The
Connection Machine
System

CMMD Reference Manual

Version 1.1
January 1992

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1991
Revised, January 1992

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

About This Manual

Objectives of This Manual

The *CMMD Reference Manual* describes the CMMD library, a library of communication routines used for creating message-passing programs (sometimes called MIMD programs) to run on the Connection Machine CM-5 supercomputer. It provides

- a brief introduction to the library and to the host/node message-passing model that it implements.
- a "quick reference" list of routines provided by the library, organized by which processors (host, node, or both) and how many processors (one, two or more, or all) can or must call the routine.
- reference chapters for each functional group of routines. These chapters provide information on the routines themselves and, in some cases, on the way in which the routines function and the uses to which they may be put.

Intended Audience

This manual is written for programmers who are developing or porting message-passing programs to run on the Connection Machine CM-5 supercomputer. It assumes some previous knowledge of message-passing programming.

Related Documents

CMMD User's Guide: The *CMMD Reference Manual* should be used in conjunction with the *CMMD User's Guide*, which provides an introduction to the CM-5 supercomputer itself and to the manner in which message-passing programs execute on that machine. Programmers new to the CM-5 supercomputer are urged to read the first two chapters of the user's guide before beginning programming on the machine.

Later chapters of the user's guide describe the tools for compilation, linking, debugging, and program analysis.

Manual Pages: The reference descriptions for individual routines provided in this manual are also available on-line as manual pages accessible via the **man** command.

Revision Information

This edition of the *CMMD Reference Manual* documents Version 1.1 of the CMMD library. Readers should note that this library is still under development and is therefore subject to change.

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
bold typewriter	CMMD functions, and UNIX and CM System Software commands, command options, and filenames, when they appear in syntax statements or embedded in text.
italics	Argument names and placeholders in function and command formats.
typewriter	Code examples and code fragments.
% **bold typewriter** typewriter	In interactive examples, user input is shown in **bold typewriter** and system output is shown in regular typewriter font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an Applications Engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

Chapter 1

Introduction

1.1 Introducing CMMD

The CM message-passing library, CMMD, provides facilities for cooperative message passing between processing nodes. It thus provides simple interprocessor communication that falls outside the range of the CM data parallel languages.

This library is expected to be of particular interest to users who have written C or Fortran programs for machines with MIMD architectures. Such users can port their programs to the CM-5 by replacing the original message-passing library calls with calls to CMMD routines.

The Cooperative Message-Passing Model

CMMD supports a programming model frequently referred to as host/node programming. This model involves two simultaneously running programs. One program runs on the host, while independent copies of the node program run on each processing node. On the CM-5, the host is the partition manager (PM) that controls a partition of the system, while the nodes are the processing nodes within the partition. The host begins execution by performing needed initializations (including initializing the CMMD library) and then invoking the node program; it may have little involvement in subsequent computations.

Within this general programming model, CMMD permits *cooperative* concurrent processing, in which synchronization occurs only between matched sending and receiving nodes and only during the act of communication. At all other times, computing on each node proceeds asynchronously.

This initial release of CMMD primarily supports *blocking* message sending and receiving, but does provide limited support for non-blocking message passing as well. (Future versions of the library are expected to offer further support for asynchronous message passing.) Blocking routines are synchronized routines in which senders wait for their recipients to respond before continuing execution, and vice versa. Programmers using such routines must ensure that each sending routine is matched with a receiving routine, or deadlock may ensue. (The CM-5 timesharing operating system ensures that any such deadlock affects only the erring program, and has no effect on other programs sharing the partition.)

In addition, global functions provide for broadcasting data from and reducing it to the host, for scan and reduce operations, and for global sychronization. (Like their data parallel counterparts, CMMD global functions are able to take advantage of the CM-5's hardware support for global communications.)

Two Exceptions

Two exceptions to the cooperative message-passing format exist. The first is a facility for sending non-blocking *short messages*. Using this facility, each node can send one short message to one or more other nodes and then continue its program without waiting for a response. Only one message from one given node to another can be outstanding; sending two or more messages to the same node requires some synchronization.

For example, node 1 can send node 3 a short message, then perform computations without waiting for node 3 to receive the message. If node 1 then sends a second message to node 3, the system software will check the status of the first message. If that first message has been received, the second is also sent as a non-blocking message. If, however, the first message has not yet been received by node 3, the second send will block until receipt of the first message.

The second exception is a pair of routines that operate outside the CMMD message-passing protocol and thus allow programmers to define their own protocols. These routines should be used only by programmers who are highly experienced in writing message-passing programs, as they provide almost no safeguards against disaster.

CMMD and Other CM Software

CMMD can be called from C and from Fortran 77. This manual documents the C interface (that is, it uses C syntax and data types). Section 1.3, at the end of this chapter, shows the relationship between C data types and Fortran data types.

CMMD routines are completely compatible with the current release of the CM-5 operating system, CMOST Version 7.1. Programs under the control of CMMD routines, however, cannot make calls to data parallel CM libraries, such as parallel I/O or graphics routines. Standard (serial) C calls can be used: UNIX I/O calls from the host program, for instance, or Xlib graphic routines. Future versions of the CMMD library are expected to make provision for moving data between CMMD and data parallel programming modes.

Please note that this library is under continual development and hence subject to possibly substantial changes.

This manual provides information on the CMMD routines. See the *CMMD User's Guide* for information on compiling, loading, use of timers, and debugging.

1.2 How Many Nodes?

Synchronization of processors under the message-passing model affects different numbers of processors according to the operation being performed.

- When one node sends a message, and a second receives it, those two nodes must synchronize. Until both have made their respective calls and the message is transferred, neither call can return.
- If more than two nodes are involved in a set of messages (which can happen in send_and_receive calls), all those nodes must complete their calls before any of the calls can return.
- When a global function is invoked, no call can return until every node (and sometimes the host) has made the call.
- Informational functions usually involve only one node; for example, any node may check whether it has a message pending without involving any other node.

Programs using CMMD calls have the responsibility of checking that all requisite nodes make the appropriate calls at the appropriate times. If this is not done, program performance will suffer and deadlock may ensue.

Please note that global routines can be used only when all processors in the partition take part. If some section of a program involves only a single subset of processors, it cannot make a global call on that subset without hanging the entire program.

The chart on the next several pages summarizes CMMD routines by functionality and by the number and identity of nodes that must call them. Once you are acquainted with the library, you can use this chart as a quick reference.

Succeeding chapters discuss each functional group of routines and provide reference writeups for each routine.

CMMD Function Summary

Single-Node Functions: Host Only

Enabling and Disabling Library Use

```
CMMD_enable()
CMMD_is_enabled()
CMMD_disable()
CMMD_suspend()
CMMD_is_suspended()
CMMD_resume()
```

Global Synchronization

```
CMMD_barrier_sync()
```

Single-Node Functions: Host or Any Node

Informational Functions

CMMD_self_address()
CMMD_host_node()
CMMD_partition_size()

CMMD_bytes_received()
CMMD_bytes_sent()
CMMD_msg_sender()
CMMD_msg_tag()

Polling

CMMD_msg_pending(*int node, int tag*)

Setting and Getting Global Or

CMMD_set_global_or(*int value*)
CMMD_get_global_or()

Sending Short Messages

CMMD_send_short(*int destination, int tag, void *buffer, int len*)
CMMD_wait_for_send(*int destination*)

Two-Node Functions

(Note: In any of these functions, a single node may play both roles, being both sender and receiver.)

Sending and Receiving Messages

CMMD_send (*int destination, int tag, void *buffer, int len*)
CMMD_send_v (*int destination, int tag, void *buffer, int elem_len, int stride, int elem_cnt*)

CMMD_receive (*int source, int tag, void *buffer, int len*)
CMMD_receive_v (*int source, int tag, void *buffer, int elem_len, int stride, int elem_cnt*)

CMMD_send_and_receive (*int source, int source_tag, void *inbuffer, int inlen, int destination, int dest_tag, void *outbuffer, int outlen*)
CMMD_send_and_receive_v (*int source, int source_tag, void *inbuffer, int in_elem_len, int in_stride, int in_elem_cnt, int destination, int dest_tag, void *outbuffer, int out_elem_len, int out_stride, int out_elem_cnt*)

CMMD_swap (*int processor, void *inbuffer, int inlen, void *outbuffer, int outlen*)
CMMD_swap_v (*int processor, void *inbuffer, int in_elem_len, int in_stride, int in_elem_cnt, void *outbuffer, int out_elem_len, int out_stride, int out_elem_count*)

Global Functions: All Nodes, but Not Host

Global Synchronization

CMMD_sync_with_nodes()

Reduce, Scan, and Concatenate

CMMD_reduce_<type>(*<type> value, CMMD_combiner_t combiner*)

CMMD_scan_<type>(*<type> value, CMMD_combiner_t combiner,*
CMMD_scan_direction_t direction,
CMMD_segment_mode_t smode, int sbit,
CMMD_scan_inclusion_t inclusion)

CMMD_concat_with_nodes(*void *element, void *buffer,*
int elem_length)

Global Functions: Host plus All Nodes

Enabling and Disabling Short Message Sending

CMMD_enable_short_messages()
CMMD_disable_short_messages()

Broadcast

CMMD_bc_from_host(*void *buffer, int len*)
CMMD_receive_bc_from_host(*void *buffer, int len*)

CMMD_distrib_to_nodes(*void *buffer, int elem_length*)
CMMD_receive_element_from_host(*void *buffer, int length*)

Global Synchronization

CMMD_sync_host_with_nodes()
CMMD_sync_with_host()

Reduce and Concatenate

CMMD_reduce_from_nodes_<type>(*<type> value, CMMD_combiner_t combiner*)
CMMD_reduce_to_host_<type>(*<type> value, CMMD_combiner_t combiner*)

CMMD_gather_from_nodes(*void *buffer, int elem_length*)
CMMD_concat_elements_to_host(*void *element, int elem_length*)

1.3 C and Fortran 77

Fortran 77 calling sequences for CMMD routines are identical to C calling sequences, in terms of routine names, parameter names, and parameter order.

