Chapter 13

Communication with Computation

This chapter discusses C* library functions that let you perform computations on parallel values that are being transmitted. Most of these functions use grid communication. The functions differ in these ways:

- *The kinds of computation that are available for each function.* See Section 13.1.
- *The way in which parallel variable elements are selected.* For example, some functions let you divide the parallel variable elements into groups called *scan classes*. You can then operate on each scan class independently. See Section 13.2.
- *The way in which the function reports the results of the computation.* For example, **`scan`** provides a running total of its computations; **`spread`** provides only the final result.

Include the file **`<cscomm.h>`** when calling any of the functions discussed in this chapter.

13.1 What Kinds of Computation?

The **`scan`**, **`reduce`**, **`spread`**, **`multispread`**, and **`global`** functions let you specify a *combiner type* that indicates the kind of computation or combining you want carried out on the parallel data. Each of these functions is overloaded for some subset of the combiner types listed in Table 4.

Table 4. Combiner types.

Combiner	Meaning
CMC_combiner_max	Take the largest value among the specified parallel variable elements.
CMC_combiner_min	Take the smallest value among the specified elements.
CMC_combiner_add	Add the values of the specified elements.
CMC_combiner_copy	Copy the values of the specified elements.
CMC_combiner_multiply	Multiply the values of the specified elements.
CMC_combiner_logior	Perform a bitwise logical inclusive OR on the specified elements.
CMC_combiner_logxor	Perform a bitwise logical exclusive OR on the specified elements.
CMC_combiner_logand	Perform a bitwise logical AND on the specified elements.

These combiner types are also used by the **send** function, which is described in the next chapter.

13.2 Choosing Elements

Several of the C* functions discussed in this chapter provide methods for choosing the subsets of parallel variable elements on which they are to operate. The terminology we use in referring to these subsets of elements comes from **scan**, which is the most general of the functions that use these methods.

13.2.1 The Scan Class

Two positions belong to the same *scan class* if their coordinates differ only along a specified axis. These functions use the concept of a scan class: **scan**, **reduce**, **copy_reduce**, **spread**, **copy_spread**, **enumerate**, **rank**, and **multispread**.

To see how scan classes work, consider the 2-dimensional shape shown in Figure 63.

Note for users of CM-200 C*: This and other shapes in this chapter are smaller than legal size in the CM-200 implementation of C*, so that they are easy to visualize.

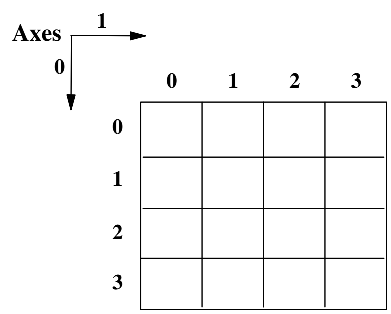

Figure 63. A 4-by-4 shape.

If you specify axis 0 as an argument to one of the functions listed above, you get the scan classes shown in Figure 64. Positions [0][0], [1][0], [2][0], and [3][0] differ only in their coordinates for axis 0; therefore, they belong to the same scan class. Position [0][1] does not belong to this scan class, because it has a different axis 1 coordinate; it belongs to a scan class with positions [1][1], [2][1], and [3][1].

Thus, specifying axis 0 for this shape creates four separate scan classes, each of which is a column of positions through axis 0 in the shape. Functions like `scan` operate on each of these scan classes independently.

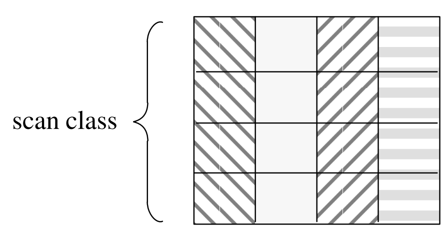

Figure 64. Scan classes for axis 0 of a 2-dimensional shape.

Specifying axis 1, on the other hand, creates four different scan classes, each one consisting of a row of positions through axis 1 in the shape, as shown in Figure 65.

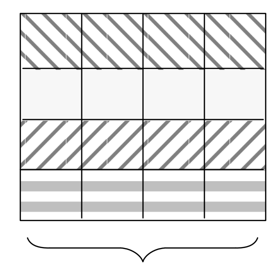

Figure 65. Scan classes for axis 1 of a 2-dimensional shape.

If you have a 1-dimensional shape, there is, of course, only one axis you can specify, and only one scan class for the shape. You can, however, subdivide a scan class, as we discuss below.

If you have a 3-dimensional shape, specifying an axis gives you a *set* of scan classes consisting of the rows of positions that cross this axis. For example, in a 2-by-2-by-2 shape, specifying axis 0 creates these four scan classes:

[0][0][0] and [1][0][0]

[0][1][0] and [1][1][0]

[0][0][1] and [1][0][1]

[0[[1][1] and [1][1][1]

To operate on more than one dimension in a multi-dimensional shape (for example, on planes of positions instead of rows of positions), you must use the **`multispread`** or **`copy_multispread`** function; these functions are discussed in Section 13.8.

The Scan Subclass

Only active positions participate in computations within a scan class. The active positions within a scan class are referred to as the *scan subclass*.

13.2.2 The Scan Set

There may be times when you want a function to operate independently on different parts of a scan subclass. The **`scan`**, **`enumerate`**, and **`rank`** functions let you do this by subdividing a scan subclass into *scan sets*.

To create scan sets, declare a **`bool`**-size parallel variable of the shape on which the function is to operate, and initialize it to 0. This parallel variable is referred to as the *sbit*; it is used as the **`sbit`** argument to the functions listed above. Assign a 1 to an element of this parallel variable to mark the beginning of a scan set at that element's position. In the simplest case, the scan set for each position starts either at the beginning of the scan subclass, or at the nearest position below it in the scan subclass that has its sbit set to 1.

Figure 66 shows a 1-dimensional shape divided into scan sets. In the figure, the scan set for position 1, for example, consists of positions 0 and 1 (the scan subclass starts at position 0, so the scan set starts there also, even if the sbit for that position isn't set to 1). The scan set for position 7 consists of positions 5, 6, and 7, since **`[5]sbit`** is set to 1, thus starting a new scan set.

