once “set up” MMPI\_Barrier may be called any number of times to synchronize a communicator’s processors, until MMPI\_Disable is called for that communicator.

3.3.3 Serve

The MMPI\_Serve\_handler routine is called by the application to provide the request manager with application selected opportunities to complete posted requests.

```c
int MMPI\_Serve\_handler();
```

During a call to MMPI\_Serve\_handler the request manager tests the posted requests for completion. If none of the posted requests has completed MMPI\_Serve\_handler returns immediately.

3.3.4 Handler Routine Interactions

Collective communication routines, including MMPI\_Barrier, should not be called from within request handler routines.

As previously noted, request handler routines are given a copy of the request data object input to MMPI\_Post\_handler. Once posted, the original request data object input to MMPI\_Post\_handler abnegates control of the associated asynchronous message to the request manager’s internal copy of the request data object.

This interaction, and its implications, are illustrated in the following example.
struct data {
    MPI_Comm comm;
    MPI_Request request;
    int tag;
    int message[64];
};

int handler(
    void * extra_state,
    MPI_Request * request,
    MPI_Status * status )
{
    struct data * D = (struct data *) extra_state;
    int N, R;
    MPI_Get_count(status,MPI_INT,&N); /* Number of ints received */
    /* ... Process 'D->message[0..N-1]' ... */
    R = MPI_Start( request ); /* Restart the request */
    D->request = *request; /* Insure consistency */
    return R;
}

void start_handler( struct data * D )
{
    /* Create persistent request */
    MPI_Recv_init(D->message,64, MPI_INT,
                  MPI_ANY_SOURCE, D->tag, D->comm, &(D->request) );
    MPI_Start( &(D->request) ); /* Activate the request */
    MMPI_Post_handler(D->request,D,handler); /* Post the request */
}

This is an example of a “self sustaining” receive request, the request is persistent and has a request handler which restarts the request after processing the message. Note on line #19 that the MPI_Request data object input to the request handler is passed to MPI_Start. On on line #20 the resulting possibly updated value is copied back to the original D->request data object for consistency.

A request handler routine may also call point-to-point MPI or MMPI routines, including MMPI_Nait, MMPI_Test, or MMPI_Serve_handler. Calls to these three MMPI routines results in a recursive entry into the MMPI request manager. Recursive invocations of the request manager are safe; however, the request handler (request + routine + data) might also be recursively called if
- the request is persistent,
- the request is reactivated in the handler routine, and
• the request manager is invoked after reactivation.

In the following example, the call to `MMPI.Serve_handler` on line #15 allows the `handler` routine to be called recursively. Such a recursion will occur if pending message matches the `handler`'s request, as the request was activated by the call to `MPI.Start` on line #12.

```c
1   int handler(
2       void * extra_state ,
3       MPI_Request * request ,
4       MPI_Status * status )
5   {
6       struct data * D = (struct data *) extra_state ;
7       int N , R ;
8       MPI_Get_count(status,MPI_INT,&N); /* Number of ints received */
9       /* ... Consume 'D->message[0..N-1]' ... */
10      R = MPI_Start( request ); /* Restart the request */
11      D->request = *request ; /* Insure consistency */
12     MMPI.Serve_handler(); /* Possible recursive call */
13     return R ;
14   }
```
4 Packed Buffer Management

MPI defines several routines for packing and unpacking messages into buffers (Section 3.13 of [3]). These buffers are then sent or received using the MPI_PACKED data type specification. Allocating, packing, and unpacking these buffers can become tedious when numerous messages are packed into the same buffer. The MMPI library provides routines to manage these operations.

