Chapter 8

Global Communication

CMMD provides functions for a number of types of global communication:

- broadcast operations: **CMMD_bc_to_nodes** **CMMD_receive_bc_from_node**
- scans (parallel prefix): **CMMD_scan_{type}** **CMMD_scan_v**
- reduction: **CMMD_reduce_{type}** **CMMD_reduce_v**
- concatenation: **CMMD_concat_with_nodes**

CMMD global functions take advantage of the CM-5's hardware support for global communications.

There are some important facts to note about these operations:

- Global functions require that all nodes in the partition execute the same operation simultaneously. If any node fails to execute the appropriate global operation, the remaining nodes are unable to proceed. (Chapter 15 describes a method for defining a logical partition containing only a subset of the nodes in the physical partition, should this be required.)
- At present, global operations do not support parallel array arguments. The array arguments to a global function must be either serial arrays or vector serial arrays.
- CMMD expects typed data, such as that specified for global functions, to be aligned on its natural data-type boundaries. In particular, double-precision data must be doubleword-aligned.
- If interrupts are enabled, then asynchronous operations (arrival of messages from the network) will interrupt global operations.

If interrupts are not enabled, message-passing calls involving a node that has called a global operation may not complete until after the global operation is complete. This should be a problem only if lack of completion of the message-passing operation prevents some other node from reaching the global operation. In that case, both operations will hang.

8.1 Broadcast Functions

In a broadcast, a message is sent from a single source to all nodes. Broadcasts are global operations; they synchronize all nodes.

CMMD offers several facilities for broadcasting messages:

- One node can send data to all other nodes, via a broadcast message.
- Data read from a file or I/O device can be broadcast to all nodes. (See Chapter 12.)
- In host/node programs, the host can broadcast data to all nodes. (Host/node programs are described in Chapter 16.)

This chapter discusses broadcasts in which one node sends a message to all other nodes.

All nodes must take part in these broadcasts. One node must signal its intention to send the broadcast message; all other nodes must signal their readiness to receive the message.

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

A broadcast is a blocking operation. Once a broadcast is signaled, the hardware begins checking for responses. Only when the hardware signals that the entire broadcast is complete can any of the broadcast calls return.

Note: Asynchronous operations (arrival of messages from the network, etc.) will interrupt broadcasts, as well as other global operations.

8.1.1 Node Broadcast Functions

FUNCTIONS **CMMD_bc_to_nodes**
CMMD_receive_bc_from_node

C SYNTAX

void **CMMD_bc_to_nodes**(void **buffer*, int *buffer_desc*)
void **CMMD_receive_bc_from_node**(void **buffer*, int *buffer_desc*)

FORTRAN SYNTAX

SUBROUTINE **CMMD_bc_to_nodes**(*buffer, buffer_desc*)
<array> *buffer*
integer *buffer_desc*

SUBROUTINE **CMMD_receive_bc_from_node**(*buffer, buffer_desc*)
<array> *buffer*
integer *buffer_desc*

ARGUMENTS

buffer The buffer that holds the message being broadcast (from the sending node) and received (on all other nodes).

buffer_desc The length, in bytes, of the buffer being broadcast (from the sending node) and received (on all other nodes).

RETURNS

The broadcast functions do not return until the broadcast is complete; that is, until the all the nodes have made their calls and the data transfer has taken place. There is no return value.

DESCRIPTION

Any node can call **CMMD_bc_to_node** to broadcast a buffer of the specified length (in bytes) to all other nodes. All other nodes must then call **CMMD_receive_bc_from_node**, with the same length argument, to receive the buffer. At the completion of the operation, all nodes contain identical information in the specified buffers.

IMPORTANT

(1) There is no protection against having more than one broadcaster. Multiple broadcasts are combined in undefined ways, and generally are not useful.

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

(3) All processors within the partition must take part in this operation. Otherwise, the operation will fail and the program may hang.

(4) Broadcasts cannot call or be called from handler functions.

