Chapter 3

Cooperative Message Passing

This chapter describes the CMMD functions used for the cooperative (that is, synchronous) form of point-to-point message passing.

The cooperative message-passing functions are

```
CMMD_send_block{_v}
CMMD_receive_block{_v}
CMMD_send_and_receive{_v}
CMMD_swap
```

These functions synchronize the sending and receiving nodes before sending the data. Once a node has called one of these functions, it "blocks," or waits for its share of the transmission to be complete, before it does anything else. More precisely, the sending node waits until it has injected the last byte of the message into the Data Network; the receiving node waits until it has received into memory the last byte from the Data Network.

These functions can be used in conjunction with each other, or with the asynchronous message-passing functions described in Chapter 5. For example, a message can be transmitted via an asynchronous send plus a cooperative receive.

3.1 Buffers and Arrays

The "messages" transmitted by message-passing functions are arrays (or buffers) of data. Transmission copies the data from some location in the sending node's memory to some location in the receiving node's memory.

Most of the cooperative message-passing functions handle three kinds of arrays: serial arrays, vector serial arrays, and parallel arrays. (Asynchronous functions do not accept vector serial arrays.)

3.1.1 Serial Arrays

As explained in Section 2.3, serial arrays reside in "microprocessor memory" on the node, or (for a host program) on the host.

Serial arrays may be standard C or Fortran variables, arrays, or structures; CM Fortran "FE arrays"; or C* "scalar variables."

Serial arrays are defined to CMMD functions by location (the **`buffer`** argument) and length (the **`buffer_desc`** argument). The length supplied to the function is arbitrary. For example, a call to a send function might supply the well-known variable name **`foo`** as the **`buffer`** argument, and a length of 512 as the **`buffer_desc`** argument. CMMD would then go to the location where storage for **`foo`** began, and copy 512 bytes of memory from that location to the appropriate location on the destination node. It would do this regardless of whether the message thus created contained

- all of **`foo`**
- some fraction of **`foo`**
- all of **`foo`**, all of **`bar`**, and some fraction of **`baz`**

CMMD does not require the lengths of sending and receiving buffers to match; in particular, the receiving buffer may be longer than the sending buffer, to accommodate varying-sized messages.

Serial arrays may be sent to, or received from, other serial arrays, either block serial or vector serial. They may not be sent to, or received from, parallel arrays.

Implementation Notes: Currently, serial buffers must be less than 2^{25} bytes (32 Mbytes) in length. This limit is set by the symbolic limit **`CMMD_BYTE_LIMIT`**, and may change in future releases.

As a rule of thumb, communication operations using doubleword-aligned memory are generally faster than those using word-aligned memory; they, in turn, will be faster than those that use byte-aligned data. In some cases (but not in all), alignment may significantly impact performance.

3.1.2 Parallel Arrays

Parallel arrays reside in VU memory on the nodes. They cannot exist on the host.

Parallel arrays *always* equate to application-defined arrays or structures. Cooperative message-passing functions must always transmit either a complete parallel array or a properly defined array section. Send and receive buffers must be identical. (Asynchronous functions cannot transmit array sections.)

Parallel arrays have *shape* as well as size. An 8 x 8 parallel array is not the same as a 16 x 4 parallel array: one cannot be the destination for the other. (On the other hand, a 512 x 512 array can be the destination for a 512 x 512 section of a 1024 x 1024 array, and vice versa.)

The syntax for functions handling parallel arrays differs according to the language in which the arrays are created and used:

- In CM Fortran, a parallel array represents a CM array. (More precisely, it represents the CMRT descriptor that defines that array.) When you supply a CM array to a messaging function, you use the **`CMMD_PARALLEL_ARRAY`** keyword as the **`buffer_desc`** argument; CMMD then gets all information on array size, shape, type, and location from the CMRT descriptor.

- In C*, a parallel array represents a parallel variable. (More precisely, it represents a pointer to the pvar descriptor that defines that parallel variable.) When you supply a C* parallel variable to a messaging function, you supply the length (in bytes) of a single data element as the **`buffer_desc`** argument, and use the **`void:void`** data type for the **`buffer`** argument (as shown in the syntax statements for the various functions); CMMD then gets all information on array size, shape, type, and location from the pvar descriptor.

- In DPEAC, a parallel array represents a VU data or instruction space address in **`ALL_DPS`** space. In this case, the **`buffer_desc`** argument is simply the length in bytes of the data to be transmitted, just as if the buffer were a serial buffer. The system recognizes the address as being in VU memory, and acts accordingly.

