AUTOPACK User Manual *
Version 1.3

Raymond Loy
rloy@mcs.anl.gov
Math and Computer Science Division
Argonne National Laboratory

May 11, 2000

*This manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy.

Contents

1 Introduction

AUTOPACK is a library that provides several useful features for programs using the Message Passing Interface (MPI) [2]:

- An automatic message packing facility
- Management of send and receive requests
- Management of message buffer memory
- Determination of the number of anticipated messages from a set of arbitrary sends
- Deterministic message delivery for testing purposes

Taking advantage of the message packing, a program may send large numbers of small messages without incurring large overhead. The small messages are automatically assembled into larger packages. These packages are sent to the destination where they are automatically burst and the individual messages delivered. This allows user code to be written in a natural style while still achieving high efficiency. It avoids the need for the user to write code to pack messages. Without modifying code, experimentation with package size and other parameters is easily conducted to obtain optimal performance for a given MPP architecture.

The library automatically manages message-passing send and receive requests. In general, the user code need not be concerned with pending send requests other than to query the library to determine if they have completed or to determine the number pending. Internal space related to requests is automatically released when requests have cleared. Receive requests are managed without direct interaction from the user code.

This library incorporates a mechanism for dynamically allocating message buffer memory for both outgoing and incoming messages. This design allows for increased efficiency by avoiding memory-to-memory copy operations. In particular it makes possible the construction of message packages without copying.

Dynamic and irregular computation often results in an unpredictable number of messages communicated among the processors. It is difficult to determine how many of these messages have a given processor as their destination without synchronized global communication. Yet this information is essential since a processor must know how many messages to wait for before proceeding to the next step of an algorithm. AUTOPACK provides a way to determine the number of incoming messages that does not require barrier synchronization. This permits the code to obtain better performance through asynchronous behaviour.

While developing parallel code, it is often desirable to have messages delivered in a deterministic order. For example, this may aid in debugging or benchmarking. However, MPI guarantees message ordering only on a point-to-point basis. The AUTOPACK library provides a convenient way to guarantee message ordering globally. For a given set of message sends among the processors, a given destination processor will always receive the same sequence of messages. This feature has attendant memory and performance costs, but code that employs it can be easily switched to normal message delivery (point-to-point ordering) when the global ordering is no longer desired.

AUTOPACK is built on top of MPI and has similar syntax and semantics for sending and receiving. The complete API consists of just 18 functions; most applications will have need for less than half of these. It is easy to learn how to use and adapting existing MPI code to use AUTOPACK is minimally invasive.

Comments and suggestions are welcome.

2 Language Issues

2.1 C Binding Issues

All AUTOPACK names have the prefix "AP_". Programs must not declare any symbols that begin with this prefix. Prototypes for all public functions, and definitions of public constants, are provided in the file "autopack.h".

2.2 Fortran Binding Issues

All AUTOPACK names have the prefix "AP_", and all characters are capitals. Programs must not declare any symbols that begin with this prefix. Fortran `PARAMETER` definitions are provided in the file "autopack.fh".

The Fortran binding for a library function, if available, is documented in the function's description following the C synopsis. Functions that are present only in the Fortran interface appear in the function index in all caps.

In general, the Fortran versions of routines have the same name as the C routine, but with all letters capitalized. E.g. `AP_INIT` is the Fortran version of `AP_init()`. A Fortran routine generally has the same arguments as its C equivalent, but with value parameters being passed by reference as required by Fortran. If the C routine has a return value, the Fortran version will have an additional argument at the end of its parameter list, "`return_value`", through which the return value is passed back.

Difficulties arise because Fortran does not have the ability to deal with pointers. As a result, both sending and receiving introduce an additional memory-to-memory copy. Details are discussed in Sections 4.2 and 4.4.

3 Library Initialization and Settings

Before use the library must be initialized. The initialization call must come after the call to initialize MPI:

```
...
MPI_Init(&argc,&argv);
AP_init(&argc,&argv);
AP_setparam(size,packed,maxreq_proc,maxreq);  /* optional */
...
```

The call to `AP_setparam()` is optional to specify settings that affect the library's behavior.