8.2 Scans, Reductions, and Concatenation

Scans, reductions, and concatenation are global operations. All nodes in the partition take part in these calls: each node provides a buffer containing source data, and each receives the result of the operation.

Reduces and scans perform arithmetic or logical operations. Given a buffer containing some value (or vector of values) on 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

Reduction operations return one final value to all nodes. Scans return the cumulative results as a running tally across all nodes.

Concatenation is simply cumulative: the contents of all nodes' source buffers are appended to each other, in processor order, within the destination buffers on each node.

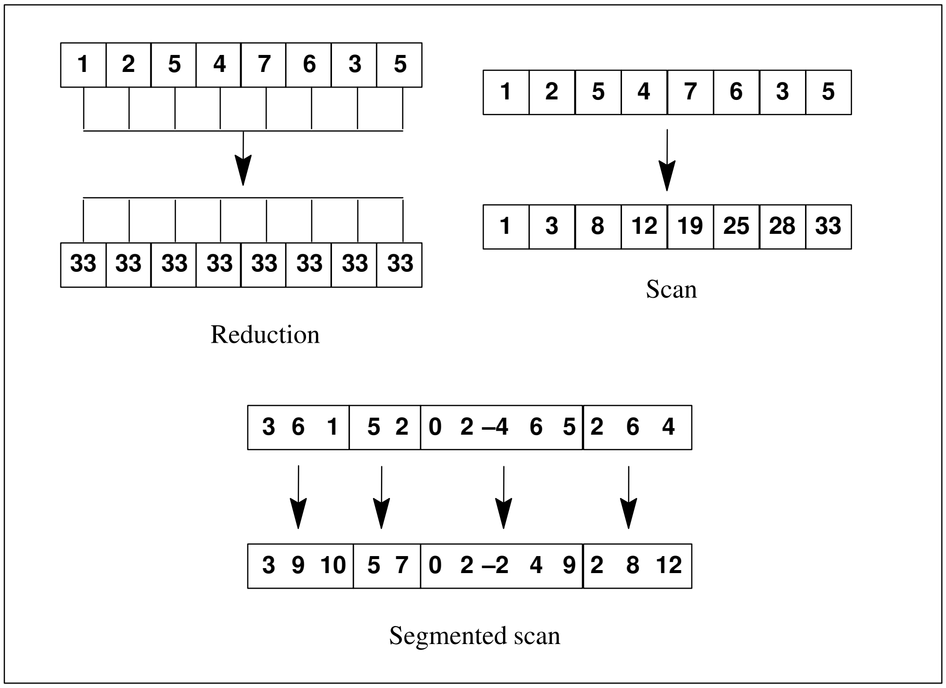

Figure 10. Reductions and scans (unsegmented and segmented).

Reduction operations, scans, and segmented scans provide three basic methods of all-to-all communication. See Figure 10. (Host/node programs can also use reduction operations to provide all-to-one communication, with all nodes reducing data to the host.)

Scan and reduction functions may perform one of a number of operations:

`add`	Summation
`max`	Return the largest value found
`min`	Return the smallest value found
`ior`	Inclusive OR operation
`xor`	Exclusive OR operation
`and`	Logical AND operation

The operation to be performed by a particular function is specified as an argument to the function. For example, a **`CMMD_reduce_int`** function, called with the **`CMMD_combiner_max`** argument, would compare integer values on all nodes and return the largest integer to all nodes.

Reductions

A reduce operation starts with values in every processor and ends with a single value in every node. 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
```

A vector reduce operation starts with a vector of values on each node, and creates (on each node) a new vector in which each specified source element has been replaced by the result of the reduction across all nodes. The new vectors may be in a new location or may overwrite the source vectors.

Thus, if four nodes held the vectors

```
[1 2 3 4]    [2 3 4 5]    [3 4 5 6]    [4 5 6 7]
```

a reduce/add without striding would create the vectors

```
[10 14 18 22] [10 14 18 22] [10 14 18 22] [10 14 18 22]
```

while a reduce/add with a stride double the element length, overwriting its source buffer, would create the vectors