  (See the *DPEAC Reference Manual* and the *CMMD User's Guide* for information on programming in DPEAC and some hints on using DPEAC with CMMD.)

Implementation Note: At Version 3.0, parallel buffers must be smaller than 2^{25} bytes for all VUs, or 2^{23} bytes (8 megabytes) for one VU. (Again, this may change at future releases.)

Programmers may also wish to note more details regarding "serial memory" and "parallel memory." Serial memory is cacheable; parallel memory typically is not. The VUs can access only parallel memory. The microprocessor can access all of memory; however, accesses into VU memory will typically be slower than accesses into cacheable memory, since each access represents an uncached load.

In addition, the microprocessor can access the memory of any single VU as if it were serial memory. Again, however, the access will be slower than an access to serial memory.

Moving Data between Serial and Parallel Memory

CMMD cannot transfer data between serial and parallel memory. If you need to move data between these two types of memory, you must first use language resources (such as CM Fortran Utility Library routines, for CMF) to move the data from one type of memory to another on the source node, and then transmit the data to the destination node.

Array Sections. You can use parallel array sections as arguments to blocking send and receive functions. Let us take a CM Fortran array as an example. The normal CMF calling rules apply. CMF evaluates any array expression, including array sections, into a canonical temporary array; it then passes the temporary array to the calling function or subroutine.

What CMMD sees, therefore, is this temporary array. Suppose a program contains the line

```
CMMD_send_block(2,CMMD_DEFAULT_TAG,
      A(1:64,256:512), CMMD_PARALLEL_ARRAY)
```

An array of 64 x 256 elements is passed to **`CMMD_send_block`**. **`CMMD_send_block`** transfers that array in its entirety and requires a destination array dimensioned 64 x 256 for the receipt of this transfer.

Suppose the matching receive call were

```
CMMD_receive_block(1,CMMD_DEFAULT_TAG,
      B(1:64,1:256), CMMD_PARALLEL_ARRAY)
```

In this case, the message would be received into a temporary buffer created by CM Fortran to hold the array section. When the message arrives, CMMD marks the buffer as modified, causing CMF to modify the original array accordingly.

The creation of these temporary arrays does, of course, consume some resources of time and space; the expense varies according to the geometry and layout involved.

3.1.3 Vector Serial Arrays

A serial array, sent by **`CMMD_send_block`**, begins at the starting place identified by the *buffer* argument, and proceeds for *buffer_desc* sequential bytes. A vector message, sent by **`CMMD_send_block_v`**, takes a number of non-contiguous elements from the buffer, and sends those as a sequential message. (In other words, it performs an implicit gather.) It uses four arguments for information: *buffer*, *elem_count* (the number of elements in the vector), *stride*, and *elem_len.*

Usually, the stride specified for **`CMMD_send_block_v`** is larger than the element length. This difference creates the vector send: *elem_len* bytes are put into the message, then (*stride* minus *elem_len*) bytes are skipped over, then the next *elem_len* bytes are added to the message, and so on, until the specified number of elements has been placed in the message to be sent.

If the stride and element length are specified as being equal, the result is the same as a non-vector send: (*elem_len* * *elem_count*) bytes are sent.

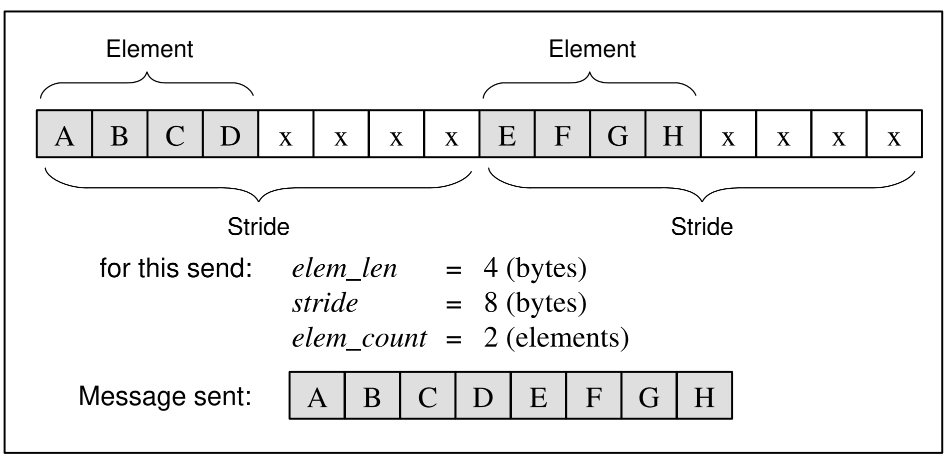


Figure 5. A vectored send.

