Chapter 5

Asynchronous Message Passing

This chapter describes the CMMD operators used for the asynchronous form of point-to-point message passing, in which the sending and receiving nodes do not have to synchronize to transmit a message.

The asynchronous message-passing functions in CMMD are:

`CMMD_send_async` **`CMMD_receive_async`**
`CMMD_send_noblock`

These functions allow one node to signal readiness to transmit a message, then continue other computations until the second node honors the request. When both nodes are ready, the ongoing computation is interrupted. At this point, the message is sent and received.

By default, message transmission is driven by the CMMD interrupt handler, acting independently of the application. If users wish to substitute application-specified polling for interrupts during some or all phases of their program, they can do so. The mechanism is explained in Chapter 10.

5.1 Asynchronous Message Passing

The two asynchronous send functions differ in how they handle data:

- **`CMMD_send_async`** sends whatever is in the buffer at the time of transmission. It is the application's responsibility to ensure that those contents remain unchanged between the time the node calls the function and the time it actually sends the data.

- If **CMMD_send_noblock** cannot send its message immediately, it writes data into a temporary buffer, then sends that buffer. This imposes a memory space penalty, but frees the original buffer for immediate re-use.

In addition, **CMMD_send_async** and **CMMD_receive_async** differ from **CMMD_send_noblock** and from the blocking message-passing functions in two other ways:

- They use explicit message control blocks (MCBs) for status tracking.
- They can invoke handler functions.

All three asynchronous functions differ from the blocking functions in that

- They do not accept vector serial arrays.
- They do not accept parallel array sections. **CMMD_send_async** and **CMMD_receive_async** accept only complete parallel arrays. **CMMD_send_noblock** does not accept parallel arrays at all.

In other respects, the characteristics of serial and parallel buffers and buffer descriptions are the same as those for the blocking functions, as described in Section 3.1.

5.1.1 Asynchronous Protocol and MCBs

When an asynchronous message-passing function is called to send (or receive) a message, it returns the integer ID of a *message control block* (MCB), a CMMD internal data structure that tracks the status of the transmission. Thus, a program can call one of the asynchronous functions, receive an MCB ID, and then simply check the MCB's fields periodically to see whether the transmission has been completed.

IMPORTANT

MCBs are a limited resource. They must be freed once the message is delivered, or the program may run out of them, in which case it will crash if it tries to allocate new MCBs. The function **CMMD_free_mcb** is used to free an MCB.

5.2 Handler Functions

The **`CMMD_send_async`** and **`CMMD_receive_async`** functions both take handler function arguments. Handler functions are user-written functions that are invoked upon completion of a message transmission. A handler function may perform computations, asynchronous communications, or both. For example, a handler function can be used to free the MCB involved in the transmission, to set a flag indicating the transmission is complete, and so on. A handler function can also initiate another message-passing communications operation. For example, it can send a reply to a received message.

Handler functions are asynchronous. Therefore, they cannot invoke any blocking functions (such as **`CMMD_send_block`** and **`CMMD_receive_block`**). Other restrictions also exist; see Section 11.3.1 for a full discussion of restrictions on handler functions.

Example: Assume that an application has laid out a 2D grid across all the nodes. A single node might need to collect data from its four nearest neighbors. Rather than initiate a two-way exchange of data with each of its neighbors, the handler might send a request message to only one of them. The receiving node could use a handler function to pass the request, along with its contribution of data, along to the next of the four neighbors, and so on around the chain of neighbors. (See Figure 7.)

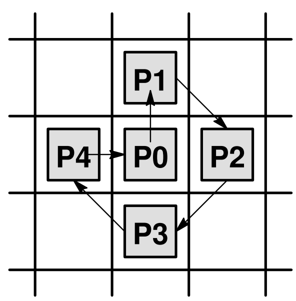

Figure 7. Handler functions can be used to implement simple "stencil" patterns.

The last neighbor in the chain then delivers the collected data back to the source node. While this might seem a somewhat roundabout way of doing things, in some cases it can produce better performance than the more obvious solution of polling all four nodes individually.

5.2.1 Handler Functions in C and Fortran

A handler function must take two arguments. The first argument is either:

- (in C) a pointer to the message control block ID of the asynchronous message
- (in Fortran) the message control block ID itself.

