Chapter 12

Grid Communication

As we mentioned in the previous chapter, there are two ways for data to be communicated from one position to another within a shape: by using the absolute address (called the *send address*) of the position, or by using the position's coordinates within the shape. Within-shape communication in regular patterns that uses positions' coordinates is referred to as *grid communication*, since the coordinates can be thought of as locating positions on an *n*-dimensional grid.

This chapter describes C* library functions that provide grid communication. These functions are faster than the general communication functions described in Chapter 14. If you use any of the functions discussed in this chapter, include the file **<cscomm.h>** in your program. You can also achieve grid communication by using the **pcoord** function, as described in Chapter 10.

All grid communication functions are overloaded so that they can be used with any arithmetic or aggregate data type.

12.1 Aspects of Grid Communication

There are several aspects to grid communication to consider before using these functions:

- axis
- direction
- distance
- border behavior

Copyright © 1990–1995 Thinking Machines Corporation

- behavior of inactive positions

12.1.1 Axis

Grid communication functions let parallel variable elements communicate along any axis of a shape. In a 2-dimensional shape like Figure 50, for example, you can specify that elements communicate along axis 0 or along axis 1.

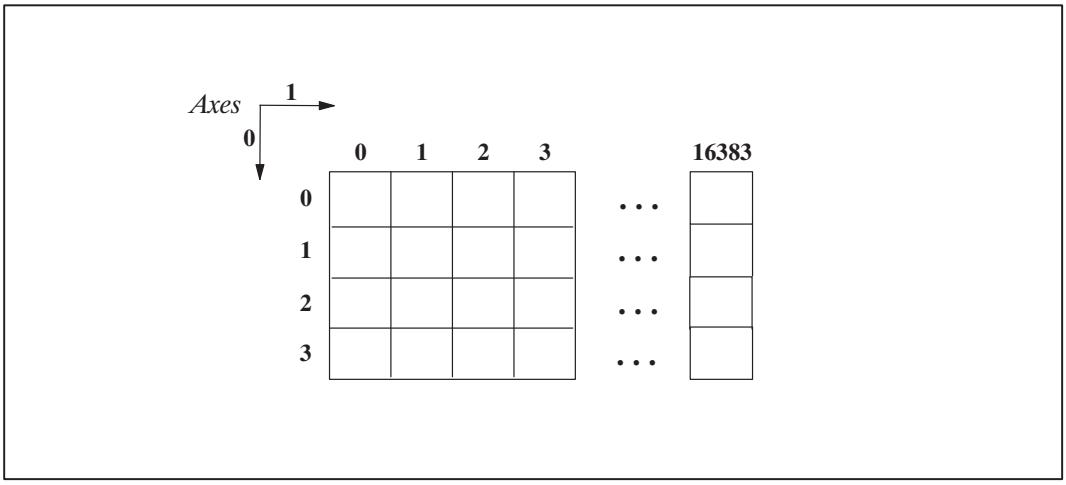

Figure 50. A 2-dimensional shape.

The functions **`from_grid`**, **`to_grid`**, **`from_torus`**, and **`to_torus`** allow communication along more than one axis — for example, an element could transmit a value to another element by sending it down axis 0, then across axis 1.

12.1.2 Direction

Parallel variable elements can also communicate in either direction along an axis using grid communication. In Figure 50, for example, parallel variable elements at position [0][2] can communicate along axis 1 with elements to the right (position [0][3]) or to the left (position [0][1]).

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

12.1.3 Distance

Parallel variables can communicate at any distance along an axis. For example, parallel variable elements at position [0][0] in Figure 50 can communicate with elements at position [0][16383].

12.1.4 Border Behavior

What happens when a parallel variable element at position [0][16383] in Figure 50 tries to get a value from the right — off the border of the grid? The behavior of grid communication at the border is handled in different ways by different functions. Specifically,

- In the functions **`from_grid`**, **`from_grid_dim`**, **`to_grid`**, and **`to_grid_dim`**, you can specify a value that the element is to receive when it tries to get a value from beyond the border. This value is referred to as the *fill value*.
- In the functions **`from_torus`**, **`from_torus_dim`**, **`to_torus`**, and **`to_torus_dim`**, the element receives the value from the opposite border of the grid — in this case, the element at position [0][16383] gets its value from position [0][0]. This is known as *wrapping*.