If the stride is smaller than the element length, **CMMD_send_block_v** sends *elem_len* bytes starting at each stride. For example, a stride of 0 would result in the same element being sent *elem_count* times.

Note that you do not specify the total length of the message in a vectored call. Rather, the length is the result of multiplying the number of elements by the length of each element. Note also that unless the element length and the stride are identical (in which case you are using a vector call to do a standard send), the buffer itself must be longer than the message to be sent from it, since its length must equal the number of elements multiplied by the stride.

As an example of regular and vectored sends, let us consider the case of a 4 x 6 matrix A, filled with 32-bit integers from 0 to 23, laid out in memory as follows:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \end{bmatrix}$$

To send the top row of the matrix as a message to node 5, you would use the call

```
CMMD_send_block (5, CMMD_DEFAULT_TAG, &A, 24)
```

To send the first column of the matrix to node 3, on the other hand, you would need a vector send, stating that you were sending four elements (*elem_count*), each four bytes long (*elem_len*), located twenty-four bytes apart (*stride*).

```
CMMD_send_block_v (3, CMMD_DEFAULT_TAG, &A, 4, 24, 4)
```

Usually, the receiving node would accept the first message with a standard receiving call (**CMMD_receive_block**) and the second with a vector receiving call (**CMMD_receive_block_v**), thus preserving the original geometry of the data. They are not, however, required to do so. Indeed, you could transpose this sample matrix by sending each row as a sequential message, but having each received as a six-element vector.

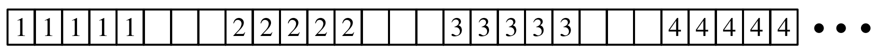
As shown in the example above, **elem_len** and **stride** usually reflect your data element sizes. Note, though, that CMMD does not constrain them to do so. If a function call requests the transfer of partial data elements, CMMD will oblige.

Note, too, that element lengths of sent and received vectors are not necessarily identical. Let us look at some examples of sending and receiving 32-bit integers. (In the diagrams below, a single box represents an integer, 4 bytes long.)

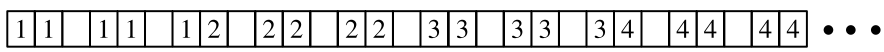
Example 1: *selem_len* > *delem_len*

```
selem_len = 20, sstride = 32,
delem_len = 8,  dstride = 12
```

The source buffer contains



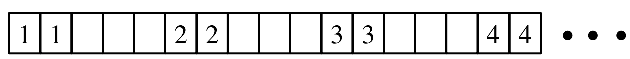
and the destination buffer (after the operation) contains



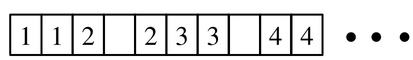
Example 2, *selem_len* < *delem_len*

```
selem_len = 8,  sstride = 20,
delem_len = 12, dstride = 16
```

The source buffer contains



and the destination buffer (after the operation) contains



The placement of the data in the destination buffer thus depends on the relationship between *stride* and *elem_len* in the receive call.

- If *stride* > *elem_len* (the usual case), the elements are placed in the specified buffer at intervals, each separated by (*stride* minus *elem_len*) bytes.
- If *stride* = *elem_len*, then the elements are placed contiguously in the buffer, as for a standard receive.

- If *stride* < *elem_len*, subsequent elements overwrite previous ones where they overlap.

Gathers and Scatters. While **`CMMD_send_block_v`** and **`CMMD_receive_block_v`** are frequently paired, so that data is sent and received with a constant vector striding, it is possible for the source and destination vector striding to be different. For instance, sequential data may be broken into a vector (thus "scattering" the data), or a vector received as sequential (thus "gathering" it). Figure 6 shows the four possible patterns.

Performance and Implementation

Vector send and receive functions are convenience functions only; they do not perform as well as block serial operations.

At Version 3.0, the following limits exist: Element length and stride must each be less than 2^{16} bytes in size; number of elements must be less than 2^{28} bytes long; and total buffer length must be less than 2^{25} bytes long. These limits are set by the symbolic limits **`CMMD_ESIZE_LIMIT`**, **`CMMD_ESTRIDE_LIMIT`**, **`CMMD_ECOUNT_LIMIT`**, and **`CMMD_BYTE_LIMIT`**. They may change in future versions of the library.