4.1 Create, Reset, Free

MMPIBuf is an opaque object type, similar to other MPI opaque object types. MMPIBuf must be created and freed just as other MPI opaque data objects.

    int MMPIBuf.create( int len, MPI_Comm comm, MMPIBuf * buf);
    int MMPIBuf.reset( int len, MPI_Comm comm, MMPIBuf * buf);
    int MMPIBuf.copy( MMPIBuf srcbuf, MMPIBuf * buf);
    int MMPIBuf.free( MMPIBuf * buf);

MMPIBuf.create creates a buffer of an initial size len which is used to send or receive a message via the MPI communicator comm. MMPIBuf.reset "recreates" the buffer with the revised parameters. MMPIBuf.copy creates a copy of the source buffer srcbuf. MMPIBuf.free destroys the buffer and reclaims its memory.

4.2 Query

MMPIBuf data objects may be queried for various attributes.

    int MMPIBuf.capacity( MMPIBuf buf, int * len);
    int MMPIBuf.pointer( MMPIBuf buf, void ** ptr);
    int MMPIBuf.position( MMPIBuf buf, int * pos);
    int MMPIBuf.size( MMPIBuf buf, int * size);
    int MMPIBuf.comm( MMPIBuf buf, MPI_Comm * comm);
    int MMPIBuf.remain( MMPIBuf buf, int * rem);

Capacity is the current allocated length of the buffer. Pointer, position, size, and comm are arguments passed to MPI_Unpack. The remainder is the difference between the size and the position, i.e. the remainder to be unpacked.

4.3 Pack and Send

A MMPIBuf is packed with the routine MMPIBuf_pack. This routine has a similar interface to the MPI routine MPI_Pack. The two significant differences are:

- several calling arguments are "compressed" and
- if the buffer is too small it is automatically lengthened.
int MMPI_Buf_pack(
    void * inbuf,    /* IN */
    int incount,    /* IN */
    MPI_Datatype datatype,    /* IN */
    MMPI_Buf * outbuf );    /* IN OUT */

A packed buffer may be sent with any MPI send routine, with the MPI_Packed data type, buffer size, and buffer communicator. The following MMPI send routines provide syntactic wrappers for sending a MMPI buffer.

int MMPI_Buf_send( MMPI_Buf buf, int dest, int tag);
int MMPI_Buf_rsend( MMPI_Buf buf, int dest, int tag);
int MMPI_Buf_ssend( MMPI_Buf buf, int dest, int tag);
int MMPI_Buf_bsend( MMPI_Buf buf, int dest, int tag);
int MMPI_Buf_isend( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_irsend( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_issend( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_ibsend( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_send_init( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_rsend_init( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_ssend_init( MMPI_Buf buf, int dest, int tag, MPI_Request * request);
int MMPI_Buf_bsend_init( MMPI_Buf buf, int dest, int tag, MPI_Request * request);

4.4 Receive and Unpack

A message may be received into a MMPI buffer. The MPI receive statement must include the buffer’s pointer, length, and communicator and have the MPI_Packed data type specification. The following MMPI receive routines provide syntactic wrappers for receiving an MMPI buffer.
Once an MPI buffer has been received the size of the received message must be extracted from the MPI_Status and registered with the buffer. The following routine performs this task.

```c
int MMPI_Buf_status( MMPI_Buf buf, MPI_Status * status);
```

A received buffer is unpacked with `MMPI_Buf_unpack`. This routine has a similar interface to the MPI routine MP_Unpack. The significant difference is that several calling arguments have been "compressed".

```c
int MMPI_Buf_unpack(  
    MMPI_Buf inbuf,       /* IN */  
    void * outbuf,        /* IN OUT */  
    int outcount,         /* IN */  
    MPI_Datatype datatype); /* IN */
```

The following is an example of a receive and where the status is extracted and registered with the buffer.

```c
int src, tag;
MPI_Status status;
MMPI_Buf recv_buf;

MMPI_Buf_create( 1024, MPI_COMM_WORLD, &recv_buf);

MMPI_Buf_recv( recv_buf, src, tag, &stat);
MMPI_Buf_status( recv_buf, &stat);
```
5 Consumer

An MMPI Consumer is a global opaque object which may receive multiple messages of dynamic size and content. These messages are sent to a consumer handler on a designated processor. A consumer handler is an application provided routine and pointer, similar to the request handler.