```
[10 2 18 4]   [10 3 18 5]   [10 4 18 6]   [10 5 18 7]
```

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
```

Vector scans scan across the nodes for each specified element of the vector. Given the two-element vectors

```
  [0 1]   [2 3]   [4 5]   [6 7]
```

an upward inclusive scan/add would produce

```
  [0 1]   [2 4]   [6 9]   [12 16]
```

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

```
         0    1    0    1
```

an upward inclusive segmented scan/add would return

```
         4   13    7   13
```

Segmented vector scans follow the same rules as segmented non-vector scans.

Concatenation

Concatenation simply appends the value from each processor to the values of all preceeding processors (in processor identifier order). Concatenation always proceeds from the lowest to the highest node; it is never segmented.

For example, given the four values

```
a    b    c    d
```

concatenation would produce

```
abcd    abcd    abcd    abcd
```

8.2.1 Reduction Functions

NOTE

Please note that the interfaces for vector and non-vector reduction functions differ somewhat:

- Each data type uses a separate non-vector function; all data types share a single vector function. (In the vector function, the combiner provides the data type.)
- Non-vector functions return their results as the return value of the function. Vector functions have no return values; they write their results into a buffer supplied as an argument to the function.

Please note also that single-precision reductions produce double-precision results; in all other cases, the result data type matches the input data type.

Reduction to All Nodes

FUNCTIONS **CMMD_reduce_***type*

C SYNTAX

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

FORTRAN SYNTAX

INTEGER FUNCTION **CMMD_reduce_int(***value, combiner***)**
integer *value*
integer *combiner*

REAL*8 FUNCTION **CMMD_reduce_float(***value, combiner***)**
real*4 *value*
integer *combiner*

REAL*8 FUNCTION **CMMD_reduce_double(***value, combiner***)**
real*8 *value*
integer *combiner*

ARGUMENTS

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, 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 operands, allowable combiners (see note, below) are

CMMD_combiner_fadd
CMMD_combiner_fmax
CMMD_combiner_fmin

For double operands, allowable combiners (see note, below) are

CMMD_combiner_dadd
CMMD_combiner_dmax
CMMD_combiner_dmin

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

Note: For back compatibility, float and double operands may also use the combiners **CMMD_combiner_add**, **CMMD_combiner_max**, and **CMMD_combiner_min**. The use of these combiners for such operands is deprecated however, and its continued support is not ensured.

RETURNS

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 do not return until all nodes have called **CMMD_reduce_***type*.

HANDLERS

The global functions cannot call or be called from handler functions.

Vector Reduction to All Nodes

FUNCTION **CMMD_reduce_v**

C SYNTAX

```
void CMMD_reduce_v(void *dest, void *source,
                   CMMD_combiner_t combiner,
                   int stride, int elem_count)
```

FORTRAN SYNTAX

```
SUBROUTINE CMMD_reduce_v(dest, source, combiner, stride,
                         elem_count)
          <array> dest, source
          integer combiner, stride, elem_count
```

ARGUMENTS

dest The buffer that will hold the vector of reduced values. It must be a serial array, not a parallel one.

source The local buffer that holds the vector being reduced. It must be a serial array, not a parallel one.

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 operands, allowable combiners are

`CMMD_combiner_fadd`
`CMMD_combiner_fmax`
`CMMD_combiner_fmin`

For double operands, allowable combiners are

`CMMD_combiner_dadd`
`CMMD_combiner_dmax`
`CMMD_combiner_dmin`

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

stride — An integer specifying the distance in bytes between the starting addresses of vector elements. Should be a multiple of the relevant data type.

elem_count — A positive integer specifying the number of elements in the vector.

RETURNS

This function does not return until all nodes have called it, and it does not return a value.

DESCRIPTION

The vector reduce function computes the value of the specified reduce operation for each specified source buffer element over all the nodes. Every node thus computes the same set of values. The function does not return until all nodes have called **`CMMD_reduce_v`**.