At Version 3.0, cooperative vectored transmissions are implemented with temporary arrays. This implementation may change at future releases. Users may find it preferable to create their own temporary arrays, gather data into those, and use the block serial send and receive routines (scattering data again on the receiving node if necessary). This is true particularly if the same temporaries will be used more than once.

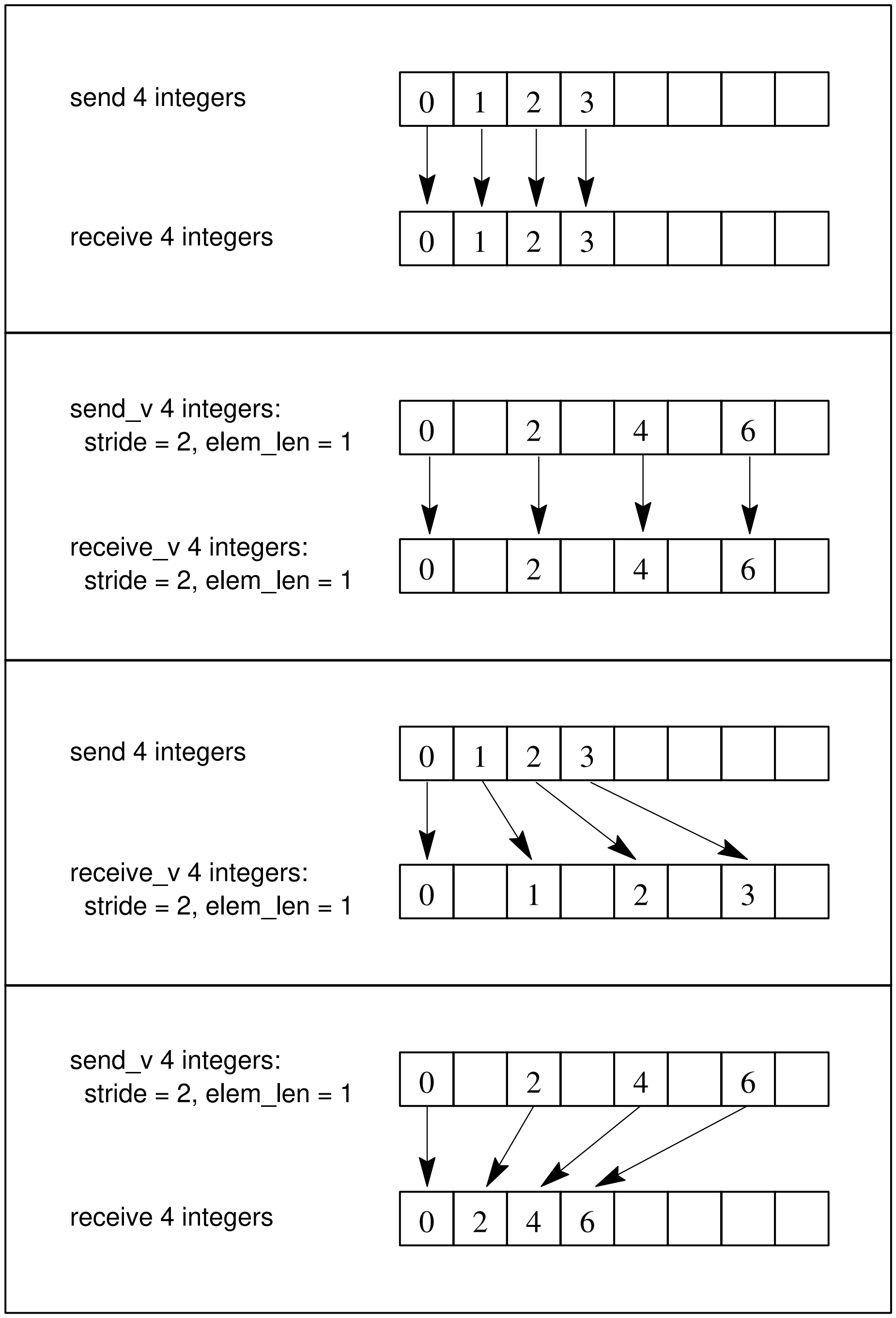


Figure 6. Sending and receiving data.