5.1 Consumer Handler

The interface requirement for the consumer handler routine is given in the following C type definition.

```c
typedef int (*MMPLCon_handler)(
    void * extra_state,
    int source,
    MMPLBuf recv_buf);
```

Just as with the request handler, the consumer handler data is an anonymous pointer value.

5.2 Create and Free

Creation of a global consumer opaque object is a synchronous operation. Global tags are reserved for the global object when it is created and released when it is freed. As such the MMPLCon_create and MMPLCon_free must be called synchronously on all processors of the given communicator.

```c
int MMPLCon_create(
    MPI_Comm comm,       /* IN */
    void * extra_state,  /* IN */
    MMPLCon_handler recv_handler,  /* IN */
    MMPLCon * consumer);  /* OUT */

int MMPLCon_free( MMPLCon * consumer );

int MMPLCon_reset( MMPLCon * consumer );
```

A global consumer data object allocates buffers on each processor as required to receive dynamic messages. These buffers may be deallocated without freeing the MMPI buffer via the MMPLCon_reset routine. This routine returns the consumer to its initial/created state.
5.3 Pack and Send

Consumer messages are point-to-point messages. Each message sent to a consumer is an explicit operation within the application code, whereas consumer's receive operation is automatically completed. Each received consumer message is processed by the consumer handler.

A consumer message is a packed MMPI buffer. The message buffer must be initialized with a call to MMPI.Con.init and sent with a call to MMPI.Con.send. Between these two calls the buffer may be arbitrarily packed, for subsequent unpacking by a matching consumer handler.

```c
int MMPI_Con_init(
    MMPI_Con consumer, /* IN */
    MMPI_Buf * buf);    /* IN OUT */

int MMPI_Con_send(
    MMPI_Buf buf,       /* IN */
    int destination,    /* IN */
    MMPI_Con consumer); /* IN */
```

The `MMPI_Con_init` routine sets up a header in the buffer which is processed by a consumer request handler on the destination processor. The `MMPI_Con_send` routine insures that the receive buffer on destination processor is ready to receive the message. If the destination processor is not ready, the `MMPI_Con_send` routine blocks via `MMPI.Wait`. Once the destination processor is ready the message is sent with a call to `MPI.ReSend`.

5.4 Test and Wait

After a consumer message is received and processed by the destination consumer handler, an acknowledgement message is returned to the source processor. The application may test or wait for this acknowledgement.

```c
int MMPI_Con_wait(
    MMPI_Con consumer,         /* IN */
    int destination );         /* IN */

int MMPI_Con_test(
    MMPI_Con consumer,         /* IN */
    int destination,            /* IN */
    int * flag );               /* OUT */
```
5.5 Query

An MMPI consumer may be queried for its communicator, handler function, and handler data.

```c
int MMPI_Con_comm(
    MMPI_Con consumer, /* IN */
    MPLComm * comm); /* OUT */
```

```c
int MMPI_Con_func(
    MMPI_Con consumer, /* IN */
    MMPI_Con.handler * handler); /* OUT */
```

```c
int MMPI_Con_data(
    MMPI_Con consumer, /* IN */
    void ** extra_state ); /* OUT */
```

5.6 Consumer Handler Semantics

An application's consumer handler routine may be called recursively. Such a recursive call occurs if 1) the consumer handler routine calls MMPI_Test, MMPI_Wait, or MMPI.Serve_handler (a call to MMPI.Barrier is from within a handler is erroneous) and 2) another message is sent to the consumer handler while the previous message is being processed. Consistency of the recv_buf message buffer is insured by providing the consumer handler routine with a copy of the received message buffer. Thus recursive calls to the consumer handler routine are given a unique recv_buf message buffer; however, each call is given the same extra_state argument.