12.1.5 Behavior of Inactive Positions

What happens when positions in the grid are inactive? For example, a parallel variable element at position [0][0] tries to get the value of an element at position [0][1], but position [0][1] is inactive.

Different functions handle inactive positions in different ways, depending on whether parallel variables are seen as sending their values to other positions or getting values from other positions. The distinction is the same one made for parallel left indexing; see Section 10.1.6. Specifically,

- In a get operation, a parallel variable element in an active position can get a value from an element in an inactive position, but an element in an inactive position cannot get a value from any position. The functions **`from_grid`**, **`from_grid_dim`**, **`from_torus`**, and **`from_torus_dim`** use get operations.

Copyright © 1990–1995 Thinking Machines Corporation

- In a send operation, a parallel variable element in an active position can send a value to an element in an inactive position, but an element in an inactive position cannot send its value. The functions **to_grid**, **to_grid_dim**, **to_torus**, and **to_torus_dim** use send operations.

Note that the issue of getting from or sending to inactive positions requires passing some parallel variables in the grid communication functions by reference, rather than by value. See Chapter 10 for a discussion of this issue.

Table 3 summarizes the features of the grid communication functions.

Table 3. Features of grid communication functions.

Function	Multiple Axes?	Wrapping?	Get or Send?
from_grid	Yes	No	Get
from_grid_dim	No	No	Get
from_torus	Yes	Yes	Get
from_torus_dim	No	Yes	Get
to_grid	Yes	No	Send
to_grid_dim	No	No	Send
to_torus	Yes	Yes	Send
to_torus_dim	No	Yes	Send

12.2 The from_grid_dim Function

Use the **from_grid_dim** function to communicate along one axis of a grid, without wrapping. **from_grid_dim** is a get operation, as described in Chapter 10.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

12.2.1 With Arithmetic Types

Like all grid communication functions, **from_grid_dim** can be used with arithmetic data types, as well as with parallel structures and parallel arrays. The version of **from_grid_dim** for arithmetic data types has this definition:

```
type:current  from_grid_dim (
    type:current *sourcep,
    type:current value,
    int axis,
    int distance);
```

where

sourcep is a scalar pointer to the parallel variable from which values are to be obtained. The parallel variable can be of any arithmetic type; it must be of the current shape.

value is a parallel variable of the current shape whose values are to be used when elements try to get values from beyond the border of the grid. The parallel variable must be of the same arithmetic type as the parallel variable pointed to by **sourcep**.

axis specifies the axis along which the communication is to take place.

distance specifies how many positions along the axis the values are to travel. For example, if **distance** is 2, each parallel variable element gets a value from an element whose position is two greater along the specified axis. **distance** can be a negative number, which reverses the direction in which the data is to travel.

from_grid_dim returns the source values in their new positions. You can assign these values to a parallel variable of the current shape and of the same arithmetic type as the source parallel variable; this "destination" parallel variable can be viewed as the parallel variable that is doing the "getting."

Note the difference between **from_grid_dim** and the corresponding use of **pcoord** described in Chapter 10: **pcoord** does not provide a fill value when an element tries to get from beyond the border.

Copyright © 1990–1995 Thinking Machines Corporation

Examples

Figure 51 shows three parallel variables of the same shape.