Note the extra level of indirection in C. In C programs, the first argument of a handler function is a *pointer* to an MCB. In Fortran programs, however, the first argument is the actual *integer* ID of an MCB. This is because in Fortran, arguments are passed by reference. By making the first argument a pointer in C handlers, CMMD can use the same internal interface for both C and Fortran.

The second argument to a handler function is the handler argument supplied in the original call to **`CMMD_send_async`** and **`CMMD_receive_async`**. If no such data structure is needed, the value of the second argument should be 0.

5.2.2 Restrictions on Asynchronous Messages

`CMMD_send_noblock` takes only block serial array arguments. It does not accept either vector serial arrays or parallel arrays.

`CMMD_send_async` and **`CMMD_receive_async`** only allow block serial or parallel array arguments. Vector serial arrays must be copied into block serial temporary arrays for transmission. CM Fortran parallel array sections must also be copied to temporaries for sending; see the description of **`CMMD_send_async`** for more information.

Note, moreover, that parallel array operations are not supported within handlers. If a CM Fortran or a C* program calls a handler function from **`CMMD_send_async`** or **`CMMD_receive_async`**, the handler should not perform any parallel array operations, nor call any CMF or C* statement that manipulates parallel data.

The asynchronous message-passing functions can be used only for one-way transmission between two nodes; there is no asynchronous analog to the **`CMMD_send_and_receive`** operator. For patterns of communication in which more than two nodes need to be involved, handler functions can be used to sequence communications sent between a number of nodes; when a node finishes transferring data to one destination, its handler function can make the next message-passing call.

5.3 Program Termination

A program that uses asynchronous messages must make sure that all messages have been sent and received before the program returns or exits.

For this reason, such programs should call either **`CMMD_all_msgs_wait`** or **`CMMD_all_msgs_done`** (making sure they get a response of **`TRUE`** in the latter case) before allowing the program to end.

5.4 Asynchronous Send Function

FUNCTION **`CMMD_send_async`**

C SYNTAX

CMMD_mcb **`CMMD_send_async(`**int *dest_node*, int *tag*, void **buffer*,
int *buffer_desc*,
void (**handler*)(CMMD_mcb **mcb*, void **handler_arg*)
void **handler_arg*)

FORTRAN SYNTAX

INTEGER FUNCTION **`CMMD_send_async(`***dest_node, tag, buffer,*
buffer_desc, handler,
handler_arg)
integer *dest_node, tag, buffer_desc*
<array> *buffer*
SUBROUTINE *handler(mcb, handler_arg)*
integer *mcb*
<array> *handler_arg*

SUBROUTINE *handler(mcb, handler_arg)*
integer *mcb*
<array> *handler_arg*

C* SYNTAX, for Parallel Arrays Only

int CMMD_send_async(int *dest_node*,
int *tag*,
void:void **buffer*,
int *array_element_length*,
void (**handler*)(CMMD_mcb **mcb*, void **handler_arg*),
void **handler_arg***)**

ARGUMENTS

dest_node — The node to which the message is to be sent.

tag — The user tag for the message. Can be any positive 32-bit integer, or **CMMD_DEFAULT_TAG**.

buffer — The location in memory at which the data that constitutes the message begins.

buffer_desc — For serial arrays, a non-negative integer that specifies the number of bytes to be sent. For CM Fortran parallel arrays, the keyword **CMMD_PARALLEL_ARRAY**. For C* parallel arrays, the length, in bytes, of a single data element.

handler(mcb, handler_arg)
A user-supplied function (or **NULL**), with arguments:

- *mcb* — In C, a pointer to the message control block ID returned by the **CMMD_send_async** function. In Fortran, the MCB ID itself.
- *handler_arg* — A user-defined data structure passed to the handler function. If no data structure is needed, should be **NULL** or 0.

RETURNS

The function returns as soon as it has queued its RTS (request to send). The function returns the integer ID of a message control block (MCB). The MCB tracks the state of the transfer, including whether the transfer has completed, whether it generated errors, and how many bytes were actually transferred.

Return values are:

Return Value	Meaning
0 or positive integer	The MCB attached to this end of this transmission.
CMMD_ERRVAL	No MCBs were available for this transmission.