The consumer handler recursion semantics are controlled by the MMPI Consumer's request handler. A significantly simplified version of MMPI Consumer's request handler is given here.

```c
int con_handler(
    void * extra_state ,
    MPI_Request * request ,
    MPI_Status * status )
{
    MMPI_Con con = (MMPI_Con) extra_state ;
    int src = status->MPI_SOURCE ;
    MMPI_Buf buf = MMPI_BUF_NULL ;
    int ret ;

    /* Consumer message received into 'con->buf' via 'request' */

    MMPI_Buf_status( con->buf , status );
    MMPI_Buf_copy( con->buf , &buf );

    MPI_Start( request );
}
```
/* Another consumer message may now be received into \'con->buf\' */

    ret = (*con->handler)( con->extra_state, src, buf );
    MMPI_Buf_free( &buf );
    return ret ;
}

Recursion is easily avoided, simply do not call MMPI_Test, MMPI_Wait, or MMPI_Serve_handler from within the consumer handler routine. When recursion is permissible, consistency of the shared extra_state and any other shared data is the responsibility of the application.
6 Log File Management

The name of the MMPI log file may be selected immediately following initialization of MPI via call to:

```c
int MMPI_Log_init( char * base_name );
```

where the input `base_name` is appended with the rank of the processor in the `MPI_COMM_WORLD` communicator. The base name for the log file will default to `MMPI.LogP#` unless otherwise specified through an initial call to `MMPI_Log_init`.

The MMPI log file is accessed for output as

```c
int MMPI_Log_file_d(); /* C file descriptor */
FILE * MMPI_Log_file(); /* C FILE buffer */
ostream & MMPI_Log_stream(); // C++ output stream
```

The MMPI log file is also written to by the routines

```c
void MMPI_Log_message( char * who, char * msg);
void MMPI_Log_abort( char * who, char * msg, int mpi_err);
```

where `who` is the name of the calling routine, `msg` is the message to log for that routine, and `mpi_err` is the MPI error code to be passed to `MPI_Abort`. 
7 Examples

Two examples of using the MMPI library are given here. The first example uses MMPI to define a remote put operation. In the second example MMPI is used to define a remote get operation. In both examples the “memory” which is shared is a designated vector declared on each processor. Remote get and put operations to this shared vector are defined by a subvector displacement and length.

In both examples each processor puts or gets a random set of subvectors to or from a randomly determined processor. The source, destination, and content of the put and get messages has been randomized to illustrate how easily such unpredictable messages can be managed with MMPI.

7.1 Remote Put Example

Three routines are defined in the remote put example:

- subvec, a random subvector generation routine,
- con_handler, an MMPI Consumer handler, and
- main, the main program.

The subvec routine generates a subvector of a random displacement, length, and destination processor. The main routine packs this subvector into a buffer and sends it to an MMPI Consumer on the destination processor. The message buffer is unpacked on the destination processor by the con_handler routine and then summed into the “shared” vector.

1    #include <stdio.h>
    #include <stdlib.h>
    #include "mmpi.h"

5    /* Random subvector generation */

extern int subvec( /* Return 0 when done */
    int * dest , /* OUT: Destination processor */
    int * disp , /* OUT: Subvector displacement */
    int * len , /* OUT: Subvector length */
    double * val ); /* OUT: Subvector values */

    /* Consumer subprogram */

15   extern int con_handler(
        void * extra_state , /* IN OUT: Consumer subprogram data */
        int source , /* IN: Source of the message */
        MMPI_Buf recv_buf ); /* IN: Message contents */

20   #define GLEN    100
    #define LLEN    10
main( int argc , char * argv[] )
{
    MMPI_Buf send_buffer = MMPI_BUF_NULL;
    MMPI_Con consumer = MMPI_CON_NULL;
    double shared_data[ GLEN ];
    double send_data[ LLEN ];
    int i , proc , disp , len ;
    for ( i = 0 ; i < GLEN ; ++i ) shared_data[i] = 0.0 ;

    MPI_Init( &argc, &argv ); /* Initialize MPI */
    MMPI_Log_init( "Demo1.P" ); /* Name the MMPI Error/Warning file */
    MMPI_Enable( MMPI_COMM_WORLD );