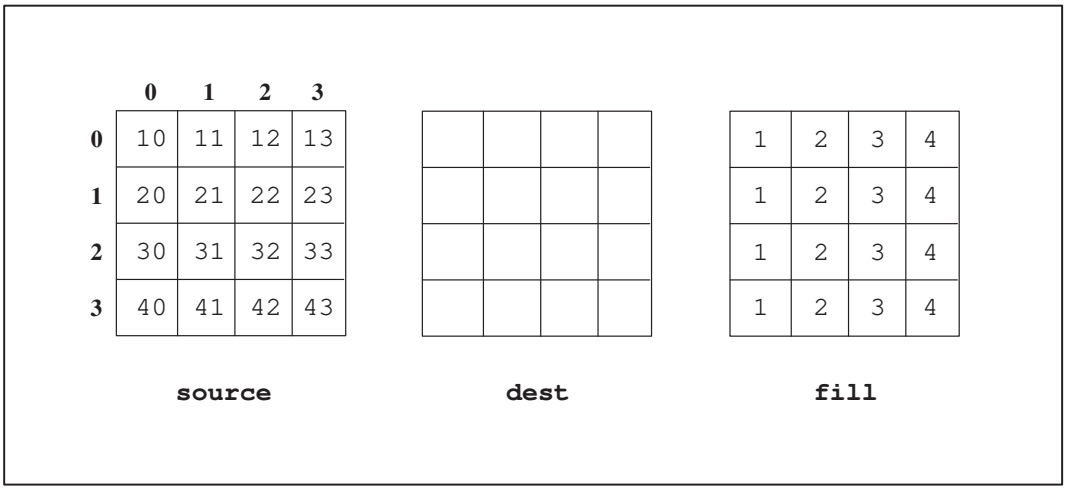

Figure 51. Three parallel variables of shape `ShapeA`.

The goal is for **`dest`** to get values of the parallel variable pointed to by **`sourcep`** that are one position lower along axis 0. This is equivalent to scalar C* statements like these (except that all operations happen at the same time):

```
[1][0]dest = [0][0]source;
[2][0]dest = [1][0]source;
[3][0]dest = [2][0]source;
[1][1]dest = [0][1]source;   /* . . . and so on */
```

In the case where **`dest`** tries to get a value of **`source`** from beyond the border (for example, the **`dest`** element at position [0][0]), it is to use the value from the corresponding element of **`fill`**.

The code below accomplishes this:

```
#include <cscomm.h>

shape [256][256]ShapeA;
int:ShapeA source, dest, fill;

/* Code to initialize parallel variables omitted. */
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
main()
{
    with (ShapeA)
         dest = from_grid_dim(&source, fill, 0, -1);
}
```

Figure 52 shows the results.

Note that we use −1 for the **distance** argument, even though the values move to higher-numbered positions along the axis. As mentioned above, **from_grid_dim** is a get operation; in this case, the element in the higher-numbered position is viewed as getting the data from the lower-numbered position, and that is why a negative distance is used.

Note also the values of **fill** that are used when **dest** attempts to get from beyond the border of the grid.

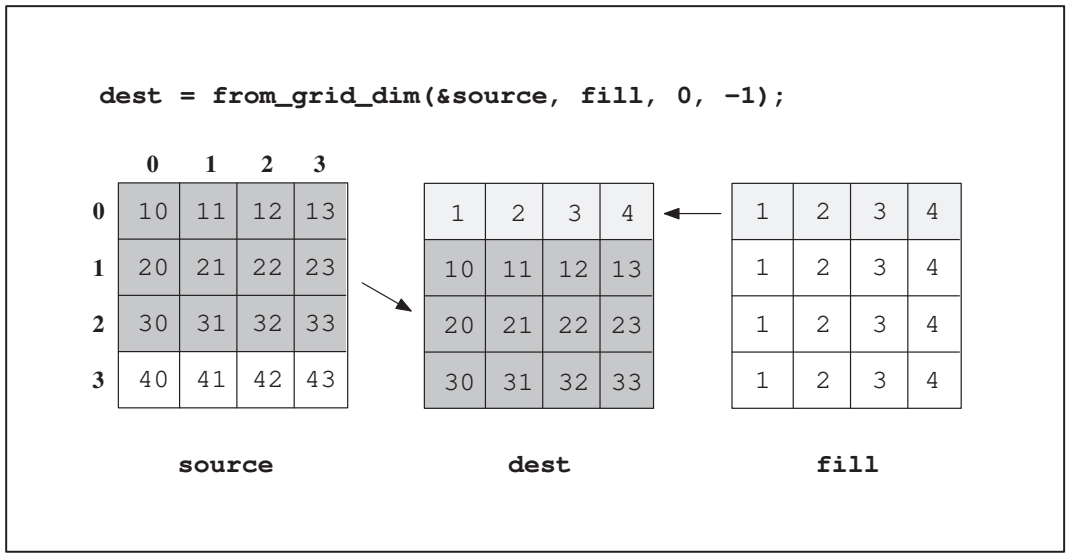

Figure 52. An example of the from_grid_dim function.