- **size:** Controls the library's memory block allocation size which also determines package size. This is only a default; if the user requests a single message buffer larger than this, enough memory will be allocated for a block that can hold the message.
- **packed:** If nonzero, enables message packing. Messages are automatically grouped into packages and their actual send may be delayed. If this argument is zero, message packing is disabled. Messages must still be allocated through the library but they are sent individually. This can be useful for making comparisons.
- **maxreq_proc:** This sets a per-destination limit on the number of MPI send requests that will be posted at one time. This can help to prevent overflowing the capacity of some MPI implementations.
- **maxreq:** This sets an overall limit on the number of MPI send requests that will be posted at one time. It takes precedence over `maxreq_proc`. If `maxreq` is negative, no global limit is set (it is effectively the number of processors multiplied by `maxreq_proc`).

Different architectures and MPI implementations will achieve their best performance with different settings.

4 Sending and Receiving Messages

4.1 Message Buffer Memory Model

AUTOPACK assumes the responsibility for memory allocation for all outgoing and incoming messages. This model of allocation allows the library to do several things more efficiently. Although the user is unaware of it, the library allocates messages going to the same destination contiguously in memory. Thus, when the time comes to send a package, no memory-to-memory copying is necessary; it simply sends a contiguous block of these messages. On the receive side, a package is stored in memory and pointers to the individual messages within are returned to the user rather than requiring a copy operation to dispense each message. For both sending and receiving, the underlying memory allocation is done in large blocks for efficiency.

Note: This model is best suited for dynamic situations where the message must be constructed before it is sent. For example, the user might want to allocate a struct, fill in its fields, send it, then delete it. It is not the most appropriate for situations where the data to be sent already exists in memory. For example, if the sender has a large array and wants to sent its contents to an existing array on the destination. In that case, standard MPI calls are the most efficient way to accomplish the communication. In the case where the data already exists in memory, but is small, the user must evaluate the tradeoff between the overhead of sending a small message versus the overhead of copying the message from its current place in memory to a buffer allocated by AUTOPACK . If there are many messages the latter is likely to be more efficient.

4.2 Sending

In the spirit of `malloc()`, the user must call a function to allocate an outgoing message. At the time of allocation, the user must declare not only the size of the message but its destination and tag. The user may then store data in this memory area, and notify the library when it is ready to be sent. After sending the message, the user is not longer permitted to access the message buffer (Figure 1).

```
...
int *message;

message= (int *)AP_alloc(destination,tag,2*sizeof(int));
message[0]=10;
message[1]=20;
AP_send(message);
...
```

Figure 1: Allocating and sending a message in C

In Fortran, there is no way to access a message buffer allocated by the library, so it is not possible to have separate calls for message allocation and sending. Fortran is provided with a single routine `AP_BSEND()` to perform the allocation, copy the data from a Fortran variable to the message buffer, and send the message (Figure 2).

```
...
INTEGER*4 message(2)

message(0)=10
message(1)=20
CALL AP_BSEND(message,2*4,destination,tag)
...
```

Figure 2: Sending a message in Fortran

4.3 Send Semantics

All message sends are non-blocking and return immediately. This corresponds to MPI's "immediate" send mode. Whether packing is enabled or not, messages are *non-overtaking* as with MPI.

An AUTOPACK send operation is complete when the message has been passed to MPI, and MPI is done using the message buffer. There are some requirements for send completion:

- *The library does not guarantee any send to complete until after a subsequent call is made to* `AP_flush()`.

  Rationale: If the user has enabled message packing, packages are only sent when full. User allocation of a message send buffer which is too large to fit into the remaining space of a package causes the package to be sent and a new one allocated. A call to `AP_flush()` forces the library to send any partially full packages. It is good programming practice to call `AP_flush()` even if packing is not enabled, so that in the future packing may be enabled easily.