    /* Synchronous creation of Consumer object.
       The 'extra_state' is the shared vector 'shared_data'. */
    MMPI_Con_create( MPI_COMM_WORLD, shared_data, con_handler, &consumer );

    /* Begin asynchronous loop */
    while ( subvec( &proc, &disp, &len, &send_data ) ) {

        /* Pack and send subvector */
        MMPI_Con_init( consumer, &send_buffer );
        MMPI_Buf_pack( &disp , 1 , MPI_INT , &send_buffer );
        MMPI_Buf_pack( &len , 1 , MPI_INT , &send_buffer );
        MMPI_Buf_pack( send_data, len, MPI_DOUBLE , &send_buffer );
        MMPI_Con_send( send_buffer, proc , consumer );
    }

    /* End asynchronous loop. Done with the consumer. */
    MMPI_Buf_free( &send_buffer );
    MMPI_Con_free( &consumer ); /* MMPI_Barrier called internally */

    /* Print the results */
    for ( i = 0 ; i < GLEN ; ++i ) {
        if ( 0 == i % 5 ) fprintf( MMPI_Log_file(), "\n" );
        fprintf( MMPI_Log_file(), " %f", shared_data[i] );
    }
    fprintf( MMPI_Log_file(), "\n" );

    MMPI_Disable( MPI_COMM_WORLD );
    MPI_Finalize();
    return 0 ;
}

    /* Subvector summation handler */
int con_handler( void * extra_state, int source, MMPI_Buf Buf )
{
  double data[ LLEN ];
  int disp, len, i;
  double * shared_data = (double *) extra_state;

  /* Unpack the subvector */
  MMPI_Buf_unpack( Buf, &disp, 1, MPI_INT );
  MMPI_Buf_unpack( Buf, &len, 1, MPI_INT );
  MMPI_Buf_unpack( Buf, data, len, MPI_DOUBLE );

  /* Summation */
  for ( i = 0; i < len; ++i ) shared_data[disp+i] += data[i];

  return MPI_SUCCESS;
}

/* Generate random subvector contributions */
int subvec( int * dest, int * disp, int * len, double * val )
{
  static int count = 0;
  static int num = 0;
  static int proc = -1;
  static int nproc = -1;
  int i;

  if ( 0 == num ) {
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &proc );

    srand( proc ); /* Different random numbers on each processor */

    num = 10 + rand() % 10; /* Random number of iterations */
  }

  *disp = rand() % (GLEN - LLEN); /* Random displacement */
  *len = 1 + rand() % (LLEN - 1); /* Random length */
  *dest = rand() % nproc; /* Random destination */

  for ( i = 0; i < *len; ++i ) val[i] = 1 + proc;

  return ( ++count < num );
}
Initialization (Lines #33-39)

Immediately following the call to MPI_Init, which initializes MPI, the MMPI log file is named with a call to MMPI_Log_init. The given log file name is appended with the processor's rank within the MPI_COMM_WORLD communicator. This file will be opened for writing upon the first call to MMPI_Log_file_d, MMPI_Log_file, or MMPI_Log_stream. The MMPI log file will not be opened unless one of these routines is called.

The call to MPI_Init activates tag management for the MPI_COMM_WORLD communicator. Recall that this routine contains a call to MPI_Bcast.

The MMPI Consumer is created with a call to MMPI_Con_create on line #39. This must be a synchronous call performed by all processors of the input communicator, MPI_COMM_WORLD. The second and third arguments, shared_data and con_handler, are the handler data and handler routine which is called to process messages sent to the consumer. The final argument, consumer, is the MMPI Consumer object. The handler data must be a pointer convertible to void * and the handler routine must conform to the MMPI_Con_handler type specification (Section 5). The handler data (shared_data) is passed to the handler routine (line #74) via the routine's first argument, labeled extra_state in the example.