Now let's take the data in Figure 52 and move the values in **dest** two positions lower along axis 1, but leaving them in **dest**. In scalar C* code:

```
[0][0]dest = [0][2]dest;
[0][1]dest = [0][3]dest;
[1][0]dest = [1][2]dest;  /*  . . . and so on */
```


Copyright © 1990–1995 Thinking Machines Corporation

In this case, the source parallel variable is the same as the destination parallel variable. This is legal. This statement does the job:

```
dest = from_grid_dim(&dest, fill, 1, 2);
```

A positive integer is used for the distance, because the elements in the lower-numbered positions along the axis are getting data from the elements in the higher-numbered positions.

Figure 53 shows the results.

Note that the elements of **dest** at positions [*n*][2] and [*n*][3] (where *n* is any axis 0 coordinate) are assigned the values from the corresponding elements of **fill**, because they attempt to get values from beyond the border of the grid.

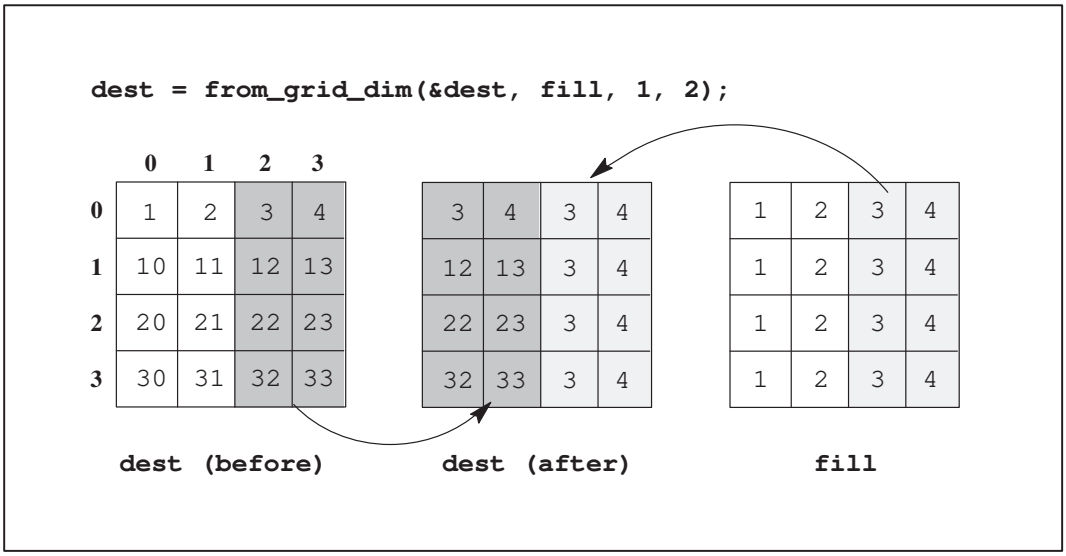

Figure 53. Another example of the `from_grid_dim` function.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

When Positions Are Inactive

Finally, let's see what happens when positions in a shape are inactive. The code fragment below makes position [2] inactive, using the simple data in Figure 54, and then calls **from_grid_dim**:

```
where (source != 7)
  dest = from_grid_dim(&source, fill, 0, -1);
```

Figure 54 shows the results.

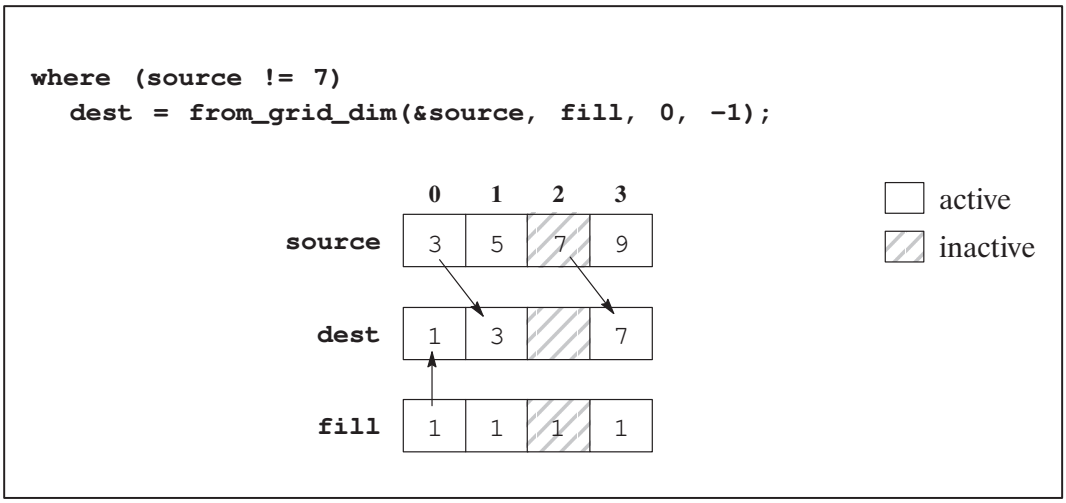

Figure 54. An example of from_grid_dim when a position is inactive.