DESCRIPTION

CMMD_send_async initiates sending the contents of the specified buffer, tagged with the specified tag, to the given destination node. The tag may be **CMMD_DEFAULT_TAG**.

CMMD_send_async returns the integer ID of a message control block (**CMMD_mcb**) when it has queued the message to be sent. Thus, it returns almost immediately. The user program then has two responsibilities:

- The user program must maintain the buffer in its correct state until the message has actually been sent and received.
- The user program must free the **CMMD_mcb** when it is no longer needed (that is, when the message has been sent and received).

The program can define a handler routine to be invoked when the message has been sent. To define a handler for a particular message, give the handler function's name as the handler argument to the **CMMD_send_async** function. The handler will be invoked when the message is sent.

If the program does not provide a handler function, it can still query the message's MCB for information and free the MCB, using the auxiliary functions described in Chapter 6.

Handlers. The handler function takes two arguments: a pointer to the MCB and the user-supplied *handler_arg*. (Note that in Fortran handlers the MCB argument is the actual integer ID.)*

* In CMMD Version 2.0, handler functions could only take one argument, the MCB. They now take two arguments. The lack of a second argument to the handler function under 3.0 will cause ANSI compilers to report an error. To avoid this, add a 0 as a second argument to the function. Non-ANSI compilers probably will not complain; they will just pass garbage as the second argument, which the handler function will quite properly ignore.

The handler function may perform computations or asynchronous communications. It may also free the MCB for the calling function.

Handlers may not assume that any function they invoke will complete before the handler itself returns. They may not invoke blocking communication functions, nor call any CMF or C* functions that manipulate parallel data. (For a complete list of restrictions, see Section 11.3.1.)

If the result of a call to **`CMMD_send_async`** or **`CMMD_receive_async`** is assigned to a global variable, it is not guaranteed that the assignment will be made before the handler function is called; therefore, the handler function should refer to only the MCB passed into it, and not refer to the (possibly unassigned) global variable.

Buffers. If the receiving node does not know the length of the message to be sent to it, it can specify an oversize buffer length and accept as many (or as few) bytes as the sender desires to send. The unused portion of the buffer is then in an undefined state.

Buffers may not overlap with other buffers that are being used in communication operations.

CM Fortran Array Expressions. Whenever a CM Fortran array expression is used to define a buffer for **`CMMD_send_async`**, the program must take the following steps:

1. Evaluate the expression into a temporary buffer.
2. Pass the temporary buffer to the function.
3. Poll for completion of the operation.
4. Copy any results back into the final destination.

The reason for this is as follows: When an array expression is passed into a function, the CM Fortran compiler first evaluates the expression into a temporary array, and then passes the temporary array into the function. When the function returns, if the temporary array is marked as modified, it is spread back into the final destination. The temporary array is then deallocated.

Now assume that the function involved is an asynchronous send. The function is called, creates and evaluates the temporary, returns with an MCB, and proceeds to deallocate the temporary. Some time later, the receiving node signals that it is ready to accept the message — but the temporary array to be used in transmission no longer exists! To avoid this scenario, your programs must create and handle their own temporary buffers for the asynchronous function.

5.5 Asynchronous Receive Function

FUNCTION **CMMD_receive_async**

C SYNTAX

```
CMMD_mcb CMMD_receive_async(int source_node, int tag,
            void *buffer, int buffer_desc,
            void (*handler)(CMMD_mcb *mcb,
            void *handler_arg),
            void *handler_arg)
```

FORTRAN SYNTAX

```
INTEGER FUNCTION CMMD_receive_async(source_node, tag, buffer,
                                    buffer_desc,  handler, mcb
                                    handler_arg)
        integer source_node, tag, buffer_desc
        <array> buffer
        SUBROUTINE handler (mcb, handler_arg)
                integer mcb
                <array> handler_arg
```

```
SUBROUTINE handler(mcb, handler_arg)
        integer mcb
        <array> handler_arg
```

C* SYNTAX, for Parallel Arrays Only

```
int CMMD_receive_async(int source_node,
                       int tag,
                       void:void *buffer,
                       int array_element_length,
                       void (*handler)(CMMD_mcb *mcb, void *handler_arg),
                       void *handler_arg)
```

ARGUMENTS

source_node The node from which the message is to be received; can be **CMMD_ANY_NODE**.

tag The user tag for the message to be received: can be any positive integer, or **CMMD_DEFAULT_TAG**, or **CMMD_ANY_TAG**.

buffer The area in memory that will hold the received message.

buffer_desc For serial arrays, a non-negative integer that specifies the number of bytes to be sent. For CM Forran parallel arrays, the keyword **CMMD_PARALLEL_ARRAY**. For C* parallel arrays, the length, in bytes, of a single data element.