Asynchronous Part (Lines #41-54)

The main routine's "asynchronous part" performs a random number of iterations on each processor. Within this part of the example program each processor is "loosely" Single Instruction Multiple Data (SIMD). The processors are performing similar but not identical operations, which is emphasized by each processor performing a different number of iterations.

Within the asynchronous part of the main routine consumer messages are generated with a similar format: subvector displacement, subvector length, and subvector values. The subvector is packed into a message buffer in four steps (lines #47-50):

- MMPI_Con_init is called to initialize the message buffer for the given consumer,
- two calls to MMPI_Buf_pack pack the displacement and length, and
- a final call to MMPI_Buf_pack packs the subvector values.

The packed buffer is sent to the MMPI Consumer on the destination processor with a call to MMPI_Con_send (line #51).

The consumer messages of this example have both a random destination and random size. Thus the number of messages, message source-destination, and exact message content cannot be determined a-priori. It is applications with this type of non-deterministic communication patterns which motivated the development of MMPI.
Re-Synchronization and Conclusion (Lines #57-70)

The main routine ends the asynchronous part by calling MMPI_Con_free on line #57. This MMPI routine contains a call to MMPI_Barrier which synchronizes the consumer's processors. Once the consumer's processors have been synchronized, it is certain that all pending consumer messages have been processed and so it is safe to destroy the consumer.

The resulting “shared” vector shared_data is printed by each processor to the MMPI log file. A call to MMPI_Log_file returns the C FILE pointer to this file.

Consumer Handler (Lines #74-91)

The con_handler subprogram, beginning on line #74, is called to process each message received by the consumer. Recall that the value of the extra_state argument passed to the con_handler routine is the shared_data pointer which is passed to MMPI_Con_create on line #39. The source argument identifies the source of the message while the Buf argument hold the message contents.

The con_handler routine casts the extra_state to a pointer of the appropriate type on line #78. The put subvector's displacement, length, and values are unpacked from the MMPI Buffer Buf on lines #82-84. The put subvector is then summed into the appropriate subvector of shared_data on line #88.

Random Subvector Routine (Lines #95-119)

The random subvector generation routine subvec has been included to complete the example. Note that the processor rank within MPI_COMM_WORLD is used to seed the random number generator on line #107. Thus each processor's subvec routine should generate a different set of random subvectors and message destinations.

7.2 Remote Get Example

The remote get example also defines three routines:

- subvec_get, a random subvector retrieval routine.
- con_handler, an MMPI Consumer handler, and
- main, the main program.