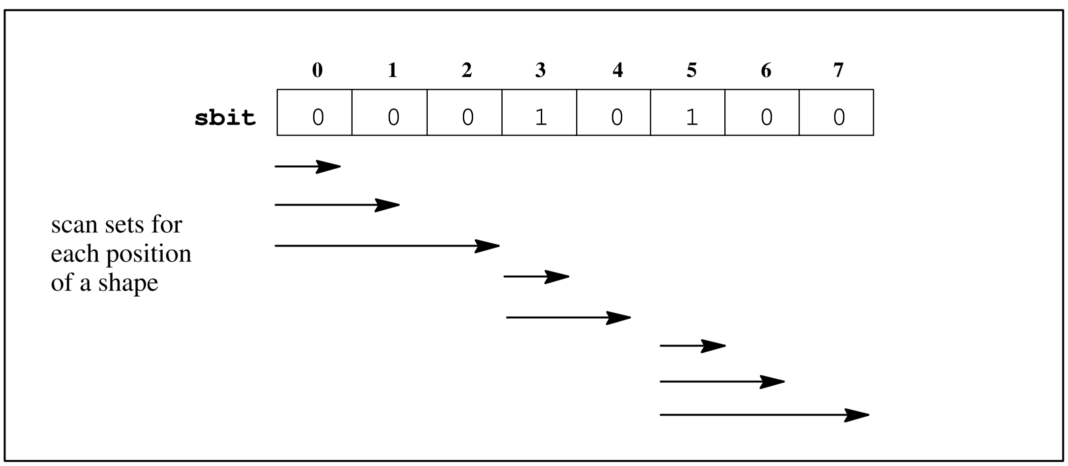

Figure 66. Scan sets in a 1-dimensional shape.

Note than scan sets include only active positions; see Section 13.2.3, however, for a more in-depth discussion of inactive positions and scan sets.

To show how scan sets work, let's use an example in which we keep a running total of the values in the parallel variable **`data`** (this is a scan operation, as discussed in Section 13.3). The results are shown in Figure 67.

	0	1	2	3	4	5	6	7
sbit	0	0	0	1	0	1	0	0
data	0	1	2	3	4	5	6	7
running_total	0	1	3	3	7	5	11	18

Figure 67. An operation that provides a running total, using scan sets.

In the example, **`[1]running_total`** contains the sum of **`[0]data`** and **`[1]data`**, since 0 and 1 are the positions in its scan set. **`[3]running_total`** contains only the value in **`[3]data`**, since **`[3]sbit`** is set to 1, thus starting a new scan set in this position.

You actually have more flexibility than this in how you can divide up scan subclasses:

- Whether an operation is *inclusive* or *exclusive* affects the way scan sets are interpreted; see "Inclusive and Exclusive Operations," below. The example in Figure 67 shows an inclusive operation.
- There are two ways of interpreting the sbit; see Section 13.2.3. In particular, this affects the way scan classes are divided when there are inactive positions, and when an operation proceeds in a downward direction. The example in Figure 67 shows an operation that proceeds in an upward direction.

Inclusive and Exclusive Operations

The way in which scan sets work when you are performing a particular operation depends on whether the operation is *inclusive* or *exclusive*. (NOTE: In this section, we are ignoring the effect of *segment bits* and *start bits*; these are discussed in the next section.)

In an *inclusive* operation (specified by **`CMC_inclusive`**), an element participates in the operation for its position—in other words, the scan set for a position contains that position. As we mentioned, Figure 67 shows the results of an inclusive operation.

In an *exclusive* operation (specified by **`CMC_exclusive`**), the scan set for an element does not contain the element itself — in other words, it does not participate in the operation for its position. Figure 68 shows the results of an exclusive operation, using the same data as that shown in Figure 67.

	0	1	2	3	4	5	6	7
sbit	0	0	0	1	0	1	0	0
data	0	1	2	3	4	5	6	7
running_total	0	0	1	0	3	0	5	11

Figure 68. An exclusive operation on scan sets.

Note the difference between the two results. In the inclusive operation, for example, **`[2]running_total`** receives the running total for **`[0]data`**, **`[1]data`**, and **`[2]data`**; in the exclusive operation, **`[2]running_total`** receives the running total only for **`[0]data`** and **`[1]data`**. When there are no preceding elements in the scan set (for example, in **`[3]running_total`**), the element receives the identity for the operation.

13.2.3 Segment Bits and Start Bits

There are two different kinds of sbits: *segment bits* and *start bits*. Use the **smode** argument to the **scan**, **enumerate**, or **rank** function to specify which kind of sbit you want, as discussed below.

If smode Is CMC_segment_bit

If the value of the **smode** argument is **CMC_segment_bit**, the sbit is considered a *segment bit*, and it divides a scan subclass into segments, as follows:

- An sbit element set to 1 starts a new segment, whether or not the element appears in an active position.
- The way in which the segment bit divides the scan subclass is not affected by the direction of the operation.
- Operations in one segment never affect values of elements in another segment.

If smode Is CMC_start_bit

If the value of the **smode** argument is **CMC_start_bit**, the sbit is considered a *start bit*, and scan classes are divided as follows:

- An sbit element set to 1 divides a scan subclass *only if its position is active*.
- The division *is* affected by the direction of the operation. When the direction is downward, for example, the division occurs from the higher coordinate to the lower coordinate.
- When an operation is *exclusive*, the position whose sbit element is set to 1 will receive a value from the preceding scan set.

These differences between segment bits and start bits are discussed below.

Inactive Positions

When the sbit is a segment bit, a new scan set is created, even though the position where it starts is inactive. Figure 69 shows an example (the scan sets displayed are for positions [2], [4], and [7]).

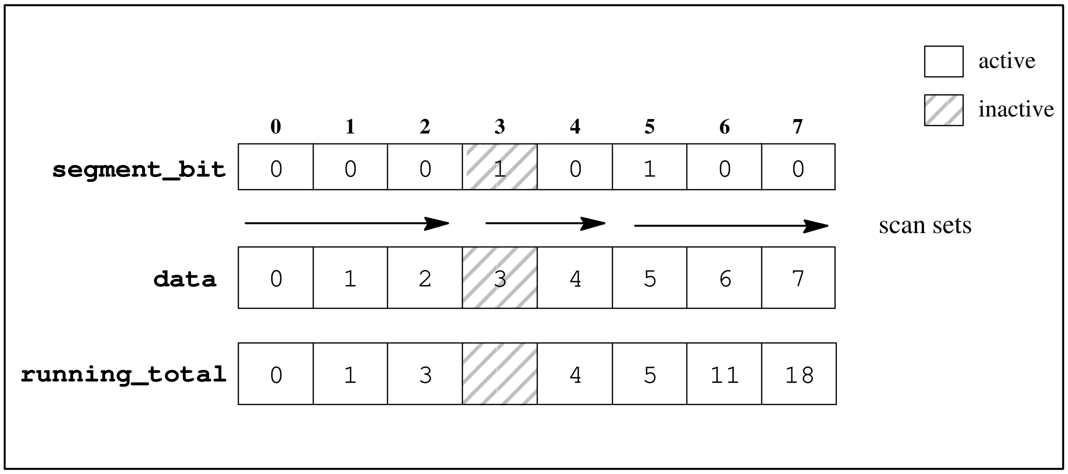

Figure 69. An inclusive operation in an upward direction on segment-bit scan sets, with an inactive position.