3.2 Sending a Message

FUNCTIONS **CMMD_send_block**
CMMD_send_block_v

C SYNTAX

int **CMMD_send_block(**int *dest_node*, int *tag,* void **buffer*,
int *buffer_desc***)**

int **CMMD_send_block_v(**int *dest_node*, int *tag*, void **buffer*,
int *elem_len*, int *stride*, int *elem_count***)**

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_send_block(***dest_node, tag, buffer,*
*buffer_desc***)**
integer *dest_node*, *tag*, *buffer_desc*
<array> *buffer*

INTEGER FUNCTION **CMMD_send_block_v(***dest_node, tag, buffer,*
elem_len, stride,
*elem_count***)**
integer *dest_node, tag*, *elem_len*, *stride*, *elem_count*
<array> *buffer*

C* SYNTAX, for Parallel Arrays Only

int **CMMD_send_block** (int *dest_node*,
void:void **buffer*,
int *array_element_length*)

ARGUMENTS

dest_node An integer identifying the node to which the message is to be sent.

tag A positive integer that serves as a label for the message, or **CMMD_DEFAULT_TAG**.

buffer The area in memory that contains the message to be sent.

buffer_desc	(Non-vector sends only.) A buffer descriptor. For serial arrays, or parallel variables in DPEAC, a non-negative integer specifying the length of the buffer, in bytes. For CM Fortran parallel arrays, the keyword **`CMMD_PARALLEL_ARRAY`**. For C* parallel arrays, the length in bytes of a single data element.
elem_len	(Vector sends only.) An integer specifying the length of each element in the vector.
stride	(Vector sends only.) An integer specifying the distance in bytes between the starting addresses of vector elements. This should be either a multiple of *elem_len*, or 0.
elem_count	(Vector sends only.) An integer specifying the number of elements in the vector.

RETURNS

These functions do not return until the message has been transmitted. Return values are as follows:

Return Value	Meaning
0	The transfer succeeded; the full message was sent.
1	Failure — not all array elements were sent.
`CMMD_ERRVAL`	An error occurred.

See Section 4.3 for a description of the accessor functions you can use to determine how many bytes of a message were actually sent.

DESCRIPTION

The **`CMMD_send_block`** functions send the contents of a specified buffer to the given destination node. (A buffer length of 0 sends a zero-length message.) The node must be inside the partition; otherwise, an error results. The message may be identified by a user-supplied tag; the symbol **`CMMD_DEFAULT_TAG`** is the standard default tag.

The transfer does not take place until the receiving node has made a call to one of the CMMD receive functions, indicating that it is ready to receive the message. The function blocks until the transfer takes place.

Buffers. Serial arrays must be sent to serial buffers, parallel arrays to parallel buffers. If you need to transfer data between serial and parallel arrays, you must do the transformation and the transfer in two separate steps.

For parallel arrays, source and destination buffers must be identical in size and shape.

Buffers should not overlap other buffers used in communication operations.

For more detailed information on buffers, see Section 3.1.

Vector Sends. A serial array, sent by **`CMMD_send_block`**, begins at the starting place identified by the *buffer* argument, and proceeds for *buffer_desc* sequential bytes. A vector message, sent by **`CMMD_send_block_v`**, takes a number of non-contiguous elements from the buffer, and sends those as a sequential message. (In other words, it performs an implicit gather.)

Ordinarily, the stride specified for **`CMMD_send_block_v`** is larger than the element length. This difference creates the vector send: *elem_len* bytes are put into the message, then (*stride* minus *elem_len*) bytes are skipped over, then the next *elem_len* bytes are added to the message, and so on, until the specified number of elements has been placed in the message to be sent.

If the stride and element length are specified as being equal, the result is the same as a non-vector send: (*elem_len* * *elem_count*) bytes are sent.

If the stride is smaller than the element length, **`CMMD_send_block_v`** sends *elem_len* bytes starting at each stride. For example, a stride of 0 would result in the same element being sent *elem_count* times.

Note that you do not specify the total length of the message in a vectored call. Rather, the length is the result of multiplying the number of elements by the length of each element. Note also that unless the element length and the stride are identical (in which case you are using a vector call to do a standard send), the buffer itself must be longer than the message to be sent from it, since its length must equal the number of elements multiplied by the stride.

Handlers. **`CMMD_send_block`** and **`CMMD_send_block_v`** cannot use handlers and cannot be called by handler functions.