- *The library does not guarantee any send to complete until after a call to* `AP_check_sends()` *returns a value less than or equal to 0. The call is not just informational; it is required to ensure progress.*

  Rationale: Parameters `maxreq_proc` and `maxreq` (see `AP_setparam()`) govern the number of simultaneous MPI send requests the library will make. Once either limit has been reached, the library will defer sending any additional messages. The status of prior MPI send requests is checked when the user calls `AP_check_sends()`. Whenever a send request completes, the library will automatically post as many deferred messages as possible. A negative return value indicates there are no remaining deferred sends; a 0 return value indicates all sends are complete and their buffer space has been freed.

As with MPI, completion of a send operation does not necessarily imply that the destination has received the message.

For convenience, the library will also try to process deferred sends whenever `AP_recv()` retrieves new messages from MPI. While progress will be made sending deferred messages, the user will not be aware if any remain and must eventually call `AP_check_sends()`.

4.4 Receiving

On the receiving side, the user calls `AP_recv()` giving criteria for the message it wishes to receive. Flags may be specified to select various options such as blocking. Memory is automatically allocated to receive messages from MPI. If a message is available that matches the user's criteria, a pointer into the library's storage area is returned. When the user is done examining the contents of the message, a call to the library notifies it that the space may be reclaimed (Figure 3).

```
...
int *ret_msg;

if (AP_recv(MPI_ANY_SOURCE,MPI_ANY_TAG,flags,
    &ret_msg,&ret_size,&ret_sender,&ret_tag))
  {
    printf("Received message %d %d\n",ret_msg[0],ret_msg[1]);
    AP_free(ret_msg);
  }
...
```

Figure 3: Receiving and freeing a message in C

In Fortran, the `AP_RECV()` routine is unable to pass back a pointer to the received message. Instead, it returns an integer descriptor. Once the user code has examined the tag and size, and decided which Fortran variable to store the message in, this descriptor is given to `AP_COPY_FREE()` to copy the message there and free the library's buffer (Figure 4).

```
      ...
      INTEGER m(2), ret_msg

      call AP_RECV(MPI_ANY_SOURCE,MPI_ANY_TAG,flags,
     1             ret_msg,ret_size,ret_sender,ret_tag,
     2             return_value)

      if (return_value.ne.0) then
        call AP_COPY_FREE(m,ret_msg,ret_size)
        write(6,*) 'Received message',m(0),m(1)
      endif
      ...
```

Figure 4: Receiving and freeing a message in Fortran

5 Asynchronous Reductions

AUTOPACK provides a way to determine the number of incoming messages that result from a group of sends (Figure 5). Each processor calls `AP_send_begin()` prior to sending its messages, and `AP_send_end()` afterwards. During this interval, on each processor the library automatically keeps track of how many messages are sent to each destination. A global reduction of this data yields the total number of messages to be received on each processor. The library can perform this reduction without global synchronization. A call to `AP_recv_count()` queries the result of the reduction. If the return value is zero, the reduction is not complete and the processor should wait for additional messages. It is safe to block on incoming messages as long as the `AP_DROPOUT` flag is specified. This flag causes the call to unblock if the result of the reduction arrives. When `AP_recv_count()` indicates the reduction is complete, the argument will indicate the number of messages that were sent to this processor. This number can be compared to the number already received to see how many more are yet to arrive. After receiving all its messages, a processor should call `AP_check_sends()` before proceeding so as to ensure that any deferred messages are sent out.

The entire process may be repeated without performing any synchronization. However, be careful that messages sent from processors that enter the second stage early are not accidentally received by processors in the first stage, which will cause confusion in the count. One way to avoid this problem is to use distinct tags for sending in each stage of the communication, and do not use `MPI_ANY_TAG` when receiving.

```
...
tag=AP_send_begin();

for (...)
  {
    msg=AP_alloc(dest,tag,size);
    ...
    AP_send(msg);
  }

AP_send_end();

i=0;
while ( !(AP_recv_count(&count) || i<count )
  if ( AP_recv(MPI_ANY_SOURCE, tag, AP_BLOCKING|AP_DROPOUT,
               (void **)&msg, &size, &sender, &tag) )
    {
        ...
        AP_free(msg);
    }

AP_check_sends(AP_WAITDEFER);
...
```

Figure 5: Using asynchronous reduction to determine number of messages to receive.