Data types, however, are declared differently. The following table shows translations from C data types to Fortran 77 data types.

C	Fortran
int	integer
char	character
`CMMD_combiner_t`	integer
`CMMD_scan_direction_t`	integer
`CMMD_segment_mode_t`	integer
`CMMD_scan_inclusion_t`	integer
unsigned	integer
float	real
double	double precision

In the ANSI C programming language, *void* is a special data type that has no meaningful values. The equivalent of a Fortran **SUBROUTINE** (a subprogram that returns no value) is expressed in C as a function whose return type is *void*.

A widespread C programming convention is that the type "pointer to void" represents a pointer to any desired type. If a subroutine has a formal parameter of type "pointer to void", then a pointer of any type may correctly be used as the corresponding actual argument. The called routine must then assume or deduce the properties of the data pointed to, usually from information conveyed by the other parameters.

The CMMD library uses this convention for all cases in which an argument is a pointer to an area of memory that either contains data to be sent or is reserved for data to be received. Pointers indicate only the starts of memory areas; the sizes of the areas are specified through other parameters.

Chapter 2

Initialization

2.1 Initializing CMMD

Data parallel and message-passing program execution make different demands on the CM-5's communications networks (the Control Network and the Data Network), and thus require different settings for network *participation*. For message-passing programs using CMMD, these settings are controlled by two pairs of functions, which must be called from the host. The first pair, **CMMD_enable** and **CMMD_disable**, perform the initial tasks necessary first to enable message passing and later to disable message passing and restore the network setting to the state it was in when **CMMD_enable** or **CMMD_resume** was last called. The second pair, **CMMD_suspend** and **CMMD_resume**, are used to suspend and resume message passing temporarily within the course of a program (for example, to allow use of some other library).

Programs or routines using CMMD should therefore begin with the host calling **CMMD_enable** and end with the host calling **CMMD_disable**. Calls by the host to **CMMD_suspend** and **CMMD_resume** may be placed where necessary within the program (if they are needed).

Each of these calls requires that the system be in the appropriate state: for instance, an error results from trying to disable message passing when it is not enabled. Therefore, two informational routines are provided: **CMMD_is_enabled** tells whether message passing has been enabled; **CMMD_is_suspended** tells whether message passing is currently suspended.

2.2 Initializing the Short Message Facility

At this initial release, CMMD uses a model of cooperative, or loosely synchronous, message passing. A short message facility within CMMD does, however, allow the non-blocking sending and receiving of short messages (up to 16 bytes). This facility must be enabled and disabled separately from CMMD itself. A program enables CMMD and starts passing messages. At some point, when the sending of short messages is useful, the program enables that facility, creating short-message buffers on all the nodes. When the facility is no longer useful, it may be disabled and its buffer space reclaimed. If the facility is still enabled when CMMD itself is disabled, it will be disabled automatically as part of the overall disabling.

The routines that enable and disable the sending of short messages are **CMMD_enable_short_messages** and **CMMD_disable_short_messages**. These routines must be called synchronously by all nodes and the host; they are discussed at the end of this chapter.

2.3 Functions That Initialize CMMD

CMMD_enable()

CMMD_enable must be called by the host at the beginning of any program that uses CMMD routines. It records the current states of communications in the networks, allocates space for message buffers in the host and the nodes, and initializes variables needed for message-passing operations, and synchronizes the host and nodes.

CMMD_is_enabled()

CMMD_is_enabled returns **TRUE** if CMMD is currently enabled (that is, if it has been enabled and is not suspended). Otherwise, it returns **FALSE**. Only the host can call this function.

CMMD_disable()

CMMD_disable must be called by the host at the termination of a program that uses CMMD routines. It synchronizes the host with the nodes, deallocates the space originally allocated in the host and the nodes for message buffers, and restores the original states of the communications networks. (That is, it returns the networks to the state found when **CMMD_enable** or **CMMD_resume** was last called.)

An error is signaled if CMMD is not currently enabled. If it has been suspended, it must be resumed before it can be disabled.

CMMD_suspend()

CMMD_suspend returns control temporarily to the host processor, to allow data parallel processing. The routine synchronizes the host with the nodes, saves the current states of the communication networks, and restores the states that the networks were in before the latest **CMMD_enable** or **CMMD_resume** was called. This routine can be called only by the host.

CMMD_suspend signals an error if CMMD has not been enabled or if it is already suspended.

CMMD_is_suspended()

CMMD_is_suspended returns **TRUE** if message passing has been enabled and then suspended; otherwise, it returns **FALSE**. Only the host can call this routine.

CMMD_resume()

If CMMD has been suspended, **CMMD_resume** saves the current states of the communications networks and restores the communications network states in effect before the last call to **CMMD_suspend**. The user program should ensure that host and nodes are synchronized after making this call before beginning message passing again.

CMMD_resume can be called only by the host. It returns an error if CMMD is not in a suspended state.

2.4 Functions That Initialize the Short Message Facility

CMMD_enable_short_messages()

CMMD_enable_short_messages synchronizes the host and all nodes and allocates internal storage necessary to support the non-blocking sending of short messages via the **CMMD_send_short** function. It must be called on the host and all nodes.

CMMD_enable_short_messages has no effect on a program's ability to use CMMD calls other than **CMMD_send_short**. All standard CMMD calls can be used while the **send_short** facility is enabled.

An error is signaled if the facility is already enabled.

CMMD_disable_short_messages()

CMMD_disable_short_messages disables the non-blocking sending and receiving of short messages. It must be called on the host and all nodes.

On each node, the call waits until all short messages sent from this node have been received (e.g., by **CMMD_receive**). It then frees the internal storage allocated on that node for short message support.

If short message passing is enabled at the time that CMMD itself is disabled, then this function is called internally by **CMMD_disable**.

An error is signaled if this function is called when the facility is not enabled.

Chapter 3

Processor Information

Processors, both host and nodes, must address each other explicitly during message passing. Therefore, routines are needed to provide host and node identifiers. **CMMD_host_node** provides the host identifier, while **CMMD_self_address** provides the calling node's own identifier.

For each partition, the set of node identifiers consists of the integers from 0 to the number of nodes in the partition minus 1, inclusive. The function **CMMD_partition_size** returns the size of the current partition. The host identifier is an integer outside the range of the partition size.

3.1 Processor Information Functions

CMMD_self_address()

Called from a process running on a given node, **CMMD_self_address** returns the node identifier for that node.

Node identifiers are integers, from 0 to the maximum number of processors in the partition –1, inclusive. For example, every 128-node partition contains nodes 0 to 127. Node identifiers are logical identifiers: programs and programmers need never concern themselves with physical processor addresses.

CMMD_host_node()

CMMD_host_node returns the host identifier (an integer not in the partition set). It can be called from the host itself or from any node.

CMMD_partition_size()

CMMD_partition_size returns the number of processors in the current partition. It can be called from the host or from any node.

Chapter 4

Message Passing

4.1 Introduction

Blocking and Non-Blocking Message Passing

This initial version of CMMD primarily supports *cooperative* message passing, in which the sending and receiving of messages are synchronized. Most of the message-passing routines discussed in this chapter fit this model. They not only pass information from one node to another, but also synchronize the nodes in so doing. They are therefore called *blocking* routines.

CMMD does, however, allow the non-blocking sending and receiving of short messages (up to 16 bytes). This facility must be enabled and disabled separately from CMMD itself, using the routines **CMMD_enable_short_messages** and **CMMD_disable_short_messages**. These routines, which must be called by host and all nodes, are discussed in Chapter 2.

Two routines are used to send short messages: **CMMD_send_short** to actually send the message, **CMMD_wait_for_send** to allow users to impose some measure of synchronization, should they wish to do so. These routines are discussed in Section 4.4, at the end of this chapter. No special routines are needed for receiving short messages: **CMMD_receive** or **CMMD_receive_v** may be used.