3.3 Receiving a Message

FUNCTIONS **`CMMD_receive_block`**
`CMMD_receive_block_v`

C SYNTAX

int **`CMMD_receive_block(`**int *source_node*, int *tag*, void **buffer*,
int *buffer_desc***`)`**

int **`CMMD_receive_block_v(`**int *source_node*, int *tag*, void **buffer*,
int *elem_len*, int *stride*, int *elem_count***`)`**

FORTRAN SYNTAX

INTEGER FUNCTION **`CMMD_receive_block(`***source_node, tag, buffer,*
*buffer_desc***`)`**
integer *source_node, tag, buffer_desc*
<array> *buffer*

INTEGER FUNCTION **`CMMD_receive_block_v(`***source_node, tag,*
buffer, elem_len, stride,
*elem_count***`)`**
integer *source_node, tag, elem_len, stride, elem_count*
<array> *buffer*

C* SYNTAX, for Parallel Arrays Only

int **`CMMD_receive_block`** **`(`**int *source_node*,
int *tag*,
void:void **buffer*,
int *array_element_length***`)`**

ARGUMENTS

source_node An integer identifying the node from which the message is to be sent **`(CMMD_ANY_NODE`** allows any node to be the sender).

tag A positive integer, or **`CMMD_DEFAULT_TAG`**, that serves as a label for the message; or, **`CMMD_ANY_TAG`**, which allows receipt of a message labeled with any tag.

buffer	The area in memory that will contain the message to be received.
buffer_desc	(Buffer descriptor, non-vector sends only.) A non-negative integer defining the number of bytes in the receive buffer, or the keyword **CMMD_PARALLEL_ARRAY**, for a CM Fortran parallel array; or the length of a single data element, for a C* parallel array.
elem_len	(Vector function only.) An integer specifying the length of each element in the vector, in bytes. Should be a multiple of the appropriate data type.
stride	(Vector function only.) An integer specifying the distance in bytes between the starting addresses of the vector elements. Should be either a multiple of *elem_count*, or 0.
elem_count	(Vector function only.) An integer specifying the number of elements in the vector.

RETURNS

These functions return only after the message has been received and copied into the specified buffer. They return an integer value, as follows:

Return Value	Meaning
0	The transfer succeeded; full message was received.
1	Failure — not all array elements were received.
CMMD_ERRVAL	An error occurred.

If the return value is 1, your program can call **CMMD_bytes_received()** (see Section 4.3) to get the number of bytes actually received. Note that if the number of bytes actually received is less than the number requested, the excess portion of the receiver buffer is left in an undefined state.

DESCRIPTION

The **CMMD_receive_block** functions post a receive buffer as being available to receive a message of a particular length or array type, with a specified tag; they then wait for a message with the specified tag to be sent from the specified source. These routines can take the special symbol **CMMD_ANY_NODE** as the source argument, indicating that any source is acceptable, and the symbol **CMMD_ANY_TAG** as the tag argument, indicating that any tag will be accepted.

If **CMMD_ANY_NODE** is given, the program can call the function **CMMD_msg_sender()** to get the node identifier of the actual sender; if **CMMD_ANY_TAG** is used, **CMMD_msg_tag()** can be called to get the tag of the accepted message. (See Section 4.3.)

Upon transmission, data is received into the specified buffer.

As with the **CMMD_send_block** routines, the buffers must not overlap with other buffers being used in communications functions. Also, parallel arrays must be identical in size and shape on sending and receiving nodes.

Vector Messages. **CMMD_receive_block** may be used when incoming data is to be stored in non-contiguous memory locations. The call specifies that the information to be received be considered a vector of *elem_count* elements, each *elem_length* bytes long, each element to be placed in an area of the buffer that is *stride* bytes in length.

The placement of the data in the buffer thus depends on the relationship between *stride* and *elem_len*.

- If *stride* > *elem_len* (the usual case), the elements are placed in the specified buffer at intervals, each separated by (*stride* minus *elem_len*) bytes.
- If *stride* = *elem_len*, then the elements are placed contiguously in the buffer, as for a standard receive.
- If *stride* < *elem_len*, subsequent elements overwrite previous ones where they overlap.

Handlers. **CMMD_receive_block** and **CMMD_receive_block_v** can neither call handler functions nor be called by handler functions.

3.4 Simultaneous Sends and Receives

FUNCTIONS **CMMD_send_and_receive**
CMMD_send_and_receive_v

C SYNTAX

int **CMMD_send_and_receive(**
int *source_node*, int *source_tag*, void **inbuffer*, int *inbuf_desc*,
int *dest_node*, int *dest_tag*, void **outbuffe*r, int *outbuf_desc***)**

int **CMMD_send_and_receive_v(**
int *source_node*, int *source_tag*, void **inbuffer*,
int *in_elem_len*, int *in_stride*, int *in_elem_count*,
int *dest_node*, int *dest_tag*, void **outbuffer*,
int *out_elem_len*, int *out_stride*, int *out_elem_count***)**