Note that position [3] does not participate in the operation, even though it starts a new scan set.

A start bit does not start a scan set if its position is inactive. Figure 70 is an example. Note that the scan set for position [4] begins at position [0], not at position [3], as in Figure 69.

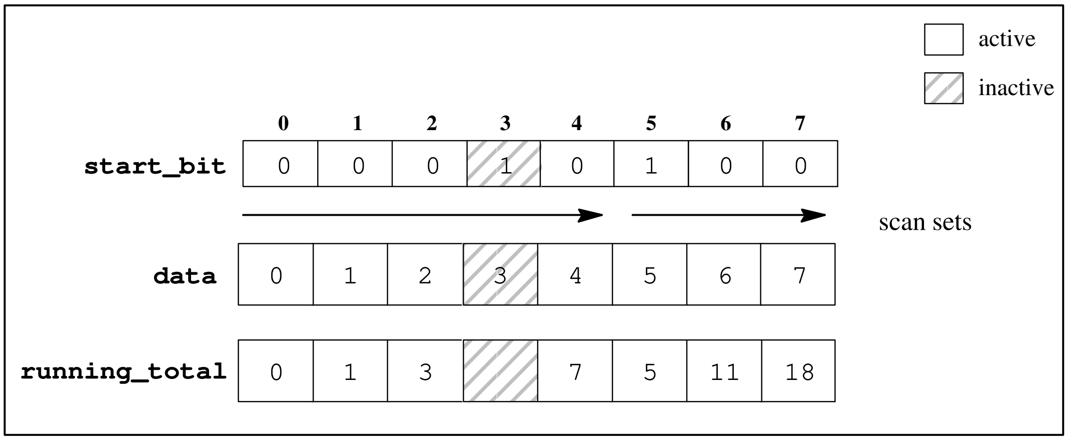

Figure 70. An inclusive operation in an upward direction on start-bit scan sets, with an inactive position.

The Direction of the Operation

When the direction of the operation is *upward*, it proceeds from lower-numbered positions to higher-numbered positions along the scan subclass. Both kinds of sbits divide the scan subclass in the same way when the direction is upward (provided that all positions are active); see Figure 66 for an example. You specify an upward direction with the argument **`CMC_upward`**.

When the direction of the operation is *downward* (specified by the argument **`CMC_downward`**), the operation proceeds from higher-numbered positions to lower-numbered positions along the scan subclass. In this case, segment bits divide the scan subclass in the same way as the sbits shown in Figure 66; however, since the operation proceeds in a downward direction, this means that a segment bit *ends* a scan set, and the operation begins again in the position with the next lowest coordinate. Figure 71 is an example; it shows the scan sets for positions [0], [3], and [5].

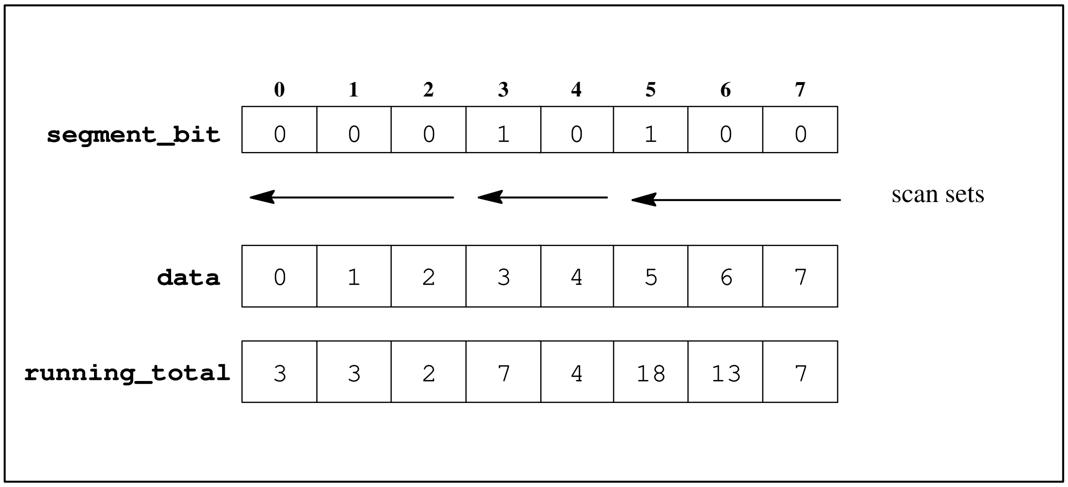

Figure 71. An inclusive operation in a downward direction on segment-bit scan sets.

Start-bit scan sets, however, follow the downward direction; in other words, start bits start scan sets, rather than ending them. Figure 72 is an example; it shows the scan sets for positions [0], [4], and [6].

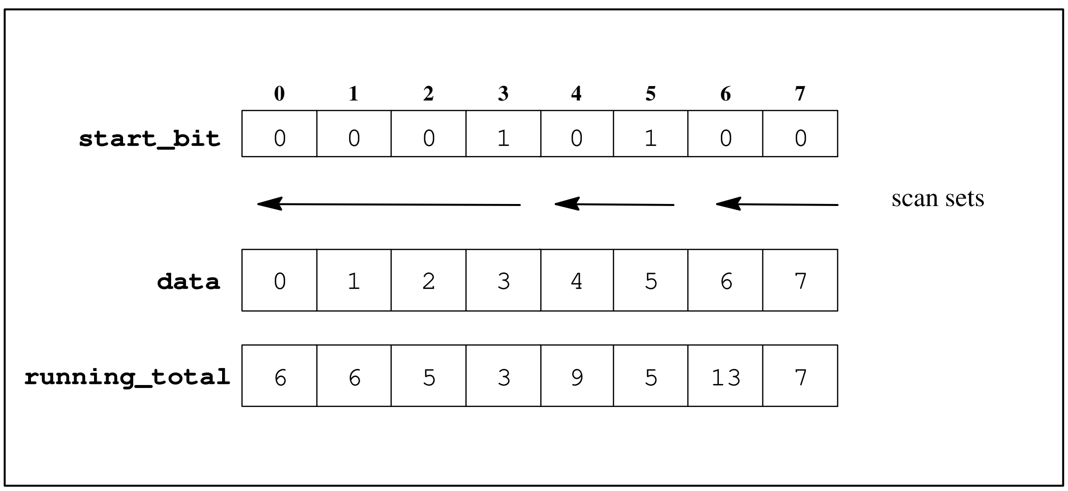

Figure 72. An inclusive operation in a downward direction on start-bit scan sets.