Patterns of Message Passing

A processor can play one of four roles in message passing:

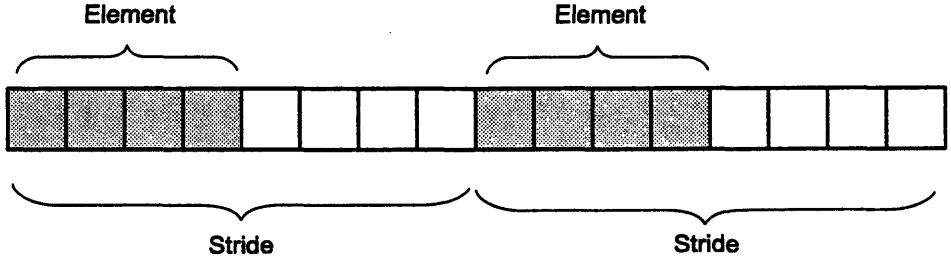
- It can send a message.
- It can receive a message.
- It can send and receive messages simultaneously. Two special cases:
  - It can take part in a *cshift*, in which all nodes simultaneously send (in one direction) and receive (from another direction).
  - It can take part in a *swap*, in which it and one other processor exchange messages, simultaneously sending to and receiving from each other.

Routines are provided for each of these roles: send, receive, send_and_receive, and swap. These routines are discussed in Sections 4.2 and 4.3.

Regular Messages and Vector Messages

Message-passing routines support two types of messages: standard messages, in which bytes are stored in normal sequential order, and vector messages, in which elements are separated by some amount of space. Each of the routines in this section, therefore, has two versions: a standard version, and a vector version (labeled with a final _v).

In a vector message, the distance between the starting position of one element and the starting position of the next element is referred to as the "stride." The stride includes one element plus the intervening space before the beginning of the next element. Normally, therefore, the stride is larger than the element size.



4.2 Functions for the Paired Sending and Receiving of Messages

4.2.1 Sending Messages

CMMD_send (*int destination, int tag, void *buffer, int len*)

CMMD_send_v (*int destination, int tag, void *buffer, int elem_len, int stride, int elem_cnt*)

destination	An integer identifying the node to which the message is to be sent.
tag	An integer from 0 to 127, inclusive, which serves as a label for the message.
**buffer*	A pointer to a buffer that contains the message to be sent.
len	The length of the buffer, in bytes.
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.
elem_cnt	(Vector sends only.) An integer specifying the number of elements in the vector.

CMMD_send and **CMMD_send_v** send the contents of a buffer of specified length, tagged with the specified tag, to the given destination node. The node must be inside the partition; otherwise, an error results. (The symbol **DEFAULT_MSG_TAG** is the standard default tag.)

Buffers may be of any length up to the maximum memory per node. A NULL buffer pointer or a length of zero causes a message of zero data length to be sent.

The message is not sent until the receiving node acknowledges that it is ready to receive a message labeled with the specified tag from this node. In its response, the receiving node specifies the maximum length of the message it is willing to receive. Normally, this is the same as the length specified by **CMMD_send**, but it may be either larger or smaller.

For example, if the receiving node does not know the length of the message to be sent to it, it can specify the maximum buffer length (or whatever shorter length

seems a reasonable maximum for the type of message expected) and accept as many (or as few) bytes as the sender desires to send.

On the other hand, if the receiving node does not have room for the full message that the sender wishes to send, it can signal that it wishes to receive a shorter message. **CMMD_send** is constrained to send no more data than the receiver has signaled that it can accept. (Please note: This is an implementation-dependent constraint that may be lifted at some future release.) Thus, it sends either the amount it planned to send or the amount **CMMD_receive** allows, whichever is less.

After sending whatever amount of data it is allowed to send, **CMMD_send** returns; it returns a value of 0 if it sent its entire message and a value of 1 if it sent a smaller amount. In the latter case, or in the case in which **CMMD_receive** allocates a "maximum-length" buffer, the program should call **CMMD_bytes_sent** to get the number of bytes actually sent.

Standard Sends and Vector Sends

A standard message, sent by **CMMD_send**, begins at the starting place identified by the **buffer* argument, and proceeds for *len* sequential bytes. A vector message, sent by **CMMD_send_v**, takes a number of non-sequential elements from the buffer, and sends those as a sequential message. (In other words, it performs an implicit gather.)

Normally, the stride specified for **CMMD_send_v** will be larger than the element length. This difference creates the vector send: *elem_len* bytes are put into the message, then (*stride* – *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. (See Figure 1).

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

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

Note that you do not specify the total length of the message in a vector 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. Figure 1 illustrates stride, element length, element count, message length, and buffer length for a vector send.

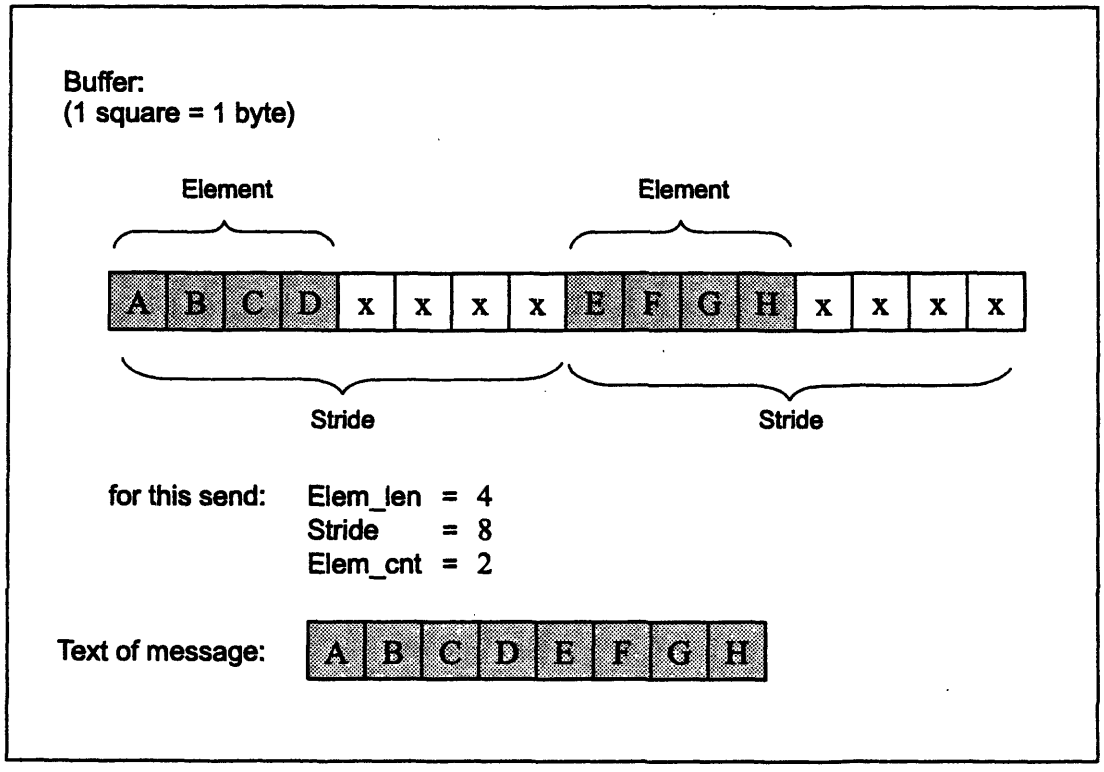


Figure 1. A vector send.

As an example of regular and vector sends, let us consider the case of a 4 x 6 matrix A, filled with self-addresses from 0 to 23, in which each element is one byte long, 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 (5, DEFAULT_MSG_TAG, &A, 6)
```

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_cnt*), each one byte long (*elem_len*), located six bytes apart (*stride*).

```
CMMD_send_v (3, DEFAULT_MSG_TAG, &A, 1, 6, 4)
```

Normally, the receiving node would accept the first message with a standard receiving call (**CMMD_receive**) and the second with a vector receiving call (**CMMD_receive_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 with a stride of 4.

More about Vector Sends

Vector sends, like standard sends, are constrained by the destination's receive request. A sending node offers to send (*selem-count* * *selem-length*) bytes; a receive message agrees to accept (*delem-count* * *delem-length*). The smaller number of the two is sent, in the following manner:

(1) Each element of the source is sent in its entirety until the appropriate number of bytes sent is reached.

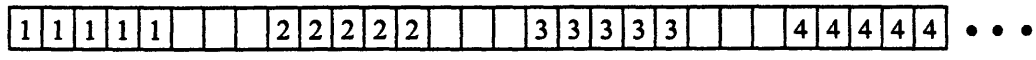
(2) If *selem_len* != *delem_len*, the source elements will be broken up and distributed across the destination's element length (not across its stride).

Note that this is in contrast to what some might expect. CMMD calls *DO NOT* send only as many bytes of each source element as will fit in each destination element.

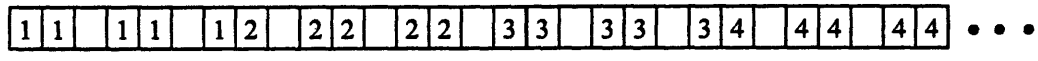
For example, if *selem_len* > *delem_len*