The subvec_get routine retrieves a subvector of a random displacement, length, and source processor. The main routine sums the retrieved subvector into a local vector. The con_handler routine accepts subvector get requests and responds with the requested subvector values.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mmpi.h"
```

TICAM 25 October 11, 1996
extern int con_handler( void * , int , MMPI_Buf );
extern int subvec_get( MMPI_Cdn consumer , int * disp , int * len , double * val );

#define GLEN 100
#define LLEN 10

int main( int argc , char * argv[] )
{
    MMPI_Cdn consumer = MMPI_CON_NULL ;
    double local_data[ GLEN ];
    double shared_data[ GLEN ];
    double recv_data[ LLEN ];
    int i , proc , disp , len ;

    MPI_Init( &argc, &argv ); /* Initialize MPI */
    MMPI_Log_init( "Demo2.P" ); /* Name the MMPI Error/Warning file */
    MMPI_Enable( MPI_CDMM_WDRLD );
    MPI_Comm_rank(MPI_CDMM_WDRLD , &proc );

    /* Initialize local and shared vectors */
    for ( i = 0 ; i < GLEN ; ++i ) local_data[i] = 0.0 ;
    for ( i = 0 ; i < GLEN ; ++i )
        shared_data[i] = 1000 * ( proc + 1 ) + i ;

    /* Synchronous creation of Consumer object. 
The 'extra_state' is the shared vector 'shared_data'. */
    MMPI_Cdn_create( MPI_COMM_WORLD, shared_data, con_handler , &consumer );

    /* Begin asynchronous loop */
    while ( subvec_get( consumer , &disp , &len , recv_data ) ){

        /* Sum received subvector */
        for ( i = 0 ; i < len ; ++i ) local_data[disp+i] += recv_data[i] ;
    }

    /* End asynchronous loop. */
    MMPI_Cdn_free( &consumer ); /* MPI_Barrier called internally */

    /* Print the results */
    for ( i = 0 ; i < GLEN ; ++i ){
        if ( i == 0 ) fprintf( MMPI_Log_file() , "\n" );
        fprintf( MMPI_Log_file() , " %f" , local_data[i] );
    }
}

TICAM 26 October 11. 1996
fprintf( MMPI_Log_file(), "\n" );

MMPI_Disable( MPI_COMM_WORLD );
MPI_Finalize();
return 0 ;
}

/* Random subvector requests */

int subvec_get( MMPI_Con consumer, int * disp, int * len, double * val )
{
    static int count = 0 ;
    static int num = 0 ;
    static int proc = -1 ;
    static int nproc = -1 ;
    MPI_Request request ;
    MPI_Status status ;
    MMPI_Buf send_buffer = MMPI_BUF_NULL ;

    int src, tag ;

    if ( 0 == num ) {
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &proc);

        srand( proc ); /* Different random numbers on each processor */

        num = 10 + rand() % 10 ; /* Random number of iterations */
    }

    *disp = rand() % ( GLEN - LLEN ) ; /* Random displacement */
    *len = 1 + rand() % ( LLEN - 1 ) ; /* Random length */
    src = rand() % nproc ; /* Random source */

    MPI_Tag_get_local( MPI_COMM_WORLD, &tag );

    /* Start receive operation for the response */
    MPI_Irecv( val, *len, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &request );

    /* Send the request */
    MMPI_Con_init(consumer, &send_buffer);
    MMPI_Buf_pack(&tag, 1, MPI_INT, &send_buffer);
    MMPI_Buf_pack(disp, 1, MPI_INT, &send_buffer);
    MMPI_Buf_pack(len, 1, MPI_INT, &send_buffer);
    MMPI_Con_send( send_buffer, src, consumer );
MMPI_Con_free( &send_buffer );

/* Wait for response. This processor’s handlers may be invoked while waiting. */
MMPI_Wait( &request, &status );

/* Release the tag */
MMPI_Tag_rel_local( MPI_COMM_WORLD, &tag );

return ( ++count < num );
}

/* Subvector ‘get’ handler */
int con_handler( void * extra_state, int source, MMPI_Buf Buf )
{
  double * shared_data = (double *) extra_state;
  MPI_Comm comm;
  int tag, disp, len;

  MMPI_Buf_comm( Buf, &comm ); /* Query the communicator */

  /* Unpack the subvector request */
  MMPI_Buf_unpack( Buf, &tag, 1, MPI_INT );
  MMPI_Buf_unpack( Buf, &disp, 1, MPI_INT );
  MMPI_Buf_unpack( Buf, &len, 1, MPI_INT );

  /* Reply with subvector values */
  MPI_Rsend( shared_data + disp, len, MPI_DOUBLE, source, tag, comm );

  return MPI_SUCCESS ;
}

---

Main Routine (Lines #13-62)

In this example the main routine declares a “shared” vector shared_data which is queried by other processors, and a local vector local_data into which queried subvectors are summed. In the asynchronous part of the main routine, lines #38-47, the subvec_get routine is called a random number of times to retrieve random subvectors. Each subvector is summed into the local_data.
Random Subvector Retrieval Routine (Lines #66-119)

The random subvector routine subvec.get generates random subvector specifications and retrieves the corresponding subvector values from a random processor. The remote get procedure in subvec.get begins on line #90 and ends on line #166. This procedure is as follows:

- start a nonblocking receive for the requested subvector values,