Data from Another Scan Set

In exclusive operations on start-bit scan sets, the first position in a scan set receives the result of the operation for the *preceding* scan set, if there is one. Figure 73 is an example.

	0	1	2	3	4	5	6	7
start_bit	0	0	0	1	0	1	0	0
data	0	1	2	3	4	5	6	7
running_total	0	0	1	3	3	7	5	11

Figure 73. An exclusive operation in an upward direction with start bits.

Compare these results with those shown in Figure 68, which assumes that the sbit is a segment bit. **`[3]running_total`** and **`[5]running_total`** receive the results from the preceding scan set, rather than 0. **`[0]running_total`** still receives 0 (the identity for the operation) because there is no preceding scan set.

What constitutes a "preceding" scan set depends on the direction of the operation, of course. In a downward direction, scan sets with higher-numbered coordinates along the axis precede scan sets with lower-numbered coordinates.

13.3 The scan Function

Use the **`scan`** function to provide running results for operations on the scan sets you specify.

The definition of **`scan`** is:

```
type:current scan (
    type:current source,
```

```
    int axis,
    CMC_combiner_t combiner,
    CMC_communication_direction_t direction,
    CMC_segment_mode_t smode,
    bool:current *sbitp,
    CMC_scan_inclusion_t inclusion);
```

where:

source is the parallel variable whose values are to be used in the operation. It must be of the current shape, and it can have any arithmetic type.

axis specifies the axis along which the scan class or classes are to be created; see Section 13.2.

combiner specifies the type of operation that **scan** is to carry out. Possible values are listed in Section 13.1.

direction specifies the direction of the operation. Possible values are **CMC_upward** and **CMC_downward**.

smode specifies whether the sbit is a segment bit or a start bit; see Section 13.2.3. Possible values are **CMC_start_bit**, **CMC_segment_bit**, and **CMC_none**. Specify **CMC_none** if there is no sbit.

sbitp is a scalar pointer to a **bool**-size parallel variable of the current shape. This parallel variable is the sbit, which creates scan sets for the operation. Specify **CMC_no_field** if there is no sbit.

inclusion specifies whether the operation is exclusive or inclusive; see "Inclusive and Exclusive Operations," above. Possible values are **CMC_exclusive** and **CMC_inclusive**.

The function returns the result of the scan in a parallel variable of the current shape and with the same type as **source.**

The types **CMC_combiner_t**, **CMC_communication_direction_t**, **CMC_segment_mode_t**, and **CMC_scan_inclusion_t** are defined by the compiler.

The **scan** function provides a running result of the operation you specify on the parallel variable you specify. If you assign this result to a parallel variable of the current shape, each element of the parallel variable receives the running result for its position. The operation is carried out independently for each scan set.

13.3.1 Examples

The example below adds the values of **data** in an upward direction and assigns the running result to **running_total**; there is no sbit, and the operation is inclusive. The results are shown in Figure 74.

```
running_total = scan(data, 0, CMC_combiner_add,
    CMC_upward, CMC_none, CMC_no_field, CMC_inclusive);
```

```
running_total = scan(data, 0, CMC_combiner_add,
  CMC_upward, CMC_none, CMC_no_field, CMC_inclusive);
```

	0	1	2	3	4	5	6	7
data	4	7	9	5	3	5	9	6
running_total	4	11	20	25	28	33	42	48

Figure 74. An example of the scan function with no sbit.

The next example assigns the minimum value of **data** in the scan set to **running_min**. The direction is downward, the operation is inclusive, and the sbit is a start bit. The results are shown in Figure 75.

```
running_min = scan(data, 0, CMC_combiner_min,
    CMC_downward, CMC_start_bit, &start_bit,
    CMC_inclusive);
```

```
running_min = scan(data, 0, CMC_combiner_min,
  CMC_downward, CMC_start_bit, &start_bit,
  CMC_inclusive);
```

	0	1	2	3	4	5	6	7
start_bit	0	0	0	0	1	0	0	0
data	4	7	9	5	3	5	9	6
running_min	3	3	3	3	3	5	6	6

Figure 75. An example of the `scan` function with a start bit and a downward direction.

Note that you would get a different result in this example if the sbit were a segment bit, since segment bits and start bits behave differently when the direction is downward.

The example below multiplies the values of **`data`** in the scan set and assigns the product to **`running_product`**. The direction is upward, the operation is exclusive, and the sbit is a segment bit. The results are shown in Figure 76.

```
running_product = scan(data, 0, CMC_combiner_multiply,
    CMC_upward, CMC_segment_bit,
    &segment_bit, CMC_exclusive);
```

```
running_product = scan(data, 0,
  CMC_combiner_multiply, CMC_upward, CMC_segment_bit,
  &segment_bit, CMC_exclusive);
```

	0	1	2	3	4	5	6	7
segment_bit	0	0	0	0	1	0	0	0
data	4	7	9	5	3	5	9	6
running_product	1	4	28	252	1	3	15	135

Figure 76. An example of the `scan` function using a segment bit and an exclusive operation.

These examples are of a 1-dimensional shape, which by definition has only one scan class. If a shape has more than one dimension, more than one scan class is created, and **scan** carries out the operation on all scan subclasses (or scan sets, if the sbit is used) at the same time.

The destination parallel variable can be the same as the source parallel variable. In other words, a statement like this is legal:

```
data = scan(data, 0, CMC_combiner_add, CMC_upward,
    CMC_none, CMC_no_field, CMC_inclusive);
```

In this case, the elements of **data** are overwritten with the results of the operation.