6 Deterministic Message Delivery

This feature allows the user to trade performance and memory in return for deterministic delivery of messages. For a given set of message sends among multiple processors, a given destination processor will always receive the same sequence of messages. This is the case regardless of the receive criteria (requested tag and source) as long as the same receive criteria are issued in the same order each time.

In order to accomplish this, use the same library constructs as for an Asynchronous Reduction (Section 5), with the exception that `AP_dsend_begin()` and `AP_dsend_end()` are called rather than `AP_send_begin()` and `AP_send_end()`

There are several restrictions and caveats to deterministic message delivery:

- Upon entry to `AP_dsend_begin()`, there must be no outstanding messages (i.e. messages that have been sent but not yet received). If any are detected it is considered a fatal error as the library will not be able to guarantee ordering.
- `AP_dsend_begin()` and `AP_dsend_end()` will both perform barrier synchronizations.
- `AP_dsend_end()` will not return until all messages from the block of sends have been received into library memory. It is not possible to start processing the incoming messages before all have arrived, and memory requirements are therefore increased.

7 MPI Issues

AUTOPACK creates its own MPI communicator and will not interfere with any normal MPI operations. The user may mix and match calls to either library as appropriate.

At present, all messages are sent and received using type `MPI_BYTE`. No data conversion is done between sender and receiver.

The library reserves use of several message tags at the top of the permissible range of tag numbers. The maximum MPI tag is set by the AP_MAX_TAG (default 32767). The user code must use message tags that are less than AP_LIB_TAG (default 32764). These constants are defined in the source file "header.h".

The library requires the message tag to be declared at the time of message buffer allocation. The library does not actually need to know the tag until the message is sent, but for simplicity the API asks for all the information up front.

At present, all receives are performed via MPI_Probe()/MPI_Iprobe() and MPI_Recv(). While this is a correct use of MPI, some MPI implementations may not operate as efficiently as they could compared to using MPI_Irecv(). Implementing the library functionality using MPI_Irecv() is planned although its use will impose some restrictions.

8 Acknowledgments

AUTOPACK is based on a prototype developed at Rensselaer Polytechnic Institute and described in [1].

The author thanks James Teresco for his suggestions and help with testing.

References

[1] J. E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Load balancing and communication optimization for parallel adaptive finite element computation. In *Proc. XVII Int. Conf. Chilean Comp. Sci. Soc.*, pages 246–255, Los Alamitos, CA, 1997. IEEE.

[2] Message Passing Interface (MPI) Forum. MPI documents. http://www.mpi-forum.org/docs/docs.html.

9 Release Notes

9.1 Version 1.3

The arguments of `AP_init()` have been changed (`AP_init(&argc,&argv)`) to be like `MPI_Init()`. To update old code, change the call to `AP_init()` to be a call to `AP_setparam()` with the same arguments, and immediately prior to that statement insert a call to `AP_init(&argc,&argv)` (or `AP_init(NULL,NULL)` if argc and argv are not available).

The global variables `AP_nprocs` and `AP_mypid` (as well as the Fortran interface functions `AP_NPROCS` and `AP_MYPID` are still supported but deprecated. New code should use `AP_rank`/`AP_RANK` and `AP_size`/`AP_SIZE`. The names were changed to more consistant with MPI.

10 Appendix A: Example Programs

This section under construction.

Some example programs and a Makefile may be found in the subdirectory ''examples''.

11 Appendix B: Library Reference

AP_COPY_FREE --- Copy message to Fortran variable and free buffer

Synopsis

```
SUBROUTINE AP_COPY_FREE(dest, src, count)
<anytype> dest
INTEGER src, count
```

Parameters

dest	a Fortran variable where the user wants to copy the message
src	the descriptor passed back by AP_RECV()
count	the size passed back by AP_RECV()

Description

After receiving a message using the Fortran binding of AP_recv(), call this function to copy the message into Fortran memory space. After copying, this function will free the message descriptor so there is no need to call AP_free().