Since **from_grid_dim** is a get operation, these rules apply:

- Elements at active positions can get values from elements at inactive positions.
- Elements at inactive positions cannot perform any gets at all.
- Elements at inactive border positions do not receive a fill value.

Note how these rules are applied in Figure 54:

- Position [2] is inactive, so it doesn't get a value from position [1]. (It keeps the value it had before the operation.)

Copyright © 1990–1995 Thinking Machines Corporation

- Position [3] gets a value from position [2], even though position [2] is inactive.

12.2.2 With Parallel Data of Any Length

The definition of **from_grid_dim** for parallel data of any length is as follows:

```
void from_grid_dim (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int axis,
    int distance);
```

In this version, the location pointed to by **destp** gets values from the location pointed to by **sourcep**, using the **axis** and **distance** arguments to determine the axis for the communication and how many positions along the axis the values are to travel. If **destp** tries to get from beyond the border of the grid, it gets values from the corresponding location pointed to by **valuep** instead. The locations pointed to by **destp**, **sourcep**, and **valuep** are all **length** **bool**s long.

You can use this version of **from_grid_dim** to transfer data that is larger than the standard data types — typically, this data would be in a parallel array or parallel structure. Note that there is no return value, and the destination is specified as the first argument to the function.

For example, in the code below, **dest_struct** gets the values of **source_struct** that are four coordinates higher along axis 0. When this takes **dest_struct** beyond the border of the grid, it gets the corresponding values of **value_struct**.

```
#include <cscomm.h>

shape [65536]ShapeA;
struct S {
    int a;
    int b;
};
struct S:ShapeA source_struct, dest_struct, value_struct;
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
main()
{
    with (ShapeA)
         from_grid_dim(&dest_struct, &source_struct,
              &value_struct,boolsizeof(source_struct), 0, 4);
}
```

12.3 The from_grid Function

The **from_grid** lets data travel along more than one axis of the grid. Like **from_grid_dim**, it is a get operation.

12.3.1 With Arithmetic Types

The definition of **from_grid** (for the version that takes arithmetic types) is

```
type:current from_grid (
    type:current *sourcep,
    type:current value,
    int distance_along_axis_0, ... );
```

where **sourcep**, **value**, and the return value are defined as they were for **from_grid_dim**.

The argument **distance_along_axis_0** specifies how many positions along this axis the data is to travel. As with **from_grid_dim**, the sign of the integer (positive or negative) indicates the direction of travel along the axis. The ellipsis (. . .) indicates a variable number of arguments. Each argument is an **int** that represents the distance along succeeding axes that the data is to travel. You must include as many arguments as there are axes in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0.

from_grid lets you combine movement along different axes. For example, in the previous section we used two calls to **from_grid_dim** so that each **dest** element got the value from the **source** element that was one position lower along axis 0 and two positions higher along axis 1. This call to **from_grid** accomplishes the same thing:

```
dest = from_grid(&source, fill, -1, 2);
```


Copyright © 1990–1995 Thinking Machines Corporation

The **-1** argument specifies the direction and distance of the communication along axis 0; the **2** argument specifies the direction and distance of the communication along axis 1. The movement along axis 1 takes place after the movement along axis 0. That is, the **dest** elements first get the **source** elements one position lower along axis 0; the **dest** elements that are two positions lower along axis 1 then gets these values from these other **dest** elements.

Note an important difference between the single **from_grid** call and the two **from_grid_dim** calls, however. With **from_grid**, the fill value is inserted only after *all* data movement is completed. No fill values are inserted when elements try to get from beyond the border in intermediate steps. This ensures that elements of the destination parallel variable receive fill values from corresponding elements of the fill parallel variable. But it yields a different result from consecutive **from_grid_dim** calls, where the fill value is inserted for each call.