```
selem_len = 5, sstride = 8,
delem_len = 2, dstride = 3
```

the source buffer would contain



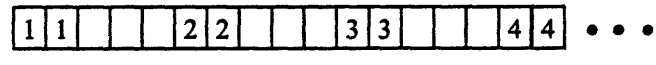
and the destination buffer (after the operation) would contain



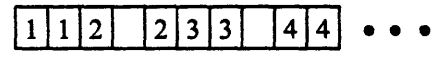
On the other hand, if *selem_len* < *delem_len*

```
selem_len = 2, sstride = 5,
delem_len = 3, dstride = 4
```

the source buffer would contain



and the destination buffer (after the operation) would contain



4.2.2 Receiving Messages

CMMD_receive (*int source, int tag, void *buffer, int len*)

CMMD_receive_v (*int source, int tag, void *buffer, int elem_len, int stride, int elem_cnt*)

source	An integer identifying the node from which the message is to be sent (**ANY_NODE** allows any node to be the sender).
tag	An integer from 0 to 127, inclusive, which serves as a label for the message (**ANY_TAG** allows receipt of a message labeled with any tag).
**buffer*	A pointer to a buffer that will contain the message to be received.
len	(Non-vector function only.) The length of the buffer, in bytes.
elem_len	(Vector functions only.) An integer specifying the length of each element in the vector, in bytes.
stride	(Vector functions only.) An integer specifying the distance in bytes between the starting addresses of the vector elements.
elem_cnt	(Vector functions only.) An integer specifying the number of elements in the vector.

CMMD_receive and **CMMD_receive_v** inform the source node that they are ready to receive a message of *len* bytes with a specified tag; they then wait for a message with the given tag to be sent from the given source. These routines can take the special symbol **ANY_NODE** as the source argument, indicating that any source is acceptable, and the symbol **ANY_TAG** as the tag argument, indicating that any tag will be accepted.

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

Once an acceptable message is sent, **CMMD_receive** and **CMMD_receive_v** copy the message into the specified buffer. They return a value of 0 if the number

of bytes received equals *len*; otherwise they return 1, and **CMMD_bytes_received()** can be called to get the number of bytes actually received.

Standard Messages and Vector Messages

All messages sent by CMMD calls are packed in sequential order. For many, this is the actual data ordering: **CMMD_receive** handles this type of message.

Other messages, however, send data that is not to be considered sequential: an array section would be one example. In this case, **CMMD_receive_v** is used, and the call specifies that the information to be received is to be considered a vector of *e* elements (*elem_count*), each *m* bytes long (*elem_length*), each element to be placed in an area of the buffer that is *n* bytes long.

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

CMMD_send_v and **CMMD_receive_v** are frequently paired, so that data is received in the same geometry from which it was sent. It is possible, however, to receive data in a geometry different from that in which it was sent: for instance, sequential data may be broken into a vector (thus "scattering" the data), or a vector received as sequential (thus "gathering" it). Figure 2 illustrates these four possible patterns.

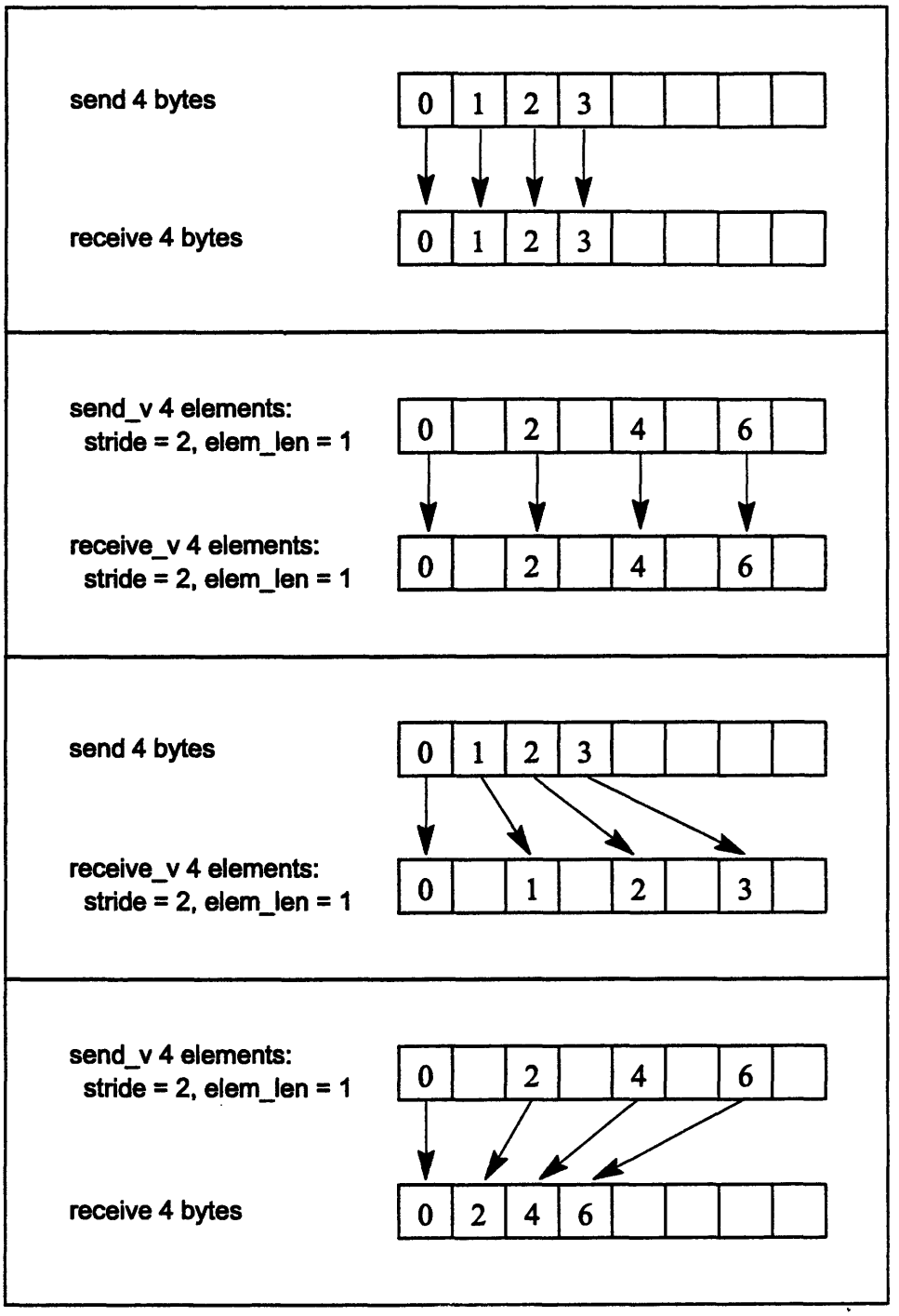


Figure 2. Sending and receiving data.

4.3 Simultaneous Sends and Receives

4.3.1 In Any Pattern

CMMD_send_and_receive (*int source, int source_tag, void *inbuffer, int inlen, int destination, int dest_tag, void *outbuffer, int outlen*)

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

source — An integer identifying the node from which a message will be received by this node.

source_tag — An integer, 0–127 inclusive, or **ANY_TAG**, labeling the message to be received.

**inbuffer* — Pointer to the buffer that will contain the message to be received.

inlen — (Non-vector functions only.) Length, in bytes, of the buffer to hold the message received by this node.

in_elem_len — (Vector functions only.) Length, in bytes, of each element in the vector to be received by this node.

in_stride — (Vector functions only.) Number of bytes between starting addresses of elements in the vector that comprises the message to be received by this node.

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

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

dest_tag — An integer, 0–127 inclusive, labeling the message that will be sent by this node.

**outbuffer* — A pointer to the buffer holding the message to be sent by this node.

outlen	(Non-vector functions only.) Length, in bytes, of the buffer to be sent by this node.
out_elem_len	(Vector functions only.) Length, in bytes, of each element in the vector to be sent by this node.
out_stride	(Vector functions only.) Number of bytes by which starting addresses of elements in the vector to be sent are separated.
out_elem_count	(Vector functions only.) Number of elements that comprise the vector to be sent by this node.

These two functions allow nodes to send and receive messages simultaneously. The routines can be used to perform common grid communication, or to send and receive in more random patterns. 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. The buffers may overlap.