The vector reduce functions do not return values. Rather, each value computed from a *source* buffer element is stored in the corresponding *dest* buffer element. The *source* and *dest* arguments may be identical, in which case the results overwrite the *source* buffer. If *source* and *dest* overlap in any other way, the results are not predictable.

Note that the data type for vector reductions is not specified by function name, but only by the combiner.

If *elem_count* is one, you may mix **`CMMD_reduce_`***type* operations with the vector operation; If *elem_count* is more than one, however, mixing vector and non-vector operations will cause the system to hang.

An example. As an example of vector reduction, let us consider the case of a 4 x 6 matrix **`A`**, filled with unsigned integers from 0 to 23, laid out in memory as follows:

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

To sum the first column of the matrix across all nodes, you would do a vector reduction with combiner **`CMMD_combiner_uadd`**, stating that you were reducing four elements (*elem_count*), each located six times the number of bytes in a *uint* apart (stride):

```
CMMD_reduce_v
   (&A, &A, CMMD_combiner_uadd, 6*sizeof(uint), 4)
```

If there were *P* processing nodes in the partition and array **`A`** contained the same data in each, then the first element in the second row of **`A`** would be replaced by 6**P* in every node, the first element in the third row would be replaced by 12**P*, and the first element in the last row would be replaced by 18**P*. All other elements would remain unchanged.

Handlers. The global functions can neither call nor be called from handler functions.

8.2.2 Scan Functions

NOTE

Please note that the interfaces for vector and non-vector scan functions differ somewhat:

- Each data type uses a separate non-vector function; all data types share a single vector function.
- Non-vector functions return their results as the return value of the function. Vector functions have no return values; they write their results into a buffer supplied as an argument to the function.

Please note also that single-precision scans produce double-precision results; in all other cases, the result data type matches the input data type.

8.2.3 Non-Vector Scan

FUNCTIONS **CMMD_scan_***type*

C SYNTAX

int **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***)**

uint **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*)

double **`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*`)`

double **`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*`)`

FORTRAN SYNTAX

INTEGER FUNCTION **`CMMD_scan_int(`***value, combiner, direction, smode, sbit, inclusion*)
integer *value*
integer *combiner, direction, smode, sbit, inclusion*

REAL*8 FUNCTION **`CMMD_scan_float(`***value, combiner, direction, smode, sbit, inclusion*`)`
real*4 *value*
integer *combiner, direction, smode, sbit, inclusion*

REAL*8 FUNCTION **`CMMD_scan_double(`***value, combiner, direction, smode, sbit, inclusion*`)`
real*8 *value*
integer *combiner, direction, smode, sbit, inclusion*

ARGUMENTS

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

*combine*r — 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 operands (see note, below), allowable combiners are

CMMD_combiner_fadd
CMMD_combiner_fmax
CMMD_combiner_fmin

For double operands (see note, below), allowable combiners are

CMMD_combiner_dadd
CMMD_combiner_dmax
CMMD_combiner_dmin

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

Note: For back compatibility, float and double operands may also use the combiners **CMMD_combiner_add**, **CMMD_combiner_max**, and **CMMD_combiner_min**. The use of these combiners for such operands is deprecated however, and its continued support is not ensured.

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

sbit	If *sbit* is non-zero, the node marks the boundary (usually the beginning) of a segment; if sbit is 0 the node is not a boundary marker. (If *smode* is set to **`CMMD_none`**, *sbit* is ignored.)	
inclusion	**`CMMD_inclusive`**	The scan is inclusive.
	`CMMD_exclusive`	The scan is exclusive.

RETURNS

`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.

DESCRIPTION

`CMMD_scan_`*type* returns the value of the specified scan operation over all the nodes.

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 causes the nodes to exit.

Direction and Inclusion. The *direction* argument determines the direction of the scan, either upward (from 0 to the highest-numbered node) or downward (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 $\leqq N$.