13.4 The reduce and copy_reduce Functions

13.4.1 The reduce Function

Use the **reduce** function to put the result of an operation into a single parallel variable element in each scan subclass.

The **reduce** function has this definition:

```
void reduce (
    type:current *destp
    type:current source,
    int axis,
    CMC_combiner_t combiner,
    int to_coord);
```

where:

destp is a scalar pointer to a parallel variable, of the current shape and of any arithmetic type. One element of each scan subclass of this parallel variable receives the result of the operation.

source is a parallel variable (of the current shape) whose values are to be used in the operation. It must be of the same type as the parallel variable pointed to by **destp**.

axis specifies the axis along which the scan class or classes are to be created; see Section 13.2.

combiner specifies the type of operation that **reduce** is to carry out. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**.

to_coord specifies the coordinate of the parallel variable pointed to by **destp** that is to receive the result of the operation.

Note these differences between **reduce** and **scan**:

- **reduce** puts the final result of the operation into a single parallel variable element of the scan subclass; it does not produce a running result.
- **reduce** does not use scan sets; therefore, it does not have the arguments **smode** and **sbit**.
- Copying with reduction is handled as a separate function, which is discussed below.

Elements of **source** that are at inactive positions do not participate in the operation. If a position specified by **to_coord** is inactive, that element of **dest** does not receive the result.

dest can be the same parallel variable as **source**; the result simply overwrites the value(s) in the specified element(s).

An Example

The statement below puts the maximum value of **data** into element 0 of **max**. The results are shown in Figure 77.

```
reduce(&max, data, 0, CMC_combiner_max, 0);
```

```
reduce(&max, data, 0, CMC_combiner_max, 0);
```

	0	1	2	3	4	5	6	7
data	4	7	9	5	3	5	9	6
max	9							

Figure 77. An example of the `reduce` function.

Incidentally, this statement is virtually equivalent to this C* statement:

```
[0]max = >?= data;
```

But note these points:

- If position [0] were inactive, the assignment statement above would work; if you used **`reduce`**, the reduction would not take place.
- The equivalence holds only for 1-dimensional shapes. In shapes with more dimensions, **`reduce`** carries out its operation separately for each scan subclass, whereas the reduction assignment carries out its operation once for all elements of the parallel variable.

13.4.2 The copy_reduce Function

Use the **`copy_reduce`** function to copy a value from one parallel variable element of a scan subclass to another parallel variable element.

The definition of **`copy_reduce`** is:

```
void copy_reduce (
    type:current *destp
    type:current source,
    int axis,
    int to_coord,
    int from_coord);
```

The arguments are the same as for the **reduce** function, except that there is a **from_coord** argument instead of a combiner. **from_coord** specifies the element of **source** from which the value is to be copied. It is copied into the **to_coord** element of the parallel variable pointed to by **destp** for each scan subclass. If either **from_coord** or **to_coord** specifies an inactive position, the copying does not take place for that scan subclass.

An Example

This example copies the values of elements in row 1 of **data** into elements of row 0 of **copy**:

```
copy_reduce(&copy, data, 0, 0, 1);
```

The results for some sample values are shown in Figure 78.

```
copy_reduce(&copy, data, 0, 0, 1);
```

data

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23

copy

10	11	12	13

Figure 78. An example of the copy_reduce function.

If the example of **copy_reduce** shown in Figure 78 were applied to a 1-dimensional shape, it would be equivalent to:

```
[0]copy = [1]data;
```

If position [0] were inactive, however, the results would be different. **[0]copy** would get the result from **[1]data** if you used the assignment statement above; it would not get the value if you used **copy_reduce**.

13.5 The spread and copy_spread Functions

13.5.1 The spread Function

Use the **spread** function to place the result of an operation into all the elements of a specified parallel variable in a scan subclass.

The **spread** function has this definition:

```
type:current spread (
    type:current source,
    int axis,
    CMC_combiner_t combiner);
```

where:

source is a parallel variable (of the current shape) whose values are to be used in the operation. It can have any arithmetic type.

axis specifies the axis along which the scan class or classes are to be created; see Section 13.2.

combiner specifies the type of operation that **spread** is to carry out. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**. See Section 13.1.

spread returns its result in a parallel variable of the current shape; the parallel variable has the same type as **source**. This destination parallel variable can be the same as the source parallel variable, in which case the elements of the source parallel variable are overwritten with the result.

The **spread** function "spreads" the result of an operation into all active elements of the destination parallel variable in a scan subclass. Like **reduce**, **spread** does not use scan sets, and it does not have a **CMC_combiner_copy** operation; copying is handled by the **copy_spread** function, as discussed below.

Inactive positions do not participate in the operation.

An Example

The code below adds the values of the elements in **`data`** in the scan subclasses of axis 1, and assigns the result to **`total`**. The results for sample data are shown in Figure 79.

```
total = spread (data, 1, CMC_combiner_add);
```

```
total = spread (data, 1, CMC_combiner_add);
```

data

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23

total

6	6	6	6
46	46	46	46
86	86	86	86

Figure 79. An example of the `spread` function.

13.5.2 The copy_spread Function

Use the **`copy_spread`** function to copy a value from an element of a parallel variable in a scan subclass to all elements of a parallel variable in the scan subclass.

The **`copy_spread`** function has this definition:

```
type:current copy_spread (
    type:current *sourcep,
    int axis,
    int coordinate);
```

where:

`sourcep` is a scalar pointer to a parallel variable, one value of which is to be copied.

`axis` specifies the axis along which the scan class or classes are to be created.

`coordinate` is the coordinate along **`axis`** that specifies the source parallel variable element whose value is to be copied.

The function returns a parallel variable of the current shape and the same arithmetic type as the parallel variable pointed to by **`sourcep`**, containing the results of the operation.

If a specified element of the source parallel variable is inactive, its value is copied. However, inactive positions of the destination parallel variable do not receive a result.

An Example

The code below copies the value from element [*n*][1] of **`data`** to elements of **`copy`** in the same scan subclass along axis 1. The results are shown in Figure 80.

```
copy = copy_spread(&data, 1, 1);
```

`copy = copy_spread(&data, 1, 1);`

data

	0	**1**	**2**	**3**
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23

copy

1	1	1	1
11	11	11	11
21	21	21	21

Figure 80. An example of the `copy_spread` function.