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 **TRUE** if the number of bytes received equals *inlen* and the number of bytes sent equals *outlen*; otherwise they return **FALSE**, and **CMMD_bytes_received()** and **CMMD_bytes_sent()** can be called to get the number of bytes received and sent, respectively.

CMMD_send_and_receive handles sequential data, while **CMMD_send_and_receive_v** exhibits gather/scatter behavior.

4.3.2 Further Notes

(1) The strides for sent and received messages do not have to be equal. For example, to perform a transpose in which the sends are vectored and the receives are sequential, set *in_stride* as needed for the sends and set it equal to *in_elem_len* for the receives. (For more information on vector messages, see the entry for **CMMD_send**.)

(2) The send_and_receive functions should be used when a program needs to perform circular shifts on an array. Each node sends in one direction and receives from another direction, as in the example diagrammed in Figure 3 below.

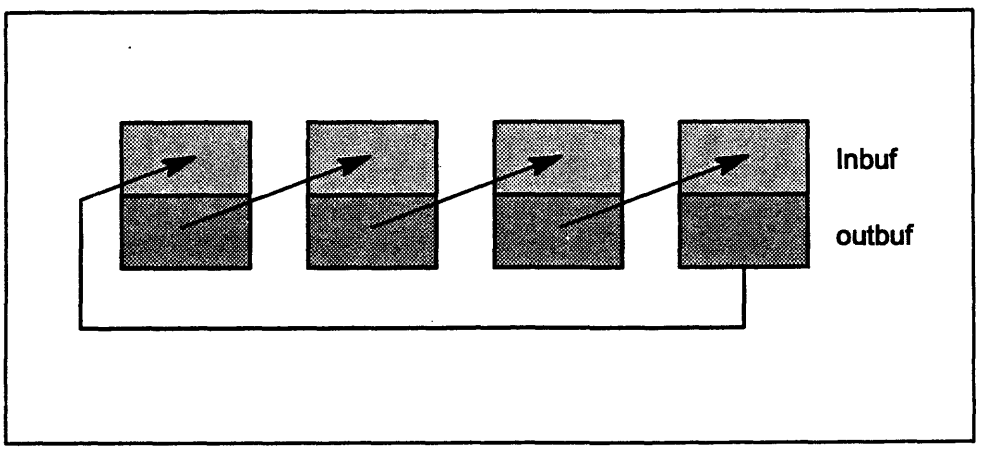


Figure 3. A circular shift on 4 nodes.

(3) 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**: 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**: receives from 2

4.3.3 Swaps: An Exchange between Two Nodes Only

CMMD_swap (*int processor, void *inbuffer, int inlen, void *outbuffer, int outlen*)

CMMD_swap_v (*int processor, 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_cnt*)

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

**inbuffer*	A pointer to the buffer that will hold the received message.
inlen	(**CMMD_swap** only.) Length, in bytes, of the buffer that will hold the message received by this node.
in_elem_len	(**CMMD_swap_v** only.) Length, in bytes, of each element in the vector to be received by this node.
in_stride	(**CMMD_swap_v** only.) Number of bytes between starting addresses of elements in the vector to be received by this node.
in_elem_count	(**CMMD_swap_v** only.) Number of elements that comprise the vector to be received by this node.
**outbuffer*	A pointer to the buffer holding the message to be sent by this node.
outlen	(**CMMD_swap** only.) Length of the buffer to be sent by this node.
out_elem_len	(**CMMD_swap_v** only). Length, in bytes, of each element in the vector to be sent by this node.
out_stride	(**CMMD_swap_v** only.) Number of bytes by which starting addresses of elements in the vector to be sent are separated.
out_elem_count	(**CMMD_swap_v** only.) Number of elements that comprise the vector to be sent by this node.

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.

For an explanation of sequential versus vector routines, see **CMMD_send**.

4.4 Non-Blocking Short Message Sending

CMMD_send_short (*int destination, int tag, void *buffer, int len*)

destination	An integer identifying the node to which the message is to be sent.
tag	An integer from 0 to 127, inclusive, which serves as a label for the message.
**buffer*	A pointer to a buffer that contains the message to be sent.
len	The length of the buffer, in bytes.

CMMD_send_short sends a message of up to **CMMD_SHORT_MESSAGE_BYTES** (16) from this node's buffer, labeled with the specified tag, to the destination node. If no previous short send from this node to the specified destination node is outstanding, **CMMD_send_short** returns immediately; unlike **CMMD_send**, it does not wait for the destination to receive the message. If a previous short send from this node to the specified destination node is still in transit, the call waits until that message has been received (e.g., by **CMMD_receive**) before returning.

Note that a given node can send a single short message to any number of destination nodes, without having to wait for acknowledgment.

An error is signaled if **CMMD_send_short** is called when the short message facility is not enabled; that is, before the host and all nodes have called **CMMD_enable_short_messages**, or after they have called **CMMD_disable_short_messages**.

CMMD_wait_for_send (*int destination*)

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

CMMD_wait_for_send checks to see whether a prior short message from this node to the specified destination node is outstanding (not yet received). If such a message exists, the function waits until that message has been received (e.g., by **CMMD_receive**). If destination is **ANY_NODE**, the function waits until all previous messages from this node to any destination have been received.

Before sending a message to the specified node (**n**), **CMMD_send_short**(**n**, ...) automatically calls **CMMD_wait_for_send**(**n**), thus ensuring that a second send

to node **n** does not occur until the first has been received. A program would call **CMMD_wait_for_send** explicitly if the programmer wanted to ensure that a message to one node was received before a message to another node was sent, but did not want to wait immediately after the first send. The pattern might be