- send the subvector request to the “source” processor’s consumer,
- wait for the “source” processor to reply with the subvector values.

The remote get procedure’s nonblocking receive requires a message tag which is guaranteed to be unique within the MPI communicator. Such a tag is obtained with a call to MMPI.Tag.get.local. The returned tag is guaranteed to 1) not conflict with any of MMPI’s internal tags for that communicator and 2) not be returned by any other call to MMPI.Tag.get.local for the communicator, until the tag is explicitly released by the application. Thus as long as the application obtains all tags from the MMPI.Tag.* routines the tags will be unique to the communicator.

The nonblocking receive is started with a call to MPI.Irecv on line #96. This call to MPI.Irecv explicitly specifies the tag and source, MPI.ANY_TAG and MPI.ANY_SOURCE are not used.

The subvector request is packed into a message buffer and sent to the MMPI Consumer on the “source” processor (lines #101-105). Note on line #102 that the tag of the non-blocking receive is included in the subvector request message. This enables the “source” processor to form the appropriate reply message.

Once the request message has been sent the subvec.get routine calls the MMPI.Wait routine to wait for the “source” processor to reply with the requested subvector values. The MMPI.Wait routine is called, as opposed to the MPI.Wait routine, so that the local processor may complete other asynchronous messages while waiting. Consider the situation where two processors simultaneously send a subvector requests to one another. If each processor were to call MPI.Wait within the subvec.get routine the example would deadlock. However, by calling MMPI.Wait instead the processors may respond to each other’s request while waiting for their own request to be satisfied.

After the subvector request has been received, as indicated by the call to MMPI.Wait returning, the previously obtained tag is is not longer needed. This tag is released for subsequent re-use by the call to MMPI.Tag.rel.local on line #116.

Consumer Handler (Lines #123-142)

The example’s MMPI Consumer handler processes the subvector request by sending the requested subvector values to the requesting processor. Recall that the requesting processor started an asynchronous receive prior to sending out the subvector request. This ordering of operations on the requesting processor allows the con.handler routine to call MPI.Rsend (line #139) when replying with the requested subvector values.
8 Conclusion

The MMPI source code is available in the public domain under GNU license. The MMPI source code and other information can be obtained at

http://www.ticam.utexas.edu/carter/mmpi.html

This WWW site includes updates to this report, list of host+MPI configurations under which MMPI has been tested, and relevant WWW links.

The MMPI library is being used in the development of two parallel libraries:
- Scalable Distributed Dynamic Array (SDDA) and
- Parallel Linear Algebra Package (PLAPACK) [5].

The Scalable Distributed Dynamic Array (SDDA) library consists of a small set of C++ components supporting distributed object management. The SDDA uses MMPI to provide a “shared memory” view of a set of distributed data objects, where every processor may randomly access data objects regardless of their distribution.

The Parallel Linear Algebra Package (PLAPACK) uses MMPI to support application filling and querying of distributed vectors and matrices. MMPI is used to provide the application with a “shared memory” view of the PLAPACK vectors and matrices, where every processor may randomly access any part of a distributed vector or matrix.

Requirements for asynchronous message management in MPI are identified and analyzed in this report. The MMPI library represents one approach to meeting these requirements. These requirements, as presented in this report, will be submitted to the Message-Passing Interface Forum (see http://parallel.nas.nasa.gov/MPI-2/). The goal is to have the MPI-2 standard meet the asynchronous message management requirements presented here.

Acknowledgements

Development of the MMPI library was funded by ARPA Contract DABT63-92-C-0042.

The MMPI functionality was originally developed within the “alpha version” Scalable Distributed Dynamic Array (SDDA). Collaborators for this “alpha version” SDDA include Dr. James C. Browne, Dr. Robert van de Geijn, and Dr. Abani K. Patra. Separation of the MMPI functionality from the SDDA, and its current MPI-like syntax and semantics, was motivated in part by the MPI-2 effort. We thank Dr. Steven Huss-Lederman for discussions regarding MPI request handler semantics.
References