Figure 55 shows the results of the **from_grid** call shown above on the data in Figure 51. Compare these results with those for the two **from_grid_dim** calls shown in Figure 53 (the arrow on the left shows that **[0][2]source** ends up at **[1][0]dest**).

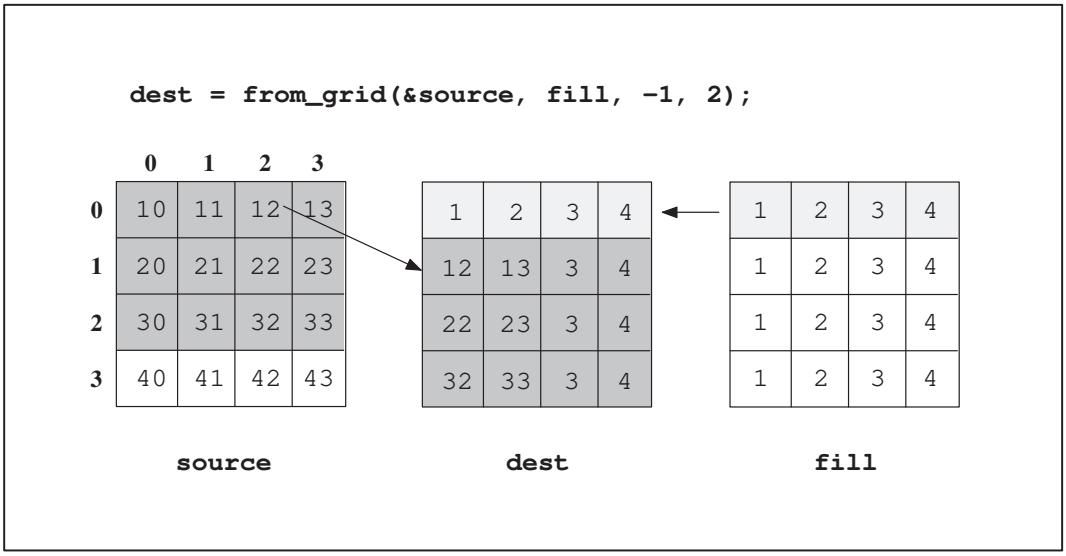

Figure 55. An example of the from_grid function.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

from_grid handles inactive positions in the same way that **from_grid_dim** does.

12.3.2 With Parallel Data of Any Length

Like **from_grid_dim**, **from_grid** has an overloaded version that can be used with parallel data of any length. Its definition is

```
void from_grid (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int distance_along_axis_0, ... );
```

Once again, **destp**, **sourcep**, and **valuep** are pointers to parallel locations that are **length** **bool**s long. Specify the data movement for each axis in the arguments **distance_along_axis_***n*. **destp** gets the value of **sourcep** based on these arguments, unless this brings it beyond the border of the grid, in which case it gets a value from the corresponding location pointed to by **valuep**.

12.4 The to_grid and to_grid_dim Functions

The **to_grid** and **to_grid_dim** functions are similar to **from_grid** and **from_grid_dim**, except that they are send operations instead of get operations. Both pairs of functions provide grid communication, with substitution of a fill value when the communication would otherwise go beyond the boundary of the grid. Both provide overloadings for arithmetic and aggregate types. The differences between the get operations and the send operations are

- in the way the distance argument is interpreted
- in the way inactive positions behave

These differences are described in more detail below.

Copyright © 1990–1995 Thinking Machines Corporation

12.4.1 With Arithmetic Types

The definitions of **to_grid** and **to_grid_dim** (for the versions that take arithmetic types) are

```
void to_grid (
    type:current *destp,
    type:current source,
    type:current *valuep,
    int distance_along_axis_0, ... );

void to_grid_dim (
    type:current *destp
    type:current source,
    type:current *valuep,
    int axis,
    int distance);
```

where

destp is a scalar pointer to the parallel variable to which values are to be sent. This parallel variable can be of any arithmetic type; it must be of the current shape.

source is the parallel variable that is to send its values. It can be of any arithmetic type; it must be of the current shape and of the same type as the parallel variable pointed to by **destp**.

valuep is a scalar pointer to a fill parallel variable whose values are to be used when elements of **source** try to send values to destinations beyond the border of the grid. It must be of the current shape and have the same type as **source**.