AP_RANK --- get the rank of this processor

Synopsis

```
SUBROUTINE AP_RANK(rank)
INTEGER rank
```

Parameters

rank on return this will be set to the rank

Description

Duplicates MPI functionality; provided for convenience in the Fortran interface.
Note - in C, the user should access the global variable AP_rank to obtain the processor rank.

AP_SIZE --- get the number of processors

Synopsis

```
SUBROUTINE AP_SIZE(size)
INTEGER size
```

Parameters

size on return this will be set to the number of processors

Description

Duplicates MPI functionality; provided for convenience in the Fortran interface.
Note - in C, the user should access the global variable AP_size to access the number of processors.

AP_alloc --- Allocate an outgoing message buffer

Synopsis

```
#include "autopack.h"

void *AP_alloc(int destpid, int tag, int size )
```

Parameters

destpid	the destination of the message
tag	the message's tag
size	size of the message in bytes

Description

Allocates a buffer for an outgoing message with the given destination, tag, and size. Returns a pointer to this buffer. After the caller has stored data in the buffer, AP_send() is used to send it.

Return Value

A pointer to the allocated buffer.

AP_bsend --- buffered send

Synopsis

```
#include "autopack.h"

void AP_bsend(void *buf, int size, int dest, int tag)

SUBROUTINE AP_BSEND(buf, size, dest, tag)
<anytype> buf
INTEGER size, dest, tag
```

Parameters

buf	the data to send
size	size (in bytes) of data to send
dest	destination rank
tag	message tag

Description

Allocates a send buffer, copies the data to this buffer, and sends it.
Try to avoid using this function because it introduces a potentially unnecessary memory-memory copy.
This function provided mainly for backwards compatibility, and for use within the Fortran interface.

AP_check_sends --- check status of underlying MPI sends

Synopsis

```
#include "autopack.h"

int AP_check_sends(int flags)

SUBROUTINE AP_CHECK_SENDS(flags, return_value)
INTEGER flags, return_value
INCLUDE 'autopack.fh'
```

Parameters

flags specify options (see below)

Description

Check to see if MPI has finished sending any messages, and free their buffer space if they are done. Also, if there are any deferred messages, send as many as possible.
Flags may be a bitwise OR of the following (in Fortran use addition):

AP_NOFLAGS

Do not block (default)

AP_BLOCKING

Block until at least one send completes

AP_WAITDEFER

Block until all deferred sends are posted to MPI (always returns <=0)

AP_WAITALL

Block until MPI completes all sends (always returns 0)

Caution must be used to avoid deadlock when using AP_BLOCKING. For example, if two processors call this after sending each other messages, neither send is guaranteed to complete before a receive is performed.

Return value

0 if all sends have completed.
If there are deferred sends, returns how many.
If there are no deferred sends, returns -1 times the number of incomplete MPI send requests.
(Note: packages are only counted as a single send.)

AP_check_sends_proc --- check sends to a single destination

Synopsis

```
#include "autopack.h"

int AP_check_sends_proc(int pid, int flags)

SUBROUTINE AP_CHECK_SENDS_PROC(pid, flags, return_value)
INTEGER pid, flags, return_value
INCLUDE 'autopack.fh'
```

Parameters

pid	destination rank to check
flags	specifies options (see below)

Description

Like AP_check_sends(), but only check sends to the given destination. If there are deferred messages to the destination, send as many as possible.
Flags may be one of the following:

AP_NOFLAGS

Do not block (default)

AP_BLOCKING

Block until the first MPI send request for the specified proc is complete, then check remaining ones without blocking.

Return Value

```
Like AP_check_sends() but value only reflects sends
deferred/waiting for the specified processor.
I.e.
0 if all sends to the destination have completed.
If there are deferred sends to the destination, returns how
many.
If there are no deferred sends to the destination, returns -1
times the number of incomplete MPI send requests.
(Note:  packages are only counted as a single send.)
```

AP_dsend_begin --- denote the start of a batch of deterministic sends

Synopsis

```
#include "autopack.h"

int AP_dsend_begin(void)

SUBROUTINE AP_DSEND_BEGIN(return_value)
INTEGER return_value
```

Parameters

None

Description

Like AP_send_begin(), but denote the start of a batch of sends that are guaranteed to arrive in deterministic order. After the sends, call AP_dsend_end().

