Chapter 6

Asynchronous Message Passing

This chapter describes the CMMD functions 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. This functionality allows the user to "overlap communication with computation" or to construct communication patterns that would otherwise cause deadlock (see Chapter 2).

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 or receive a message, then continue other computations until the second node honors the request. When both nodes are ready, the message is transmitted in the background.

By default, message transmission is driven by the CMMD interrupt handler, acting independently of the application. Users can, however, substitute application-specified polling for interrupts during some or all phases of their program, as shown in Section B.3.1 of Appendix B (and explained in more detail in Chapter 11).

6.1 Asynchronous Message Passing

Asynchronous message passing often demands more tracking than synchronized message passing. `CMMD_send_async` and `CMMD_receive_async` provide two mechanisms for tracking messages:

Copyright © 1995 Thinking Machines Corporation

- They make explicit their use of message control blocks (MCBs), allowing the application to query the MCBs for status information.
- They can invoke callback functions, which execute on completion of the send or receive. (See Section 6.3 for a discussion of callbacks.)

`CMMD_send_noblock` cannot use these mechanisms. Its distinguishing feature is that it automatically buffers its data, thus allowing immediate reuse of the original buffer area. If you want to know when the data has been received, you can pair `CMMD_send_noblock` with `CMMD_receive_async`, and have the receiving node use a callback function to notify the sender of the message's arrival (via an active message, for example).

Note that the asynchronous message-passing functions can be used only for one-way transmission between two nodes; there is no asynchronous version of the `CMMD_send_and_receive` function. For patterns of communication in which more than two nodes need to be involved, callback functions can be used to sequence messages sent between a number of nodes: when a node finishes transferring data to one destination, its callback function makes the next asynchronous message-passing call.

6.1.1 MCBs

When an asynchronous message-passing function is called to send (or receive) a message, it returns (immediately) the integer ID of a *message control block (MCB)*, which is an opaque CMMD data structure that tracks the status of the transmission. (The term *MCB* alone is used interchangeably to refer to either the integer ID or the data structure itself, since such usage is normally rendered unambiguous by the context.) A program can call one of the asynchronous functions, thus obtaining an MCB, and then simply check the MCB's attributes periodically to see whether the transmission has been completed.

Implementation Note: The internal "handshake" (see Chapter 2) used by the asynchronous functions differs slightly, in terms of timing, from the one used by the cooperative functions. The details are described in Appendix B.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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. To allow a node to free an MCB from an asynchronous send or receive call, you call the function `CMMD_free_mcb`.

6.1.2 Buffer Handling and Restrictions

`CMMD_send_noblock` and `CMMD_send_async` differ in the way they handle the contents of their buffers:

- `CMMD_send_noblock` copies its message immediately into a non-user-visible temporary buffer, then sends that buffer. This imposes a memory space penalty, but frees the user from having to worry about "trashing" the contents of the buffer before it is sent.
- `CMMD_send_async`, in contrast, does not make a safe copy of its message. The user is completely responsible for ensuring that the buffer contents are not modified until the MCB indicates that the transmission is complete.

The `_async` and `_noblock` functions also impose restrictions on the arrays they handle:

- `CMMD_send_async` and `CMMD_send_noblock` do not accept vector serial arrays. To send a vector serial array via an `CMMD_send_async` or `CMMD_send_noblock` via `CMMD_receive_async`, the application must first copy it into a block serial array. To receive a vector serial array, the application must first receive it into a block serial array and then copy it into a vector serial array.
- `CMMD_send_noblock` does not accept parallel arrays.

Copyright © 1995 Thinking Machines Corporation

- `CMMD_send_async` and `CMMD_receive_async` accept only complete parallel arrays. Parallel array sections must be copied into parallel temporary arrays for sending; see the reference page of `CMMD_send_async` for more information.

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 4.1.

6.2 Order of Message Delivery

CMMD does not guarantee that the temporal order in which asynchronous messages are transmitted will correspond to the order in which the function calls are made. For instance, suppose that node 0 makes the following four calls:

```
CMMD_send_async(1, CMMD_DEFAULT_TAG, A, 512, calla, 0);
CMMD_send_async(1, CMMD_DEFAULT_TAG, B, 512, callb, 0);
CMMD_send_async(2, CMMD_DEFAULT_TAG, B, 512, callb, 0);
CMMD_send_async(3, CMMD_DEFAULT_TAG, B, 512, callb, 0);
```

CMMD guarantees neither the temporal order in which arrays A and B are delivered to node 1 nor the order in which copies of B are delivered to nodes 1, 2, and 3. Similarly, there is no guarantee as to the order in which the call-back functions invoked by these four calls will execute.

One guarantee, however, is made. If one node sends multiple messages with a single tag (or with `CMMD_DEFAULT_TAG`) to a second node, then CMMD guarantees that the first sending and first receiving call are paired, and so on. That is, the data from the first send function called is stored in the first receive buffer posted, data from the second send function called goes into the second receive buffer to be posted, and so on.