```
send_short to node n
do some other stuff
call wait_for_send on node n
send_short to node m
```

Chapter 5

Polling

Message-passing programs need some way of identifying whether, at any given time, there are messages that are either in transit or waiting to be sent. To identify such messages, a program *polls* for them.

A process on any individual node may call **CMMD_msg_pending()** to poll for a message, and issue a message-receive call only after it knows that a message is waiting to be sent. This allows the process to avoid having to block while waiting for a message. (A receiving process that relies on polling but polls infrequently may, of course, cause sending processes to block while waiting for the receiver.)

5.1 Polling Function

CMMD_msg_pending (*int node, int tag*)

node	Integer identifying a node. (May be **ANY_NODE**.)
tag	Integer identifying a tag. (May be **ANY_TAG**.)

CMMD_msg_pending returns **TRUE** if there is a message waiting to be received from the specified node (or from any node if **ANY_NODE** is supplied as the node argument) with tag *tag* (or any tag if **ANY_TAG** is supplied as the tag argument). It returns **FALSE** otherwise.

If **ANY_NODE** is used, the function **CMMD_msg_sender()** can then be called to get the node identifier of the pending sender; if **ANY_TAG** is used, **CMMD_msg_tag** will return the tag of the pending message.

Chapter 6

Auxiliary Routines

These are the routines that tell you what really happened when you sent that message from one node to another: How much was sent or received? By what node was it sent? How was it tagged? Although their obvious uses are as responses to return values of 1 (signifying incomplete transmission or reception) or to the reception of messages sent from **ANY_NODE** or labeled with **ANY_TAG**, these informational routines can be called at any point during a program.

CMMD_bytes_received() Returns the number of bytes received by this node in its most recent message.

CMMD_bytes_sent() Returns the number of bytes sent in the last message.

CMMD_msg_sender() Returns the node identifier for the last message received except when issued following a call to **CMMD_msg_pending**. In that case:

- If the call to **CMMD_msg_pending** returned **TRUE**, **CMMD_msg_sender** returns the identifier of the node that is waiting to send a message.
- If the call to **CMMD_msg_pending** returned **FALSE**, calling **CMMD_msg_sender** causes an error.

CMMD_msg_tag()

Returns the tag of the last message received except when issued following a call to **CMMD_msg_pending**. In that case:

- If the call to **CMMD_msg_pending** returned **TRUE**, **CMMD_msg_tag** returns the tag of the message that is waiting to be received.
- If the call to **CMMD_msg_pending** returned **FALSE**, calling **CMMD_msg_tag** causes an error.

Chapter 7

Broadcasts

Broadcasts are messages sent from the host to all nodes. Two kinds exist: The host may broadcast the entire contents of the buffer to all nodes (in which case all receive identical data) or it may parcel out elements from the buffer among all nodes, one element per node.

All nodes receive data simultaneously, and all receive the same amount of data. For this reason, it is very important to ensure that all nodes have sufficient buffer space to hold the broadcast message.

The host and all the nodes must take part in these broadcasts. Once a broadcast is signaled, either by the host or by any node, the hardware begins checking for responses. Only when the hardware signals that the entire broadcast is complete can any of the broadcast calls return.

7.1 Broadcasting the Entire Buffer to All Nodes

CMMD_bc_from_host(*void *buffer, int len*)

CMMD_receive_bc_from_host(*void *buffer, int len*)

**buffer*	A pointer to a buffer that holds the message being broadcast and received.
len	The length, in bytes, of the buffer being broadcast and received.

The host process calls **CMMD_bc_from_host** to broadcast a buffer of the specified length (in bytes) to all nodes. All nodes must call **CMMD_receive_bc_from_host**, with the same length argument, to receive the buffer.

PLEASE NOTE

If length arguments are not identical across all nodes, a segmentation fault may result.

Please note also that all processors within the partition must take part in this operation. If a given program divides the partition into sections, an attempt to use global operations within a section will fail.

These functions do not return until the broadcast is complete; that is, until the host and all the nodes have made their calls.

7.2 Distributing a Buffer among the Nodes

CMMD_distrib_to_nodes (*void *buffer, int elem_length*)

CMMD_receive_element_from_host (*void *buffer, int length*)

**buffer*	A pointer to the buffer that holds the messages being sent and received. For the host, the length of the buffer (in bytes) must be at least (**CMMD_partition_size()** * *elem_length*). For a node, the length must be at least *elem_length*.
elem_length	The length (in bytes) of each element to be sent.
length	The length (in bytes) of the buffer that is to receive the element being sent.

The host process calls **CMMD_distrib_to_nodes** in order to distribute elements of the given length from the specified buffer to each node in processor order. The length (in bytes) of the buffer on the host must be at least (**CMMD_partition_size()** * *elem_length*). Only the first (**CMMD_partition_size()** * *elem_length*) bytes are sent; each node receives one element.

In response to the host call, all nodes must call **CMMD_receive_element_from_host**, specifying a buffer of the appropriate size to receive the element.

Neither the host call nor any of the node calls return until all have been made and completed.

Chapter 8

Global Synchronization

Global synchronization functions, as their name implies, serve to synchronize all nodes (and optionally the host as well) at a given point in a program. Three versions are provided:

CMMD_sync_host_with_nodes **CMMD_sync_with_host**	This pair of calls serves as a synchronization point for host and nodes together.
CMMD_sync_with_nodes	This call, sent by all nodes, allows them to synchronize themselves without the host's participation.
CMMD_barrier_sync	This call, sent only by the host, synchronizes host and nodes at the completion of all currently executing node functions.

All processors in the partition must join in these calls. Once the host or any node has begun one of these synchronization calls, the CM hardware keeps track of responses, and allows none of the calls to return until all nodes (and the host, when needed) have made their call.

In addition to these synchronous routines, two asynchronous global OR routines allow host and all nodes to signal to each other by contributing to a global OR and reading its results.

CMMD_set_global_or	Sent by host and all nodes, this call contributes a value (0 or nonzero) toward the creation of a global OR.
CMMD_get_global_or	Sent by host or any node, this call reads the current value of the global OR.

By using the **CMMD_set_global_or** function, each processor contributes to the global OR at an appropriate time; the hardware checks and updates the global value at frequent intervals; and individual processors read the value when desired. Thus, the global OR mechanism can be used as a non-blocking method of determining when all processors have reached a given state. All processors would start a task, for instance, by sending a 1. As each finished its share of the task, it would send a 0. By checking the value of the global OR (which would change to 0 only when all processors had finished), a processor could determine whether the whole task was complete and thus select its own next action.

Please note: These asynchronous global OR functions should not be confused with the synchronous global-OR reduction operation, which is explained later, in the section on Scans, Reductions, and Concatenation.

8.1 Global Synchronization Functions

CMMD_sync_host_with_nodes()

CMMD_sync_with_host()

The host calls **CMMD_sync_host_with_nodes** to synchronize itself and all the nodes. The nodes respond by calling **CMMD_sync_with_host**. These calls return only after the host and all nodes have made the call.

CMMD_sync_with_nodes()

CMMD_sync_with_nodes synchronizes the calling node with all other nodes. Once one node has made this call, all nodes must; the function does not return until they do. (Note that this routine does not involve the host.)

CMMD_barrier_sync()

A program running on the nodes of a CM-5 system alternates between two states: It can be executing a procedure, or it can be *in the dispatch loop*, waiting for the host to initiate execution of a procedure. (For more information about this execution process, see Chapter 2 of the *CMMD User's Guide*.)

The **CMMD_barrier_sync** function is called by the host only. It synchronizes the host with the completion of all previously called node procedures. It returns only when all nodes have finished execution and have returned to the dispatch loop.

PLEASE NOTE

All host-node communication for all nodes in a given program block must be complete before the host processor makes this call. If the call is made while any communication between host and node is pending, the program will hang.

CMMD_set_global_or (*int value*)

value An integer, either 0 or nonzero.