Caveats

When this is called, there must be no outstanding messages (i.e. sent but not received) in the system.
Messages sent within the bounds of AP_dsend_begin()/AP_dsend_end() will arrive in a deterministic order.
Unlike AP_send_begin(), *this call will cause a global synchronization.*
Memory requirements will be substantially higher than for AP_send_begin(), because no messages are passed on to the user until all have arrived at this processor.

Return value

Same as for AP_send_begin().

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_dsend_end --- End a batch of deterministic sends, initiate asynchronous reduction

Synopsis

```
#include "autopack.h"

void AP_dsend_end(void)

SUBROUTINE AP_DSEND_END(return_value)
INTEGER return_value
```

Parameters

None

Description

Like AP_send_end(), but end a batch of messages that are guaranteed to arrive in deterministic order.
Unlike AP_send_end(), *this call will cause a global synchronization.*

Return value

None

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_finalize --- terminate the AUTOPACK library

Synopsis

```
#include "autopack.h"

void AP_finalize(void)

SUBROUTINE AP_FINALIZE
```

Parameters

None

Description

This routine should be called when user is done using AUTOPACK.
Call before MPI_Finalize().

Return Value

None

AP_flush `--- flush all pending sends`

Synopsis

```
#include "autopack.h"

void AP_flush(void)

SUBROUTINE AP_FLUSH
```

Parameters

None

Return Value

None

Description

Flush all sends. Any open packages are closed and sent. AP_send() tells the library to send the message, but the underlying MPI send is governed by the settings passed to AP_init() or AP_setparam().

AP_free --- free a message buffer

Synopsis

```
#include "autopack.h"

void AP_free(void *buf)

SUBROUTINE AP_FREE(buf)
INTEGER buf
```

Parameters

buf the buffer to free (in Fortran, the descriptor)

Return Value

None

Description

Call this function to free message buffer space allocated by AP_recv().
Note that in the Fortran binding, it is not necessary to call this after AP_COPY_FREE() since that function frees that buffer before returning. However, this function may be of use in case the user wants to free the message without actually accessing it.

AP_init --- initialize the AUTOPACK library

Synopsis

```
#include "autopack.h"

void AP_init(int *argc, char ***argv)

SUBROUTINE AP_INIT
```

Parameters

argc pointer to number of command line arguments
argv pointer to array of command line arguments

Description

This routine must be called prior to any other AUTOPACK calls. The version for C accepts the argc and argv that are provided by the arguments to main(). They may be passed as NULL if no values are available.
During initialization, certain library parameter values will be set to default values. The user may wish to change them using AP_setparam(). In future there will be provision to set the defaults through the command line arguments.
Call this routine after MPI_Init().

Return Value

None

AP_realloc --- Reallocate an outgoing message buffer

Synopsis

```
#include "autopack.h"

void *AP_realloc(void *buf, int newsize)
```

Parameters

buf an outgoing buffer previously allocated by AP_alloc()
newsize the new desired size of the buffer

Description

Still being tested.
Changes the size of the buffer pointed to by buf to newsize bytes and returns a pointer to the (possibly moved) buffer.
The contents will be unchanged up to the lesser of the new and old sizes.
The buffer must be the most recently allocated buffer for the destination, otherwise it is an error. It is also erroneous to specify a buffer which has previously passed to AP_send().
These errors may or may not be detected depending on whether the library was compiled with NO_USER_CHECKS.
If the buffer is reduced in size, it is guaranteed not to be moved. If the buffer is increased in size, it may or may not move depending on available space in the current internal memory block (package).

Return Value

A pointer to the reallocated buffer.