- In inclusive downward scans, the value returned for a given node *N* is the combination of the source values in all nodes $\geqq N$.
- In exclusive upward scans, the value returned for a given node *N* is the combination of source values in all processors $< N$. 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 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:

smode = CMMD_segment_bit: If *smode* is **`CMMD_segment_bit`**, then *sbit* is considered a segment bit. A non-zero 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
```

smode = CMMD_start_bit. If *smode* is **`CMMD_start_bit`**, then *sbit* is considered a start bit. A non-zero 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 non-zero 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 max would produce

```
0   4   4   5
```

Handlers. The global functions can neither call nor be called from handler functions.

8.2.4 Vector Scans

FUNCTION **CMMD_scan_v**

C SYNTAX

void **CMMD_scan_v(**void **dest,* void **source*,
CMMD_combiner_t *combiner*,
CMMD_scan_direction_t *direction*,
CMMD_segment_mode_t *smode*, int *sbit*,
CMMD_scan_inclusion_t *inclusion*,
int *stride*, int *elem_count***)**

FORTRAN SYNTAX

SUBROUTINE **CMMD_scan_v(***dest, source, combiner, direction, smode, sbit, inclusion, stride, elem_count***)**
<array> *dest, source*
integer *combiner, direction, smode, sbit, inclusion*
integer *stride, elem_count*

ARGUMENTS

dest — The buffer that will hold the vector of computed values. It must be a serial array, not a parallel one.

source — The buffer that holds the vector being scanned. It must be a serial array, not a parallel one.

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 operands, allowable combiners are

`CMMD_combiner_fadd`
`CMMD_combiner_fmax`
`CMMD_combiner_fmin`

For double operands, allowable combiners are

`CMMD_combiner_dadd`
`CMMD_combiner_dmax`
`CMMD_combiner_dmin`

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 the *sbi*t acting as a segment bit.
	`CMMD_start_bit`	The scan is a segmented scan, with the *sbit* acting as a start bit.

sbit — If *sbit* is non-zero, the node marks the boundary (usually the beginning) of a segment; if *sbit* is 0 the node is not a boundary marker.

(If *smode* is set to **`CMMD_none`**, *sbit* is ignored.)

inclusion	**`CMMD_inclusive`**	The scan is inclusive.
	`CMMD_exclusive`	The scan is exclusive.

stride — An integer specifying the distance in bytes between the starting addresses of vector elements.

elem_count — An integer specifying the number of elements in the vector.

RETURNS

This function does not return until all nodes have made the function call. It has no return value.

DESCRIPTION

The vector scan function computes the value of the specified scan operation for each specified buffer element over all the nodes. This function does not return until all nodes have called the function.

The rules for *direction*, *smode*, *sbit*, and *inclusion* are the same as for non-vector scans.

No value is returned by this function. Instead, each value computed from a *source* buffer element is stored in the corresponding *dest* buffer element. The *source* and *dest* buffers may be identical, in which case the new vector overwrites the previous one. If *source* and *dest* overlap in any other way, the results are not predictable. s*ource* and *dest* must be serial buffers, not parallel ones.

Note that the type of vector scan is not specified by function name. Scan type is specified only by the combiner.

If *elem_count* is one, then you may mix **`CMMD_scan_`***type* operations with the vector operation. If *elem_count* is greater than one, however, mixing vector and non-vector scans causes the system to hang.

Handlers. The global functions can neither call nor be called by handler functions.

8.2.5 Concatenation Function

FUNCTION **`CMMD_concat_with_nodes`**

C SYNTAX

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

FORTRAN SYNTAX

SUBROUTINE **`CMMD_concat_with_nodes(`***element, buffer,*
*elem_length***`)`**

<array> e*lement, buffe*r
integer *elem_length*

ARGUMENTS

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

RETURNS

This function does not return until all nodes in the partition have made the function call. It has no return value.

DESCRIPTION

`CMMD_concat_with_nodes` concatenates elements of equal length from each node into the given (serial) 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`**.

Handlers. The global functions can neither call nor be called from handler functions.