Callable on any processor (host or node), **CMMD_set_global_or** allows a processor to contribute a value (either 0 or some nonzero integer) to a global OR function — that is, an OR in which host and all nodes may take part. The function returns when the value has been sent; it does not wait for participation by any other processor.

CMMD_get_global_or()

Callable on either the host or the nodes, **CMMD_get_global_or** returns the current value of a global OR function over all processors, host and nodes alike.

This function is asynchronous; it requires participation by no other processors.

If **CMMD_set_global_or** has not already set a value for the global OR, calls to **CMMD_get_global_or** return unpredictable results.

As contributions to this global OR may be asynchronous, the hardware checks the value at frequent intervals and updates it as needed. Note, however, that some network delay exists during reception and propagation of values; thus, there is a small but actual window between the time at which a processor sends a **set_global_or** message and the time by which that message can affect the result of another node's **get_global_or** request.

Chapter 9

Scan, Reduction, and Concatenation Operations

Scans, reductions, and concatenation are global operations. Given a buffer containing some value in each node, these global computations operate cumulatively on the buffer set to perform such tasks as

- summing the value across all the nodes
- finding the largest or smallest value
- performing a bitwise AND, OR, or XOR

For reduction operations, the final value can be returned either to all the nodes or to the host. For scans, the cumulative results are returned as a running tally across all the nodes.

All nodes within the partition must take part in these calls. If the result is to be returned to the host, then the host must also take part.

These global functions impose synchrony: those involving both host and nodes do not return until host and all nodes have made their (different) calls; those involving only nodes do not return until all nodes have made the call.

Each scan and reduction function comes in four versions: one for integer, one for unsigned integer, and one each for single- and double-precision floating-point numbers. Each version requires as input a value of the type specified in its name, and returns a value of the same type. Exceptions to this rule are the float routines, which take float arguments but return double results.

Because scan and reduction functions may perform one of a number of operations, they take as an argument one of the following symbols representing the operation to be performed.

`CMMD_combiner_add`	Add operations.
`CMMD_combiner_uadd`	Add operations (unsigned).
`CMMD_combiner_max`	Return the largest value found.
`CMMD_combiner_umax`	Return the largest value found (unsigned).
`CMMD_combiner_min`	Return the smallest value found.
`CMMD_combiner_umin`	Return the smallest value found (unsigned).
`CMMD_combiner_ior`	Inclusive OR operation.
`CMMD_combiner_xor`	Exclusive OR operation.
`CMMD_combiner_and`	Logical AND operation.

Thus, for example, a `CMMD_reduce_int` function, called using the `CMMD_combiner_max` argument, would compare the values on all nodes and return the largest value to all nodes, while a call to `CMMD_reduce_to_host` with the `CMMD_combiner_add` argument would add the values from all nodes and return the sum of the values to the host.

9.1 Reductions, Scans, and Segmented Scans

Reduction operations, scans, and segmented scans provide three basic methods of all-to-all and all-to-one communication. (See Figure 4.)

Reductions

A reduce operation starts with values in every processor and ends with a single value, either in every node or in every node plus the host processor. Values may be added, so that the sum of all values is returned; or the largest or smallest value may be chosen; or an OR or XOR may be done across all the values. In each case, one final result is returned.

Thus, on four nodes holding the values

4 9 7 6

a reduce/add would return

26 26 26 26

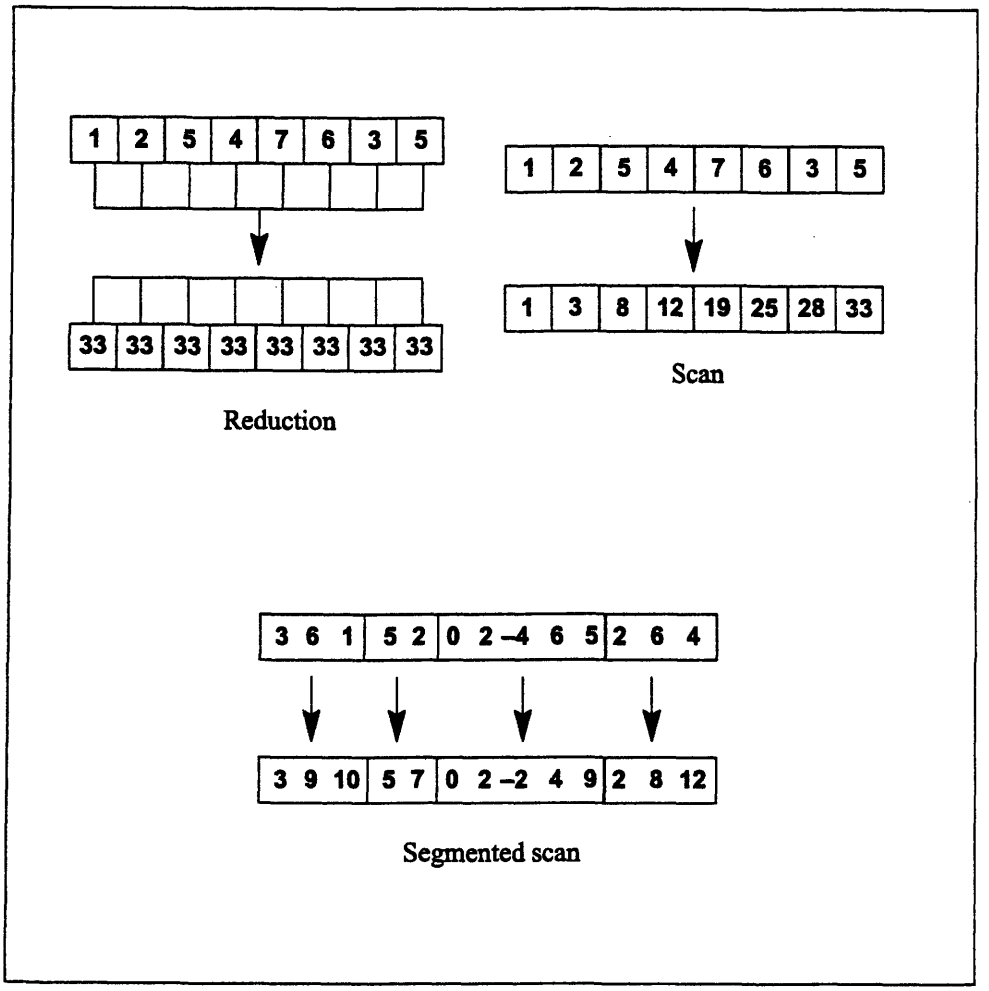


Figure 4. Global summation operations.

Scans

A scan (sometimes called a parallel prefix operation) moves from processor to processor, in processor identifier order, creating a running tally of results in each processor. The function call specifies whether the scan proceeds upward (0 to *n*) or downward (*n* to 0), and whether the scan is to be inclusive or exclusive. (In an inclusive scan, the source value contained in any given node *n* contributes to the result for node *n*; in an exclusive scan, it does not.)

Thus, with our same four values,

4 9 7 6

an upward exclusive scan/add would produce

0 4 13 20

while a downward inclusive scan/add would produce

26 22 13 6

Segmented Scans

In a segmented scan, independent scans are run simultaneously on different subgroups (or segments) of the nodes. The segments are determined at run time by an argument called the *sbit* (described later in this chapter). For example, given our four values:

4 9 7 6

and *sbit* values of

1 0 1 0

an upward inclusive segmented scan/add would return

4 13 7 13

9.2 Concatenation

Concatenation simply appends the value from each processor to the values of all preceding processors (in processor identifier order). CMMD provides two versions of concatenation: one concatenates across the nodes only, and writes the resulting value into a buffer on every node. The other concatenates values from every node into a buffer on the host. Concatenation always proceeds from the lowest to the highest node; it is never segmented.

9.3 Reduction Operations

CMMD_reduce_int (*int value, CMMD_combiner_t combiner*)
CMMD_reduce_uint (*unsigned value, CMMD_combiner_t combiner*)
CMMD_reduce_float (*float value, CMMD_combiner_t combiner*)
CMMD_reduce_double (*double value, CMMD_combiner_t combiner*)

value The value to be contributed to the operation. Its type must match that specified by the function name.

combiner One of the symbols listed below, specifying the type of operation to be performed.

For signed integer operands (**CMMD_reduce_int**), allowable combiners are

CMMD_combiner_add	**CMMD_combiner_ior**
CMMD_combiner_max	**CMMD_combiner_xor**
CMMD_combiner_min	**CMMD_combiner_and**

For unsigned integer operands (**CMMD_reduce_uint**), allowable combiners are

CMMD_combiner_uadd	**CMMD_combiner_ior**
CMMD_combiner_umax	**CMMD_combiner_xor**
CMMD_combiner_umin	**CMMD_combiner_and**

For float and double operands (**CMMD_reduce_float** and **CMMD_reduce_double**), allowable operands are

CMMD_combiner_add
CMMD_combiner_max
CMMD_combiner_min

Using anything other than an allowable combiner causes a fatal error.

The reduce functions return the value of the specified reduce operation over all the nodes. Every node thus receives the same return value. The functions will not return until all nodes have called **CMMD_reduce_***type*. The host processor is not involved. To involve the host processor, use the pair of routines described below, **CMMD_reduce_from_nodes** and **CMMD_reduce_to_host**. Note that these routines must be paired; it is an error to call **CMMD_reduce** on the nodes and **CMMD_reduce_from_nodes** on the host.

CMMD_reduce_from_nodes_int	(*int value, CMMD_combiner_t combiner*)
CMMD_reduce_from_nodes_uint	(*unsigned value, CMMD_combiner_t combiner*)
CMMD_reduce_from_nodes_float	(*float value, CMMD_combiner_t combiner*)
CMMD_reduce_from_nodes_double	(*double value, CMMD_combiner_t combiner*)

CMMD_reduce_to_host_int	(*int value, CMMD_combiner_t combiner*)
CMMD_reduce_to_host_uint	(*unsigned value, CMMD_combiner_t combiner*)
CMMD_reduce_to_host_float	(*float value, CMMD_combiner_t combiner*)
CMMD_reduce_to_host_double	(*double value, CMMD_combiner_t combiner*)

value The value to be contributed by this processor to the operation. Its type must match that specified by the function name.

combiner One of the symbols listed below, specifying the type of operation to be performed.

For signed integer operands, allowable combiners are

CMMD_combiner_add	**CMMD_combiner_ior**
CMMD_combiner_max	**CMMD_combiner_xor**
CMMD_combiner_min	**CMMD_combiner_and**

For unsigned integer operands, allowable combiners are

CMMD_combiner_uadd	**CMMD_combiner_ior**
CMMD_combiner_umax	**CMMD_combiner_xor**
CMMD_combiner_umin	**CMMD_combiner_and**

For float and double operands, allowable operands are

CMMD_combiner_add
CMMD_combiner_max
CMMD_combiner_min

Using anything other than an allowable combiner causes a fatal error.

In this pair of functions, the host calls **CMMD_reduce_from_nodes_***type* and all nodes call **CMMD_reduce_to_host_***type*. The functions return to the host processor and to each node the value of the specified reduce operation over all

the nodes including the host processor. The functions will not return until all nodes have called **CMMD_reduce_to_host_***type* and the host has called **CMMD_reduce_from_nodes_***type*.

9.4 Scan Operations

CMMD_scan_int (*int value, CMMD_combiner_t combiner, CMMD_scan_direction_t direction, CMMD_segment_mode_t smode, int sbit, CMMD_scan_inclusion_t inclusion*)

CMMD_scan_uint (*uint value, CMMD_combiner_t combiner, CMMD_scan_direction_t direction, CMMD_segment_mode_t smode, int sbit, CMMD_scan_inclusion_t inclusion*)

CMMD_scan_float (*float value, CMMD_combiner_t combiner, CMMD_scan_direction_t direction, CMMD_segment_ mode_t smode, int sbit, CMMD_scan_inclusion_t inclusion*)

CMMD_scan_double (*double value, CMMD_combiner_t combiner, CMMD_scan_ direction_t direction, CMMD_segment_mode_t smode, int sbit, CMMD_scan_inclusion_t inclusion*)