*(*handler)(mcb, handler_arg)*
A user-supplied function (or **NULL**); see description below.

mcb In C, a pointer to the message control block ID returned by the **CMMD_receive_async** function. In Fortran, the MCB ID itself.

handler_arg A user-defined data structure to be used by the handler function. If no data structure is required, this argument should be **NULL** or 0.

RETURNS

The function returns as soon as it has created an MCB describing the message to be received. The function returns a pointer to a message control block (MCB). The MCB tracks the state of the transfer, including whether the transfer has completed, whether it generated errors, and how many bytes were actually transferred.

Return values are:

Return Value	Meaning
0 or positive integer	The MCB attached to this end of this transmission.
CMMD_ERRVAL	No MCBs were available for this transmission.

DESCRIPTION

CMMD_receive_async "posts" (i.e., marks as available) a buffer to hold a message from the specified source processor, and immediately returns a pointer to a message control block.

The user program may call MCB query functions (described in Chapter 6) to determine when the message has been received. A user-supplied routine, *handler*, may also be provided. This function is invoked when the message has actually arrived; it receives a pointer to the message's MCB as input. (Handlers are described in Section 5.2.)

Note that if the result of a call to **CMMD_send_async** or **CMMD_receive_async** is assigned to a global variable, it is not guaranteed that the assignment will be made before the handler function is called; therefore, the handler function should refer only to the MCB passed into it, and not refer to the (possibly unassigned) global variable.

Buffers. If the receiving node does not know the length of the message to be sent to it, it can specify an oversize buffer length and accept as many (or as few) bytes as the sender desires to send. If the number of bytes sent is less than the number requested, the excess portion of the receiver buffer is left in an undefined state. To determine the length of the message actually received, the program can call **CMMD_mcb_len** (described in the next chapter).

Buffers may not overlap with other buffers that are being used in communication operations.

5.6 Non-Blocking Send Function

FUNCTIONS **CMMD_send_noblock**

C SYNTAX

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

FORTRAN SYNTAX

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

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 buffer that contains the message to be sent.
buffer_desc	A non-negative integer defining the number of bytes to be sent from the serial buffer.

RETURNS

CMMD_send_noblock returns as soon as it has either sent its message or copied it into a temporary buffer. Return values are:

Return Value	Meaning
0	The message was either transferred in full, or copied to a temporary buffer for later sending.
CMMD_ERRVAL	An error occurred.

DESCRIPTION

CMMD_send_noblock sends the contents of a serial buffer of specified length, tagged with the specified tag (which may be the default tag, **CMMD_DEFAULT_TAG**), to the given destination node. The destination node must be inside the partition; otherwise, an error results. **CMMD_send_noblock** cannot handle either parallel arrays or vector serial arrays.

If **CMMD_send_noblock** cannot send its message immediately, it copies the data into a temporary buffer, queues that buffer for later transmission, and returns. CMMD then waits for a receiving node to declare its readiness to accept the message; at that time, it interrupts the sending node, so that the node can transmit the message from the temporary buffer. (Note that, in Version 3.0, as in previous releases, the receiving node can agree to accept fewer bytes than are in the buffer; **CMMD_send_noblock** has no way of reporting when this occurs. Note also that this may be reported as an error; in future releases, it will be an error and may prevent the message from being sent.)

Buffers. Issues of buffer size and alignment are the same for the **send_noblock** functions as for the **send_block** functions, with one exception: there are no overlap issues with the **send_noblock** functions.

Performance. **CMMD_send_noblock** makes an internal copy of the message argument, which can degrade performance; thus it should be used only in cases in which the source buffer must be available for re-use as soon as possible.

Handlers. **CMMD_send_noblock** cannot call or be called from any handler function.