Note that, for a 1-dimensional shape, the above statement is equivalent to this statement:

```
copy = [1]data;
```

unless position [1] is inactive. In that case, the assignment statement works; **`copy_spread`**, however, would not copy **`[1]data`**.

13.6 The enumerate Function

Use the **enumerate** function to place in each active element of a parallel variable the size of its scan set. As we discuss in more detail below, **enumerate** is a generalized version of the **pcoord** function.

The **enumerate** function has this definition:

```
unsigned int:current enumerate (
    int axis,
    CMC_communication_direction_t direction,
    CMC_scan_inclusion_t inclusion,
    CMC_segment_mode_t smode,
    bool:current *sbitp);
```

All the parameters for **enumerate** have the same meanings and take the same values as the corresponding parameters for the **scan** function; see Section 13.3. Like **scan**, **enumerate** lets you specify a direction, an sbit, and whether the operation is to be exclusive or inclusive. Note, however, that the return value is an **unsigned int** of the current shape.

If you specify **CMC_inclusive**, **enumerate** includes each position in calculating the size of the scan set for that position. If you specify **CMC_exclusive**, **enumerate** does not include the position in calculating the size of its scan set.

An inactive position does not receive a value and is not included in the calculation of values for other positions; see the third example, below.

13.6.1 Examples

The first example does an exclusive enumerate in an upward direction, ignoring the sbit, and assigning the result to **number**. The results are shown in Figure 81.

```
number = enumerate(0, CMC_upward,
    CMC_exclusive, CMC_none, CMC_no_field);
```

number = enumerate(0, CMC_upward, CMC_exclusive, CMC_none, CMC_no_field);

	0	1	2	3	4	5	6	7
number	0	1	2	3	4	5	6	7

Figure 81. An example of the `enumerate` function without an sbit.

This is exactly equivalent to this use of **`pcoord`** when all positions are active:

```
number = pcoord(0);
```

Both functions initialize each parallel variable element to its coordinate along the axis. The **`enumerate`** function, however, is more versatile than **`pcoord`**. In the next example, **`enumerate`** uses the sbit as a start bit and proceeds in a downward direction, using the inclusive mode:

```
number = enumerate(0, CMC_downward, CMC_inclusive,
    CMC_start_bit, &start_bit);
```

The results are shown in Figure 82.

number = enumerate(0, CMC_downward, CMC_inclusive, CMC_start_bit, &start_bit);

	0	1	2	3	4	5	6	7
start_bit	0	0	0	0	1	0	0	0
number	5	4	3	2	1	3	2	1

Figure 82. An example of the `enumerate` function with a start bit and a downward direction.

In the example below, the sbit is a segment bit, the enumerate is exclusive, the direction is upward, and position 2 is inactive. The results are shown in Figure 83.

```
where (p1 != 9)
    number = enumerate(0, CMC_upward, CMC_exclusive,
    CMC_segment_bit, &segment_bit);
```

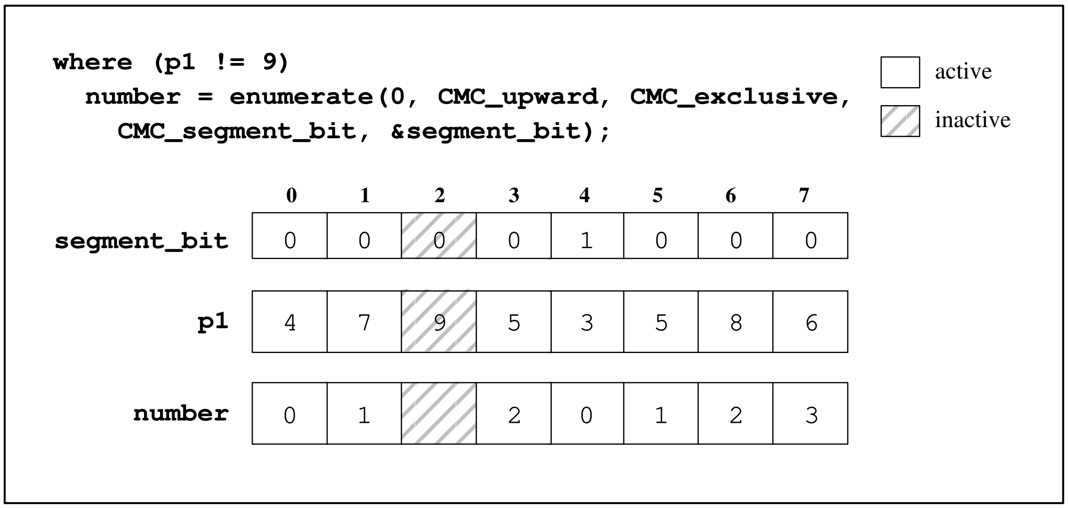

Figure 83. An example of the `enumerate` function using a segment bit and an exclusive operation, with an inactive position.

Note that the inactive position is not included in the enumeration.

13.7 The rank Function

Use the **`rank`** function to produce a numerical ranking of the values of parallel variable elements in a scan set.

The definition of **`rank`** is:

```
unsigned int:current rank (
    type:current source,
    int axis,
    CMC_communication_direction_t direction,
```

```
CMC_segment_mode_t smode,
bool:current *sbitp);
```

The parameters for **rank** have the same meanings and take the same values as the corresponding parameters for the **scan** function; see Section 13.3. Like **scan** and **enumerate**, **rank** lets you specify a direction and an sbit. It does not, however, let you specify that its operation is exclusive; the operation can only be inclusive. Also, note the behavior of **rank** with scan sets discussed below. Like the **enumerate** function, **rank** returns an **unsigned int** of the current shape.

The **rank** function returns, for each active position, the rank of the value of the specified parallel variable at that position in its scan set. Inactive positions are not included in the determination of the rank for other positions, and they do not receive a rank themselves. The ranking is from 0 to n–1, where n is the total number of positions in the scan set. The ranks are assigned as follows:

- When the direction is *upward*, the lowest value is assigned rank 0.
- When the direction is *downward*, the highest value is assigned rank 0.
- If more than one element has the same value, their ranks are assigned arbitrarily within the range of ranks they represent.
- An sbit restarts the ranking of values within the scan set; however, *it does not restart the values assigned to the ranks*. This behavior is different from that of other functions. For example, if a scan set extends from position [4] through position [15], the ranks assigned within this scan set are 4 through 15, not 0 through 11.