value The value to be contributed by this processor to the operation. Its type must match that specified by the function name.

combiner One of the symbols listed below, specifying the type of operation to be performed.

For signed integer operands (**CMMD_scan_int**), allowable combiners are

CMMD_combiner_add	**CMMD_combiner_ior**
CMMD_combiner_max	**CMMD_combiner_xor**
CMMD_combiner_min	**CMMD_combiner_and**

For unsigned integer operands (**CMMD_scan_uint**), allowable combiners are

CMMD_combiner_uadd	**CMMD_combiner_ior**
CMMD_combiner_umax	**CMMD_combiner_xor**
CMMD_combiner_umin	**CMMD_combiner_and**

For float and double operands (**CMMD_scan_float** and **CMMD_reduce_double**), allowable operands are

CMMD_combiner_add
CMMD_combiner_max
CMMD_combiner_min

Using anything other than an allowable combiner causes a fatal error.

direction

CMMD_upward
The scan starts at node 0 and proceeds to the highest-numbered node.

CMMD_downward
The scan starts at the highest-numbered node and proceeds downward to node 0.

smode

CMMD_none
The scan proceeds across all nodes.

CMMD_segment_bit
The scan is a segmented scan, with *sbit* acting as a segment bit.

CMMD_start_bit
The scan is a segmented scan, with *sbit* acting as a start bit.

sbit

If *sbit* is nonzero, the node marks the boundary (usually the beginning) of a segment; if *sbit* is zero, the node is not a boundary marker. (If *smode* is **CMMD_none**, then *sbit* is ignored.)

inclusion

CMMD_inclusive
The scan is inclusive.

CMMD_exclusive
The scan is exclusive.

CMMD_scan_*type* returns the value of the specified scan operation over all the nodes. This function does not return until all nodes have called the function. The host processor is not involved.

PLEASE NOTE

(1) Values for *direction*, *smode*, and *inclusion* MUST be identical across all nodes. Otherwise, results are unpredictable and the program may crash.

(2) For **CMMD_scan_float** and **CMMD_scan_double**, the combination of *smode* = **CMMD_start_bit** and *inclusion* = **CMMD_exclusive** is currently illegal and will cause the nodes to exit.

Direction and Inclusion

The direction argument determines the direction of the scan, either upwards (from 0 to the highest-numbered node) or downwards (from the highest-numbered node to 0). The inclusion argument determines whether a given node participates in its own value. When *smode* is **CMMD_none**, these two arguments alone work together to define which source values affect the destination value in a given processor.

- In inclusive upward scans, the value returned for a given node *n* is the combination of the source values in all nodes <= *n*.
- In inclusive downward scans, the value returned for a given node *n* is the combination of the source values in all nodes >= *n*.
- In exclusive upward scans, the value returned for a given node *n* is the combination of source values in all nodes < *n*. The first (lowest-numbered) node receives the identity value for the combiner.
- In exclusive downward scans, the value returned for a given node *n* is the combination of the source values in all nodes >*n*. The highest-numbered node receives the identity value for the combiner.

If a scan is a segmented scan, these rules apply on a per-segment basis, as explained below.

Smode and Sbit

The *smode* and *sbit* arguments define segmented scans. These are scans that tally their results across subgroups of the nodes. Every node belongs to one group, or "segment," with the group to which it belongs determined by *smode* and *sbit* as follows:

- When *smode* is `CMMD_segment_bit`

  If *smode* is `CMMD_segment_bit`, then *sbit* is considered a *segment bit.* A nonzero segment bit starts a new segment for an upward scan, but ends a segment for a downward scan. Imagine 8 nodes with the following segment bits:

  0 0 1 0 0 1 0 0

  Both upward and downward scans would have 3 segments: one would include nodes 0 and 1, another would include nodes 2–4, and a third would include nodes 5–7.

  When *sbit* is a segment bit, operations in one segment never affect the values of elements in another segment. Thus, given segment bits of

  0 0 1 0

  and values of

  4 1 5 2

  an upward exclusive max would produce

  0 4 0 5

  (See Figure 5.)

- When *smode* is `CMMD_start_bit`

  If *smode* is `CMMD_start_bit`, then *sbit* is considered a *start bit.* A nonzero start bit always starts a new segment, whether the direction of the scan is upward or downward. Thus, given 8 nodes with the following start bits:

  0 0 1 0 0 1 0 0

  an upward scan would have the same segments as the segmented scan shown above (0–1, 2–4, 5–7); but a downward scan would have segments of 7–6, 5–3, and 2–0.

In addition, if the operation is exclusive, a node with a nonzero start bit *does* receive a value from the preceding segment. The value received is the reduce of the previous segment, with the same combiner. Thus, given start bits of

0 0 1 0

and values of

4 1 5 2

an upward exclusive scan/max would produce

0 4 4 5

(See Figure 5.)

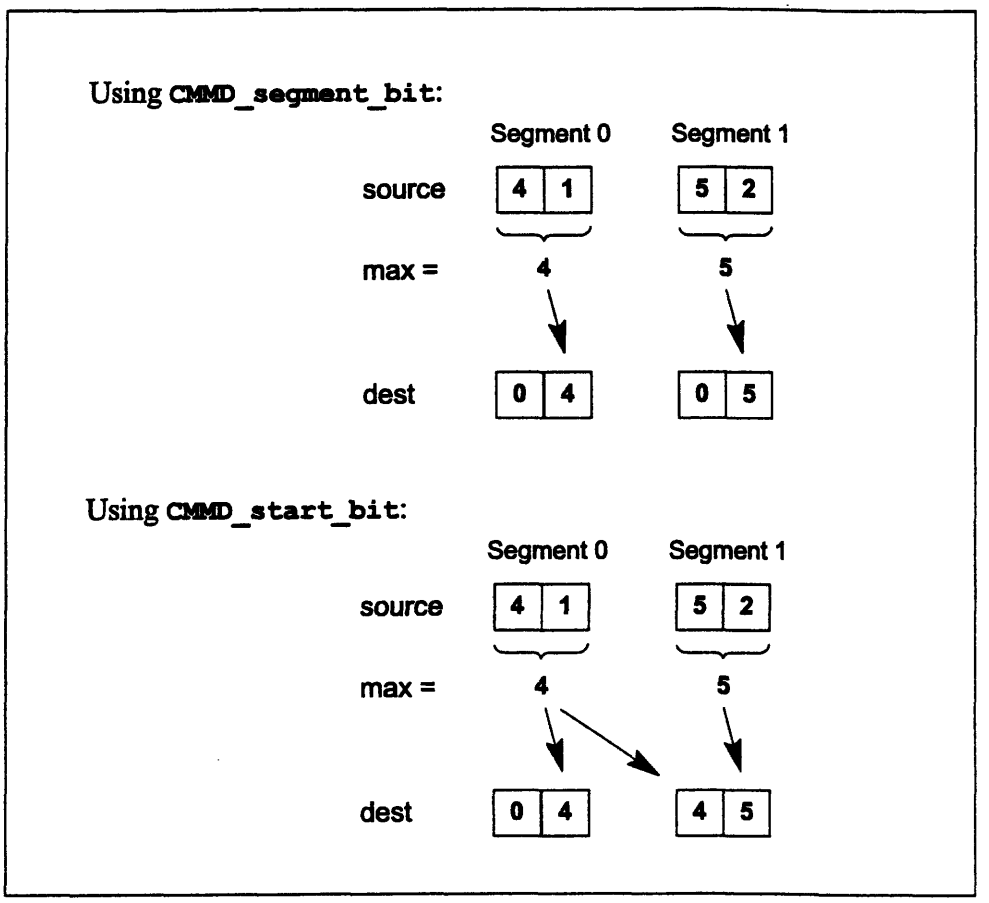


Figure 5. Upward exclusive scans with max combiners.

9.5 Concatenation Operations

CMMD_concat_with_nodes (*void *element, void *buffer, int elem_length*)

**element*	A pointer to the element this node contributes to the concatenation process.
**buffer*	A pointer to the buffer in which the returned value will be stored. Its length in bytes must be at least (**CMMD_partition_size()** * *elem_length*).
elem_length	The length in bytes of the element to be concatenated. Must be identical across all nodes.

CMMD_concat_with_nodes concatenates elements of equal length from each node into the given buffer. The length of the buffer in bytes must be at least (**CMMD_partition_size ()** * *elem_length*). This function does not return until all nodes have called **CMMD_concat_with_nodes**. The host processor is not involved.

CMMD_gather_from_nodes (*void *buffer, int elem_length*)
CMMD_concat_elements_to_host (*void *element, int elem_length*)

**buffer*	(Host only.) A pointer to the buffer in which the returned value will be stored. Its length in bytes must be at least (**CMMD_partition_size()** * *elem_length*).
**element*	(Nodes only.) A pointer to the element this node contributes to the concatenation process.
elem_length	The length in bytes of the element to be concatenated. (Must be identical for all processors.)

This pair of functions concatenates elements from each node into a buffer on the host. The element length must be identical for all processors, and the host must specify enough space to store the result. The function returns after all nodes have called **CMMD_concat_elements_to_host** and the host has called **CMMD_gather_from_nodes**.

Note that these functions are essentially the opposite of the functions **CMMD_distrib_to_nodes** and **CMMD_receive_element_from_host**. That pair distributes the contents of a buffer element-wise from the host to the nodes; this pair gathers the elements from the nodes into a buffer on the host.

Appendix A

Routines That Let You Create Your Own Protocol

PLEASE NOTE

(1) The routines documented in this appendix, **CMMD_send_packet** and **CMMD_receive_packet**, cannot be used in conjunction with other CMMD send and receive routines. The library provides no protection against doing so, but results are likely to be indeterminate. CMMD global functions, on the other hand, can be used with these packet routines.

(2) Creating a message-passing protocol is not a simple operation. Deadlocks are not only possible, they are extremely likely. Please do not use these routines unless you have very good reasons for doing so, and are experienced at message-passing multiprocessor programming.

Using **CMMD_send_packet** and **CMMD_receive_packet**, nodes can send and receive non-blocking messages of up to 20 bytes in length. The routines provide no synchronicity, nor any functionality to verify whether a message, once sent, is received somewhere. Users employing these routines must ensure that any messages sent by them are received; unreceived messages can clog the data network and cause the program to hang.

CMMD_send_packet and **CMMD_receive_packet** make no provision for headers. Users must create their own headers and their own software to parse whatever header-and-text combination they decide to use.

A.1 The Packet Routines

CMMD_send_packet (*unsigned int destination, int words_to_send, unsigned int *buffer, unsigned int type*)

destination	An integer identifying the node to which the message is to be sent.
words_to_send	The length of the buffer, in 32-bit words.
**buffer*	A pointer to a buffer that contains the message to be sent.
type	At this release, 0 is the only allowable value for this argument.

This function sends out a message to the destination node. Arguments specify the length (expressed in 32-bit words) of the packet, and the starting address of the message.

The function is non-blocking. It does not wait for any acknowledgment from the receiver. It returns **TRUE** if the message has been sent into the communications network, **FALSE** otherwise.

CMMD_receive_packet (*unsigned int *buffer*)

**buffer*	A pointer to a buffer that contains the message to be received.

The function checks for incoming messages. If it finds one, it receives the message, writes it into the buffer, and returns the number of words received. If it finds no incoming message, it returns –1.

Index

N

O

P

R

S

T

V

X