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_send_and_receive(**
source_node, source_tag, inbuffer, inbuf_desc,
*dest_node, dest_tag, outbuffer, outbuf_desc***)**
integer *source_node, source_tag, inbuf_desc,*
integer *dest_node, dest_tag, outbuf_desc*
<array> *inbuffer, outbuffer*

INTEGER FUNCTION **CMMD_send_and_receive_v(**
source_node, source_tag, inbuffer,
in_elem_len, in_stride, in_elem_count,
dest_node, dest_tag, outbuffer,
*out_elem_len, out_stride, out_elem_count***)**
integer *source_node, source_tag*
integer *in_elem_len, in_stride*, *in_elem_count*
integer *dest_node, dest_tag*
integer *out_elem_len*, *out_stride, out_elem_count*
<array> *inbuffer, outbuffer*

C* SYNTAX, for Parallel Arrays Only

int **`CMMD_send_and_receive(`**
source_node, int *source_tag*,
void:void **in_buffer*, int *inbuf_desc*,
int *dest_node*, int *dest_tag*,
void:void **outbuffer*, int *outbuf_desc***`)`**

ARGUMENTS

source_node — An integer identifying the node from which the incoming message will be received.

source_tag — A positive integer, or **`CMMD_DEFAULT_TAG`**, labeling the message to be received; or, **`CMMD_ANY_TAG`**.

inbuffer — The area in memory that will contain the message to be received.

inbuf_desc — (Non-vector function only.) The length, in bytes, of the buffer that will hold the message received by this node; or, for CMF parallel arrays, the keyword **`CMMD_PARALLEL_ARRAY`**.; or, for C* parallel arrays, the length, in bytes, of a single data element.

in_elem_len — (Vector function only.) Length, in bytes, of each element in the vector to be received by this node. Should be a multiple of the size of the relevant data type.

in_stride — (Vector function only.) Number of bytes between starting addresses of elements in the vector that comprises the message to be received by this node. Should be a multiple of *in_elem_len*, or 0.

in_elem_count — (Vector function only.) Number of elements that comprise the vector to be received by this node.

dest_node — An integer identifying the node to which this node will send a message.

dest_tag — A positive integer labeling the message that will be sent by this node.

outbuffer — The area in memory holding the message to be sent by this node.

outbuf_desc	(Non-vector function only.) The length, in bytes, of the buffer that will hold the message sent by this node; or, for CMF parallel arrays, the keyword **`CMMD_PARALLEL_ARRAY`**.; or, for C* arrays, the length, in bytes, of a single data element.
out_elem_len	(Vector functions only.) Length, in bytes, of each element in the vector to be sent by this node. Should be a multiple of the appropriate data type size.
out_stride	(Vector functions only.) Number of bytes by which starting addresses of elements in the vector to be sent are separated. Should be a multiple of *out_elem_len*, or 0.
out_elem_count	(Vector function only.) Number of elements that comprise the vector to be sent by this node.

RETURNS

`CMMD_send_and_receive` and **`CMMD_send_and_receive_v`** do not return until the calling node has sent one message and received one. They return integer values, as follows:

Return Value	Meaning
0	The transfer succeeded; full message was sent.
1	Failure — not all bytes were sent or received.
`CMMD_ERRVAL`	An error occurred.

If the return value is 1, the application can call **`CMMD_bytes_sent()`** (see Section 4.3) to get the number of bytes actually sent, and **`CMMD_bytes_received()`** to get the number of bytes actually received. Note that if the number of bytes actually received is less than the number requested, the excess portion of the receiver buffer is left in an undefined state.

DESCRIPTION

These two functions cause nodes to send and receive messages simultaneously. Any number of nodes can take part in one of these calls; the only requirement is that each node must both send a message and receive a message. (See **`CMMD_swap`** and **`CMMD_swap_v`** for a simpler way to send and receive simultaneously when two nodes are involved, each serving as both source and destination for the other.)

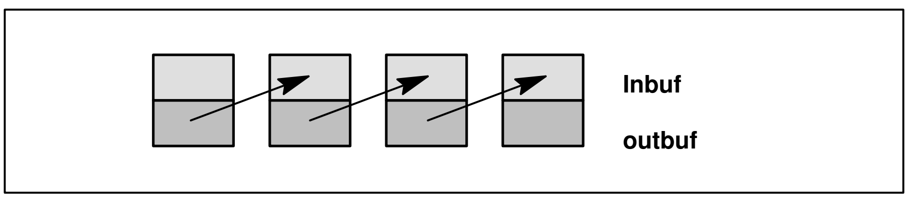
The functions cause the message in the calling node's outbuffer to be passed to the destination node at the same time that a message is read into the calling node's inbuffer from the source node. Buffers may not overlap with other buffers being used in communication functions.