distance_along_axis_0
(for **to_grid**) specifies how many positions along axis 0 the values are to travel. For example, if **distance_along_axis_0** is 2, each parallel variable element of **source** sends a value to an element of the parallel variable pointed to by **destp** whose position is two greater along axis 0. Include a distance argument for each dimension in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0. The distance can be a negative num-

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

ber, which reverses the direction in which the data is to travel.

axis (for **to_grid_dim**) specifies the axis for the communication.

distance (for **to_grid_dim**) specifies how many positions along **axis** the values are to travel, as discussed in the description of **distance_along_axis_0**.

There is no return value.

Note the way that the **distance** argument is interpreted in send operations like **to_grid** and **to_grid_dim**. Specifying a positive integer for the distance sends values to higher-numbered positions. This is different from the behavior for get operations like **from_grid** and **from_grid_dim**, where specifying a positive integer for the distance gets values from higher-numbered positions.

When Positions Are Inactive

Since **to_grid** and **to_grid_dim** are send operations, these rules apply when positions are inactive:

- Elements at active positions can send values to elements at inactive positions.
- Elements at inactive positions cannot send their values.
- Elements at border positions receive fill values even if they are inactive. This follows the general behavior of send operations, in which elements at inactive positions can be sent values.

Examples

The first example uses **to_grid_dim** to achieve the same result as the use of **from_grid_dim** shown in Figure 52. The goal is for **source** to send values to elements of **dest** that are one position higher along axis 0. When the sending goes beyond the border of the grid, values of the corresponding elements of **fill** are used instead. This code accomplishes this:

```
to_grid_dim(&dest, source, &fill, 0, 1);
```


Copyright © 1990–1995 Thinking Machines Corporation

The results are shown in Figure 56.

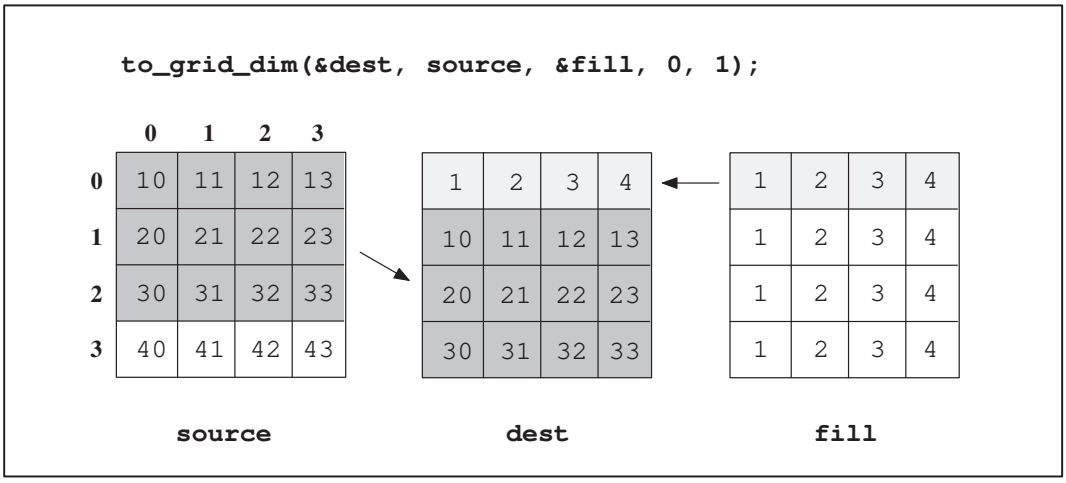

Figure 56. An example of the `to_grid_dim` function.

Similarly, to obtain the same results as those shown in Figure 53 for **`for_grid_dim`**, use this code:

```
to_grid_dim(&dest, dest, &fill, 1, -2);
```

These two calls to **`to_grid_dim`** are similar to this call to **`to_grid`**:

```
to_grid(&dest, source, &fill, 1, -2);
```

Note, however, that, as with **`from_grid`**, the fill values for **`to_grid`** are inserted only after *all* data movement has occurred. In this case, this produces a slightly different result for the single **`to_grid`** call; see Figure 55.

In all cases, note that the difference from the corresponding **`from_grid`** or **`from_grid_dim`** call is that the sign of each distance argument is reversed.