AP_recv --- Receive a message

Synopsis

```
#include "autopack.h"

int AP_recv(int sender, int tag, int flags,
            void **ret_msg, int *ret_size, int *ret_sender, int *ret_tag)

SUBROUTINE AP_RECV(sender, tag, flags, ret_msg, ret_size,
ret_sender, ret_tag, return_value)
INTEGER sender, tag, flags, ret_msg, ret_size, ret_sender,
ret_tag, return_value
INCLUDE 'autopack.fh'
```

Parameters

sender	specifies rank of the message source (may be MPI_ANY_SOURCE)
tag	specifies tag of the message (may be MPI_ANY_TAG)
flags	flags to specify options (see below)
ret_msg	the received message (or in Fortran, a descriptor)
ret_size	size of the received message
ret_sender	sender of the received message
ret_tag	tag of the received message

Return Value

Returns nonzero if a message has been successfully received, otherwise returns 0. If a message has been received, the arguments ret_msg, ret_size, ret_sender, and ret_tag pass back information about the message.
In the Fortran binding, it is not possible to return a pointer to the message in ret_msg. Instead, a descriptor is returned that may be passed to AP_COPY_FREE() to retrieve the message.

Description

Receive a message with given sender and tag (MPI_ANY_SOURCE and MPI_ANY_TAG may be used). If such a message is available, returns nonzero and sets *ret_msg, *ret_size, *ret_sender, and *ret_tag. If no message matching the criteria is found, returns zero.
Any of ret_size, ret_sender, and ret_tag may be passed NULL if the caller is not interested in the return information.
After the caller has processed the information in ret_msg, the function AP_free() must be called to free the buffer space. Failure to do this will result in a memory leak.
Flags may be a bitwise OR of the following (in Fortran, use addition):

AP_NOFLAGS

Default

AP_BLOCKING

Block until a matching message is received (by default, do not block). Will always return 1 unless AP_DROPOUT is specified.

AP_DROPOUT

Used in conjunction with AP_BLOCKING. If some change in status occurs (e.g. an asynchronous reduction message is received) before a message is available, then unblock and return 0.

AP_FIFO

When searching for messages, only look at first incoming message from each source. Default action is to search all incoming messages from a source (then continuing to the next source if MPI_ANY_SOURCE was specified).

If there are deferred sends, the library will try to process them before each attempt to receive a new message from MPI. This is the case whether the call to AP_recv() is blocking or non-blocking,

Efficiency notes

The library will be most efficient when messages are received in the same order they arrive. If all the messages have the same tag, this is not a concern. However, if the incoming messages have a variety of tags, MPI_ANY_TAG will always match the first one and is the most efficient. If a particular tag is specified, use AP_FIFO if the circumstances permit.
The library will be more efficient when a particular source is specified rather than MPI_ANY_SOURCE. If MPI_ANY_SOURCE is specified, sources will be checked in round-robin fashion starting with the rank from which the last message was received. The search is depth-first (although AP_FIFO may truncate the search).

AP_recv_count --- query the result of an asynchronous reduction

Synopsis

```
int AP_recv_count(int *count)


SUBROUTINE AP_RECV_COUNT(count, return_value)
INTEGER count, return_value
```

Parameters

count result of the reduction

Description

Queries the result of the pending asynchronous reduction to determine how many messages from the prior batch of sends were sent to this processor.

Return value

If it returns zero, the asynchronous reduction is not yet complete. The user code must call AP_recv() with the expectation of receiving more messages. If desired, it is safe to block as long as the AP_DROPOUT flag is specified (this flag makes AP_recv() unblock and return 0 if any reduction-related messages are seen). AP_recv_count() can then be queried again. If it returns nonzero, the asynchronous reduction is complete (at least as far as this processor is concerned) and *count is set to the number of messages whose destination is this processor. *It is up to the caller to compare this with the number received so far to determine whether more messages are yet to arrive.* After AP_recv_count() returns nonzero, it is not necessary for this processor to call AP_recv() any longer.

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_reduce_nsends --- start reduction with explicit send count

Synopsis

```
#include "autopack.h"

void AP_reduce_nsends(int *nsends)
```

Parameters

int *nsends - array with count of sends to each destination

Description

Still in testing.
Alternate interface to the asynchronous reduction.