Thus, the following calls do what you might expect: they move data from array A on node 0 to array A on node 1, and from array B on node 0 to array B on node 1:

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

Node 0:

```
CMMD_send_async(1, CMMD_DEFAULT_TAG, A, 512);
CMMD_send_async(1, CMMD_DEFAULT_TAG, B, 512);
```

Node 1:

```
CMMD_receive_async(0, CMMD_ANY_TAG, A, 512);
CMMD_receive_async(0, CMMD_ANY_TAG, B, 512);
```

On the other hand, the following calls will move data from array A on node 0 to array B on node 1, and from array B on node 0 to array A on node 1.

Node 0:

```
CMMD_send_async(1, CMMD_DEFAULT_TAG, A, 512);
CMMD_send_async(1, CMMD_DEFAULT_TAG, B, 512);
```

Node 1:

```
CMMD_receive_async(0, CMMD_ANY_TAG, B, 512);
CMMD_receive_async(0, CMMD_ANY_TAG, A, 512);
```

The use of tags, of course, would make these calls unambiguous and remove all order dependency. For example, rewriting our sample calls as follows makes fully explicit the fact that transfers from A to A and from B to B are desired:

Node 0:

```
CMMD_send_async(1, 5, A, 512);
CMMD_send_async(1, 6, B, 512);
```

Node 1:

```
CMMD_receive_async(0, 5, A, 512);
CMMD_receive_async(0, 6, B, 512);
```


Copyright © 1995 Thinking Machines Corporation

6.3 Callback Functions

The `CMMD_send_async` and `CMMD_receive_async` functions both take callback function arguments. Callback functions are user-written functions that are invoked upon completion of a message transmission. A callback function may perform computations, asynchronous communications, or both. For example, a callback 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 callback function can also initiate another message-passing communications operation. For example, it can send a reply to a received message.

Callback functions must follow certain rules; also, they are subject to a number of restrictions. These are discussed in Section 6.3.2.

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 callback might send a request message to only one of them. The receiving node could then use a callback 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 11.)

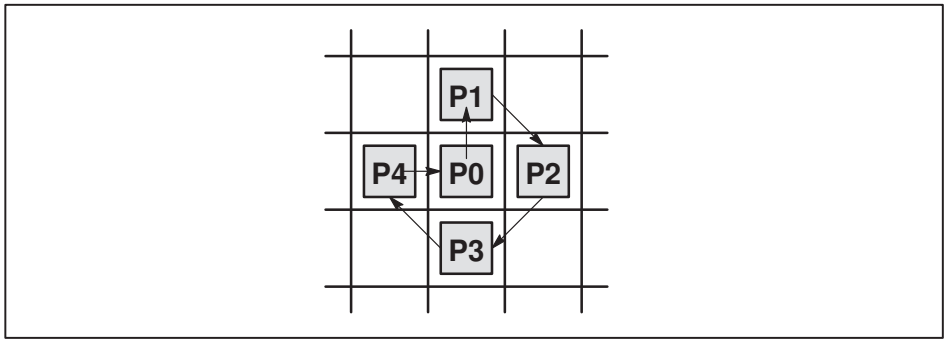

Figure 11. Callback functions can be used to implement simple "stencil" patterns.

The last neighbor in the chain then delivers the collected data 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.

Copyright © 1995 Thinking Machines Corporation

6.3.1 Callback Functions in C and Fortran

A callback 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 callback 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 callbacks, CMMD can use the same internal interface for both C and Fortran.

The second argument to a callback function is the callback 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.

6.3.2 Restrictions on Callback Functions

Callback functions cannot invoke any blocking communication functions (such as `CMMD_send_block` or `CMMD_receive_block`), nor can they perform I/O. If a callback function invokes an asynchronous communication function, it cannot assume that the function will complete before the callback function itself returns.

Callback functions should not transmit or manipulate parallel arrays, nor should they call any CM Fortran or C* statements that do so.

Callback functions must not enable or disable interrupts. (They always operate with interrupts disabled.)

Other restrictions also exist; see Section 14.1.1 for a full discussion of restrictions on handler functions and callback functions.

Copyright © 1995 Thinking Machines Corporation

6.3.3 Active Messages

Callback functions can invoke active messages. Please see Chapter 14 for details.

6.4 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.

6.5 Reference Pages

This section provides reference pages for the following functions:

```
CMMD_send_async
CMMD_receive_async
CMMD_send_noblock
```

PLEASE NOTE

Users are strongly advised to always check the return value of both the cooperative and the asynchronous functions.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_send_async

Sends a serial or parallel array to *dest_node*, optionally invoking a callback function when the send terminates.