13.7.1 Examples

The first example has no sbit and ranks the values of **data** in a upward direction; it assigns the ranks to **data_rank**. The results are shown in Figure 84.

```
data_rank = rank(data, 0,
    CMC_upward, CMC_none, CMC_no_field);
```

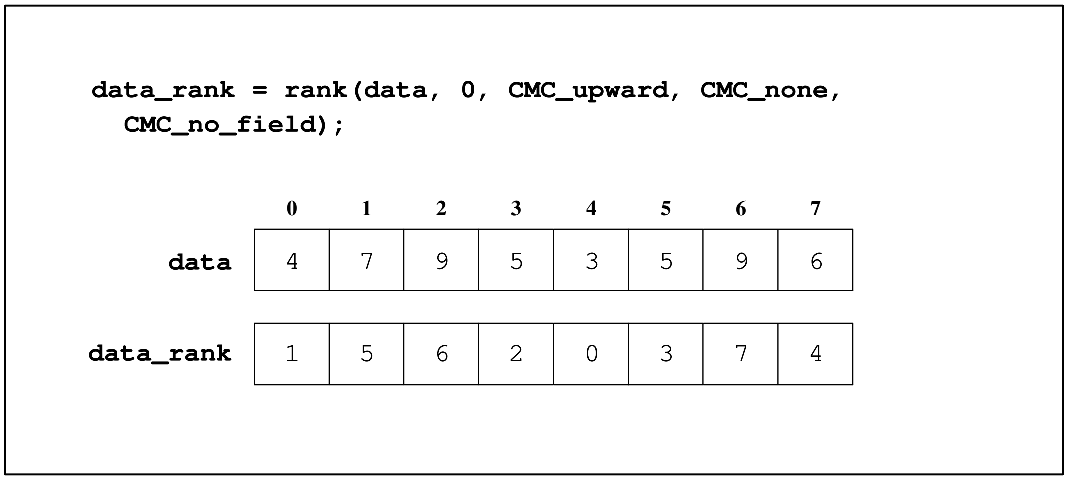

Figure 84. An example of the `rank` function with no sbit.

In the next example, the sbit is a segment bit, the direction is downward, and position 1 is inactive. The results are shown in Figure 85.

```
where (data != 7)
    data_rank = rank(data, 0, CMC_downward,
                        CMC_segment_bit, &segment_bit);
```

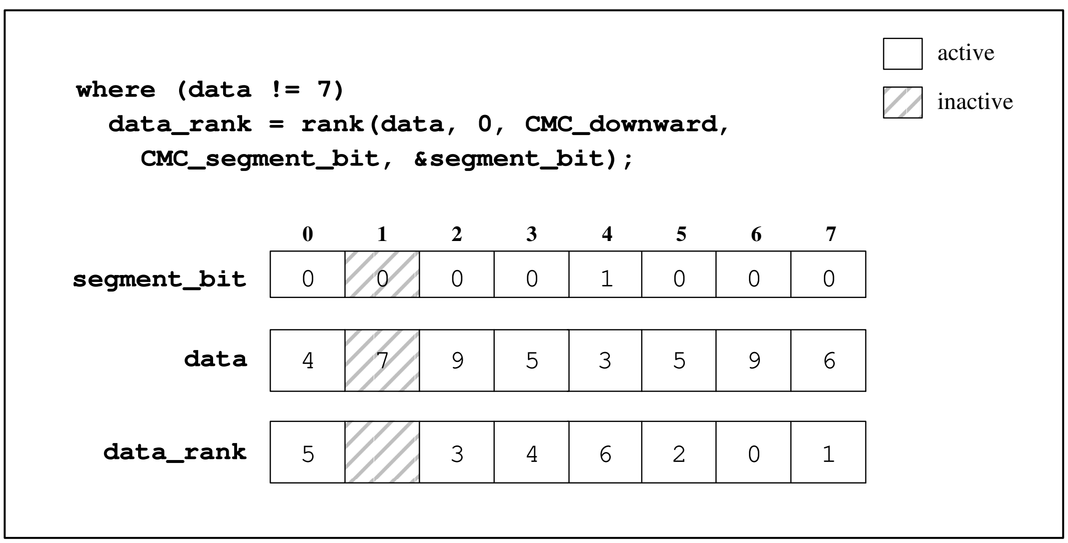

Figure 85. An example of the `rank` function using a segment bit and a downward direction, with an inactive position.

The final example uses **rank** along with parallel left indexing to actually reorder parallel variable elements according to their rank:

```
[rank(data, 0, CMC_upward, CMC_none,
CMC_no_field)]sorted = data;
```

In this example, **data** sends values to **sorted**, using the return values from **rank** as an index. The key here is to have **rank** operate on the parallel variable that is doing the sending. The results are shown in Figure 86.

```
[rank(data, 0, CMC_upward, CMC_none,
   CMC_no_field)]sorted = data;
```

	0	1	2	3	4	5	6	7
data	4	7	9	5	3	5	9	6
sorted	3	4	5	5	6	7	9	9

Figure 86. Using rank as a parallel left index to reorder parallel variable elements according to their ranks.

Note how values move in the example: **[0]data**, for example, has a rank of 1; therefore, its value (4) is sent to **[1]sorted**.

You can also achieve the same result using the **make_send_address** and **send** functions along with **rank**; see Section 14.3.3.

13.8 The multispread Function

The **multispread** function is like the **spread** function, except that you can use it to spread the result of an operation along more than one axis at the same time. This is useful in shapes that have more than two dimensions. For example, in a 3-dimensional shape, you can use **spread** to spread results along any one of the dimensions; **`multispread`** lets you spread results through entire planes of positions instead of along a single dimension.

To see how this works, consider the simple 8-position 2-by-2-by-2 shape shown in Figure 87.

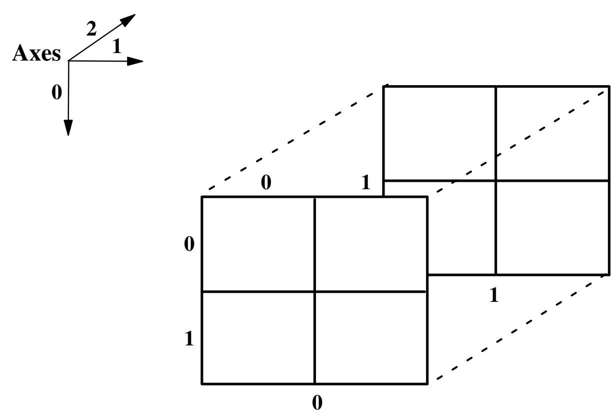

Figure 87. A 3-dimensional shape.