Return Value

None

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_send --- send a message

Synopsis

```
#include "autopack.h"

void AP_send(void *buf)
```

Parameters

buf a message buffer previously allocated via AP_alloc()

Return Value

None

Description

Send a message previously allocated with AP_alloc(). After sending the message, the user may no longer access buf. Parameter values packed, nwait, and nwait_proc (see AP_setparam()) will affect when the send actually takes place. If packing is enabled, the message will not be sent until a full package has been accumulated or the user calls AP_flush(). The send may be also be deferred depending on the current number of sends posted to MPI, and the value of nwait_proc and nwait (see AP_setparam()).
In order for the library to free memory associated with the sends, and to process deferred messages, the user should call AP_check_sends() periodically until the return value indicates no messages are deferred. Calls to AP_recv() will also expedite deferred messages.

Caveats

At present, all messages are sent using type MPI_BYTE.

AP_send_begin --- Initiate a batch of sends

Synopsis

```
#include "autopack.h"

int AP_send_begin(void)

SUBROUTINE AP_SEND_BEGIN(return_value)
INTEGER return_value
```

Parameters

None

Description

Tells the library that the user code is about to begin a batch of sends. The library will then internally keep track of how many sends go to each destination. After doing the sends, call AP_recv_count() to initiate an asynchronous global reduction to determine how many incoming message to expect.
It is an error to call AP_send_begin() again before AP_recv_count() indicates the reduction is complete.
This routine will free space (via AP_check_sends()) from any previous sends that are complete.

Return Value

Returns the number of reductions since the library was initialized. It can be used as a tag for the subsequent sends. When the maximum tag number has been reached, the count will wrap around to 0.

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_send_end --- End a batch of sends, initiate asynchronous reduction

Synopsis

```
#include "autopack.h"

void AP_send_end(void)

SUBROUTINE AP_SEND_END
```

Parameters

None

Description

This function denotes the end of a batch of sends. (To start of a batch of sends, see AP_send_begin().)
After flushing the send buffers (see AP_flush()), this function initiates an asynchronous reduction to compute how many messages this processor will receive from the batch of sends. The result of this reduction may be queried through AP_recv_count().

Caveats

Messages to implement the asynchronous reduction are handled transparently. However, each processor is obligated to call AP_recv() repeatedly until AP_recv_count() has indicated the reduction is complete. Otherwise, the reduction will not complete on this processor (and possibly others).
Messages that implement the asynchronous reduction are always sent immediately and are never deferred. This may use up to 2 MPI send requests per processor which are not counted in the nwait or nwait_proc setting (see AP_setparam()).

Return Value

None

Overview

The general organization of user code should be as follows. Each processor calls AP_send_begin(), then does some sends, then calls AP_send_end(). Then it loops calling AP_recv(), blocking if desired (but using the AP_DROPOUT flag if blocking). AP_recv_count() will indicate when the loop should terminate. It is not necessary to synchronize the processors at any point during this process.

AP_setparam --- Reset the parameters that govern behavior of AUTOPACK

Synopsis

```
#include "autopack.h"

void AP_setparam(int size, int packed, int nwait_proc, int nwait)

SUBROUTINE AP_SETPARAM(size, packed, nwait_proc, nwait)
INTEGER size, packed, nwait_proc, nwait
```

Parameters

size	preferred size in bytes for memory blocks (packages)
packed	if nonzero, enable automatic packing
nwait_proc	max number of MPI sends per destination proc
nwait	max number of MPI sends for all destinations. If nwait<0, use only per-destination limit.

Return Value

None

Description

AUTOPACK parameters may be changed at any time. Changes to size or packed will affect subsequent messages but will not affect any message already allocated. Changes to nwait_proc or nwait will govern subsequent MPI sends. If the limit is reduced, no pending sends are cancelled; the limit may not be attained until some sends complete.

User advice

Size is a guideline; allocation will be made larger if necessary. If it is too small, you will end up with one message per package/memory block. Setting it too large will waste memory and reduce potential overlap of computation/communication.