SYNOPSIS

C Syntax

```
CMMD_mcb CMMD_send_async(int dest_node, int tag, void *buffer,
         int buffer_desc, void (*callback)(CMMD_mcb *mcb,
         void *callback_arg) void *callback_arg)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_send_async(dest_node, tag, buffer,
         buffer_desc, callback, callback_arg)

    INTEGER dest_node, tag, buffer_desc
    <ARRAY> buffer
    SUBROUTINE callback(mcb, callback_arg)
    INTEGER mcb
    <ARRAY> callback_arg
```

C* Syntax, for Parallel Arrays Only

```
int CMMD_send_async(int dest_node, int tag, void:void *buffer,
         int array_element_length,
         void (*callback)(CMMD_mcb *mcb,
         void *callback_arg), void *callback_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 signed integer, or CMMD_DEFAULT_TAG.
buffer	The location in memory at which the message begins.

Copyright © 1995 Thinking Machines Corporation

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.

callback(mcb, callback_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.

callback_arg
A user-defined data structure passed to the callback function. If no data structure is needed, should be NULL or 0.

RETURNS

The function returns immediately after posting its intent to send (see Appendix B). 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
non-negative 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.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

`CMMD_send_async` returns the integer ID of a message control block (MCB) as soon as it has posted the message to be sent. Thus, it returns almost immediately. The user program then has two responsibilities:

- The user program must ensure that the buffer remains unaltered until the message has actually been sent and received.
- The user program must free the MCB when it is no longer needed (that is, when the entire message has been sent.).

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

If the program does not provide a callback function, it can still query the message's MCB for information and free the MCB, using the auxiliary functions described in Chapter 7.

Callback Functions. The callback function takes two arguments: a pointer to the MCB and the user-supplied *`callback_arg`*. (Note that in Fortran callbacks the MCB argument is the actual integer ID.)*

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

Callbacks may not invoke blocking communication functions, nor call any CM Fortran or C* functions that manipulate parallel data. (For a complete list of restrictions, see Section 14.1.1.)

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 callback function is called; therefore, the callback 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

---

* In CMMD Version 2.0, callback functions could take only one argument, the MCB. They now take two arguments. The lack of a second argument to the callback function under Version 3.0 and higher 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 callback function will quite properly ignore.

Copyright © 1995 Thinking Machines Corporation

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.

Callback and Handler Functions. This function can be called from callback or handler functions.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

CMMD_receive_async

Receives a serial or parallal array from *source_node*, optionally invoking a callback function when the message is complete.

SYNOPSIS

C Syntax

```
CMMD_mcb CMMD_receive_async(int source_node, int tag,
        void *buffer, int buffer_desc,
        void (*callback)(CMMD_mcb *mcb,
        void *callback_arg), void *callback_arg)
```

Fortran Syntax

```
INTEGER FUNCTION CMMD_receive_async(source_node, tag, buffer,
        buffer_desc, callback, mcb, callback_arg)

   INTEGER source_node, tag, buffer_desc
   <ARRAY> buffer
   SUBROUTINE callback (mcb, callback_arg)
   INTEGER mcb
   <ARRAY> callback_arg
```

C* SYNTAX, for Parallel Arrays Only

```
int CMMD_receive_async(int source_node, int tag,
        void:void *buffer, int array_element_length,
        void (*callback)(CMMD_mcb *mcb,
        void *callback_arg), void *callback_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.

Copyright © 1995 Thinking Machines Corporation

buffer_desc For serial arrays, a nonnegative 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.

**callback(mcb, callback_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.

callback_arg
A pointer to any user-defined data structure to be used by the callback function. If no data structure is required, this argument should be `NULL` or 0.

RETURNS

The function returns immediately after creating a message control block (MCB) that describes the message to be received (see Appendix B). The function returns a pointer to the 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
nonnegative integer	The MCB attached to this end of this transmission.
`CMMD_ERRVAL`	No MCBs were available for this transmission.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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 7) to determine when the message has been received. A user-supplied routine, called a *callback*, 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. (Callback functions are described in detail in Chapter 14.)

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 callback function is called; therefore, the callback 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.

Callback and Handler Functions. This function can be called from callback or handler functions.

Copyright © 1995 Thinking Machines Corporation

CMMD_send_noblock

Sends a serial array to *dest_node*, creating a temporary buffer for the data and returning immediately.

SYNOPSIS

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 nonnegative 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 nonnegative integer defining the number of bytes to be sent from the serial buffer.

RETURNS

CMMD_send_noblock returns as soon as it has copied its message to a safe, non user-visible 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.

Version 3.3, April 1995
Copyright © 1995 Thinking Machines Corporation

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.

`CMMD_send_noblock` 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 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 degrades performance; thus it should be used only in cases in which the source buffer must be available for re-use as soon as possible.

Callback and Handler Functions. This function can be called from callback functions or handler functions. However, calling it from a handler is not recommended if the buffer is long, since handlers should always be short-running functions.

Copyright © 1995 Thinking Machines Corporation