As we mentioned in Section 13.2.1, specifying axis 0 creates four scan classes for this shape:

[0][0][0] and [1][0][0]

[0][1][0] and [1][1][0]

[0][0][1] and [1][0][1]

[0][1][1] and [1][1][1]

In each scan class, the positions differ only along axis 0. These scan classes are shown in Figure 88.

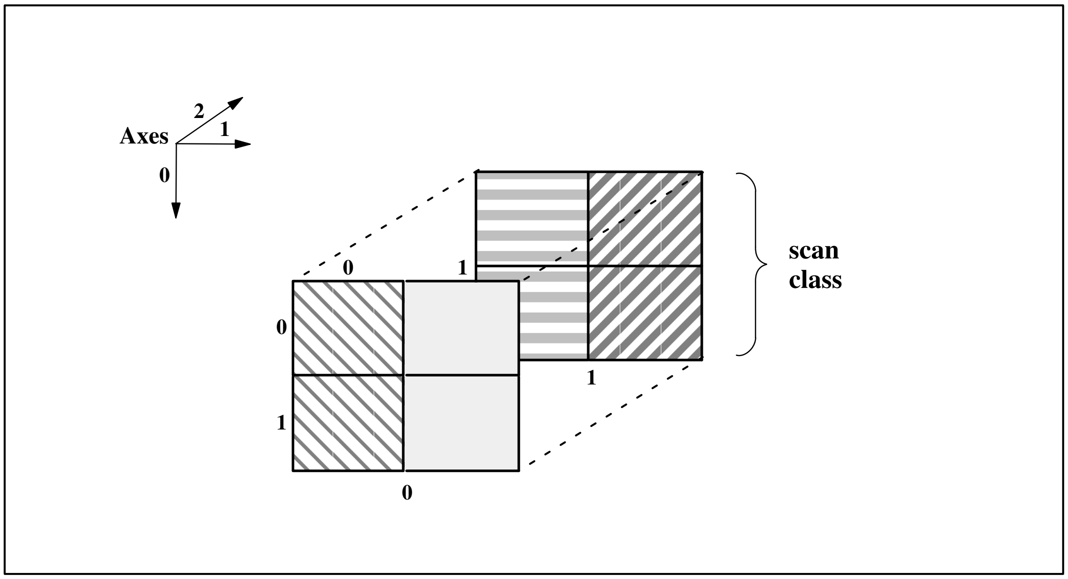

Figure 88. Scan classes in a 3-dimensional shape.

For the **`multispread`** function, you can specify more than one axis along which the positions can differ. In this case, let the positions differ along axes 0 and 1; axis 2 is fixed. This results in two sets of positions:

```
[0][0][0]
[1][0][0]
[0][1][0]
[1][1][0]
```

and:

```
[0][0][1]
[1][0][1]
[0][1][1]
[1][1][1]
```

Figure 89 shows these two sets of positions. The sets of positions in which the positions are allowed to differ along more than one axis are called *hyperplanes*. Scan classes are therefore a special case of hyperplanes, in which the positions can differ along only one axis. The **`multispread`** function operates on any kind of hyperplane.

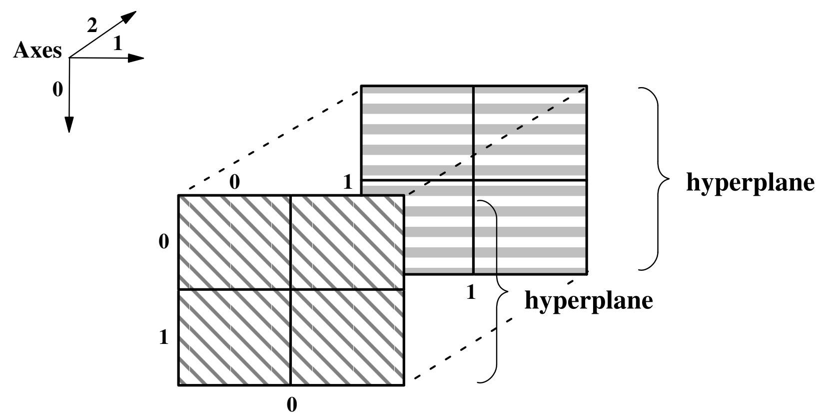

Figure 89. Hyperplanes in a 3-dimensional shape.

The **multispread** function has this definition:

```
type:current multispread (
    type:current source,
    unsigned int axis_mask,
    CMC_combiner_t combiner);
```

The only difference between this definition and that of **spread** is the **axis_mask** parameter. The **axis_mask** parameter is a bit mask that specifies the axes along which the positions in a hyperplane are allowed to differ. For example, use a bit mask of 3 to specify axes 0 and 1; use 6 to specify axes 1 and 2.

The example below assumes a 3-dimensional shape like the one shown above. In it, the values of **source** in the hyperplanes described by axes 0 and 1 are added, and the results are spread to all elements of **dest** in the same hyperplane.

```
dest = multispread(source, 3, CMC_combiner_add);
```

13.8.1 The copy_multispread Function

There is also a **copy_multispread** function, comparable to the **copy_spread** function, but available for use on hyperplanes instead of scan classes. Using **copy_multispread**, however, requires an understanding of send addresses, which are discussed in the next chapter. We therefore defer discussion of this function until Section 14.5.

13.9 The global Function

Use the **global** function to perform reduction operations on a parallel variable and assign the result to a scalar variable.

The **global** function has this definition:

```
type global (
    type:current source,
    CMC_combiner_t combiner);
```

where:

source is a parallel variable (of the current shape and any arithmetic type) upon whose values the reduction operation is to be performed.

combiner specifies the reduction operation. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**; see Section 13.1 for definitions of these values.

The function returns a scalar variable of the same type as **source**.

The **global** function provides an alternative method for performing certain reduction operations. For example, these two statements are equivalent (where **s1** is a scalar variable and **p1** is a parallel variable of the same type):

```
s1 = |= p1;
```

and:

```
s1 = global(p1, CMC_combiner_logior);
```

Both do a bitwise inclusive OR of **p1** and assign the result to **s1**.

Note that **`global`** does not have a **`combiner`** value for the reduction assignment operator **`-=`** (negative of the sum of the parallel values).

The **`global`** function operates only on active positions.