NOTES

1. In vector operations, the strides for sent and received messages do not have to be equal.

2. Sends and receives may be mixed with **`send_and_receive`** functions. For example, you might mix these calls in order to create an end-off shift on four nodes:

   Node 0: **`CMMD_send_block`**: uses boundary value, sends to node 1
   Node 1: **`CMMD_send_and_receive`**: receives from 0, sends to 2
   Node 2: **`CMMD_send_and_receive`**: receives from 1, sends to 3
   Node 3: **`CMMD_receive_block`**: receives from 2



Handlers. These functions may not call or be called by handler functions.

3.5 Two-Node Message Swapping

FUNCTIONS **`CMMD_swap`**
`CMMD_swap_v`

C SYNTAX

int **`CMMD_swap(`**int *partner*, void **inbuffer*, int *inbuf_desc*,
void **outbuffer*, int *outbuf_desc***`)`**

int **`CMMD_swap_v(`**int *partner*, void **inbuffer*,
int *in_elem_len*, int *in_stride*, int *in_elem_count*,
void **outbuffer*, int *out_elem_len*, int *out_stride*,
int *out_elem_count*)

FORTRAN SYNTAX

INTEGER FUNCTION **`CMMD_swap(`***partner, inbuffer, inbuf_desc,*
*outbuffer, outbuf_desc***`)`**
integer *partner, inbuf_desc, outbuf_desc*
<array> *inbuffer, outbuffer*

INTEGER FUNCTION **`CMMD_swap_v(`***partner, inbuffer,*
in_elem_len, in_stride, in_elem_count,
outbuffer, out_elem_len, out_stride,
*out_elem_count***`)`**
integer *partner, in_elem_len, in_stride, in_elem_count*
integer *out_elem_len, out_stride, out_elem_count*
<array> *inbuffer, outbuffer*

C* SYNTAX, for Parallel Arrays Only

int **`CMMD_swap(`**int *partner,* void:void **inbuffer*, int *inbuf_desc*,
void:void **outbuffer*, int *outbuf_desc***`)`**

ARGUMENTS

partner An integer identifying the node with which a message is to be swapped.

inbuffer The buffer that will hold the received message.

inbuf_desc (**CMMD_swap** only.) The length, in bytes, of the buffer that will hold the message received by this node; or, for CMF parallel arrays, the keyword **CMMD_PARALLEL_ARRAY**.; or, for C* parallel arrays, the length, in bytes, of a single data element.

in_elem_len (**CMMD_swap_v** only.) Length, in bytes, of each element in the vector to be received by this node. Should be a multiple of the relevant data type size.

in_stride (**CMMD_swap_v** only.) Number of bytes between starting addresses of elements in the vector to be received by this node. Should be a multiple of *in_elem_len*, or 0.

in_elem_count (**CMMD_swap_v** only.) Number of elements that comprise the vector to be received by this node.

outbuffer The buffer holding the message to be sent by this node.

outbuf_desc (**CMMD_swap** only.) For serial arrays, the length, in bytes, of the buffer that will hold the message received by this node; or, for CMF parallel arrays, the keyword **CMMD_PARALLEL_ARRAY**.; or, for C* parallel arrays, the length, in bytes, of a single data element.

out_elem_len (**CMMD_swap_v** only.) Length, in bytes, of each element in the vector to be sent by this node. Should be a multiple of the relevant data type size.

out_stride (**CMMD_swap_v** only.) Number of bytes by which starting addresses of elements in the vector to be sent are separated. Should be a multiple of *out_elem_len*, or 0.

out_elem_count (**CMMD_swap_v** only). Number of elements that comprise the vector to be sent by this node.

DESCRIPTION

CMMD_swap is identical to **CMMD_send_and_receive** (and **CMMD_swap_v** to **CMMD_send_and_receive_v**) where the source node equals the destination node. Note that users cannot specify tags for swaps. The swap functions automatically use **CMMD_DEFAULT_TAG**.

For an explanation of serial versus vector functions, see **CMMD_send_block**.

RETURNS

CMMD_swap returns integer values, as follows:

Return Value	Meaning
0	The transfer succeeded; full message was sent.
1	Failure — not all bytes were sent or received.
CMMD_ERRVAL	An error occurred.