The final example makes positions [0] and [2] inactive and then calls **`to_grid_dim`**:

```
where (source != 7)
  to_grid_dim(&dest, source, &fill, 0, 1);
```

Figure 57 shows the results.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

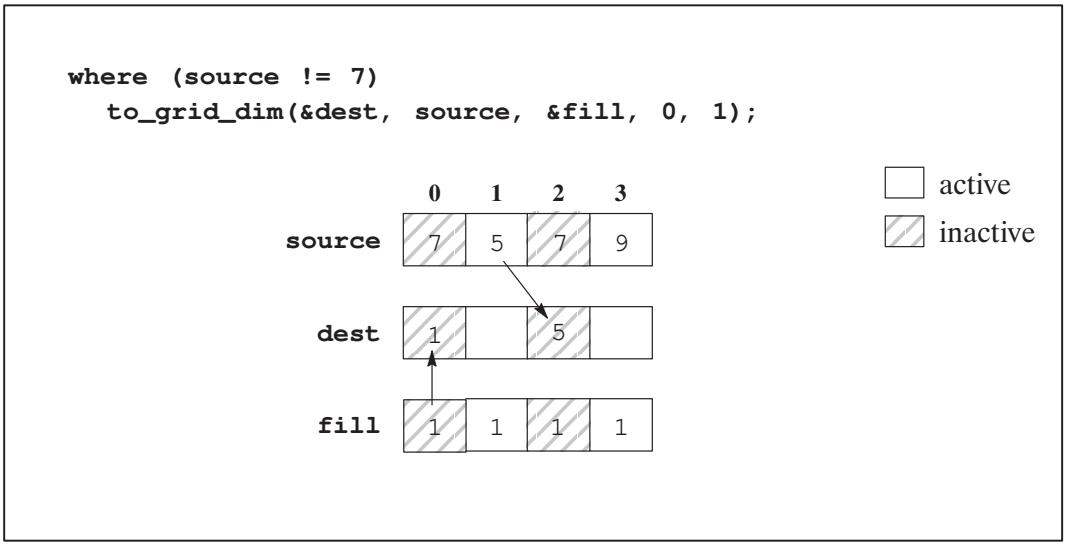

Figure 57. An example of `to_grid_dim` when position are inactive.

Note how the rules for inactive positions and send operations are applied in Figure 57:

- **`[0]source`** and **`[2]source`** are at inactive positions, so they don't send their values to **`[1]dest`** and **`[3]dest`**.
- **`[1]source`** sends its value to **`[2]dest`**, even though position [2] is inactive.
- **`[0]fill`** sends its value to **`[0]dest`**, even though position [0] is inactive.

12.4.2 With Parallel Data of Any Length

The definitions of **`to_grid`** and **`to_grid_dim`** for parallel data of any length are

```
void to_grid (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int distance_along_axis_0, ... );
```


Copyright © 1990–1995 Thinking Machines Corporation

```
void to_grid_dim (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int axis,
    int distance);
```

These versions are useful if you want to transfer data in a parallel array or parallel structure. As with the corresponding versions of **from_grid** and **from_grid_dim**, the **length** argument specifies the length in **bool**s of the locations pointed to by **destp**, **sourcep**, and **valuep**. There is no return value, and the destination is specified as the first argument to the function.

12.5 The from_torus and from_torus_dim Functions

A *torus* is a doughnut-shaped surface. The C* "torus" functions (two more are discussed in the next section) use the grid as if it were wrapped into a torus, with the opposite borders of the grid connected. If a value is required from beyond the border, it comes from the other side of the grid. Thus, these functions don't need the fill value used in the "grid" functions, since there is never a case where an element will not be able to obtain a value because it is beyond the border.

Except for this difference, **from_torus** and **from_torus_dim** are equivalent to **from_grid** and **from_grid_dim**. As with the other grid functions, there are overloaded versions for use with all arithmetic and aggregate types.

12.5.1 With Arithmetic Types

The definitions of **from_torus** and **from_torus_dim** (for the versions that take arithmetic types) are

```
type:current from_torus (
    type:current *sourcep,
    int distance_along_axis_0, ... );

type:current from_torus_dim (
    type:current *sourcep,
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
        int axis,
        int distance);
```

Let's look at how the results change when we use these functions on data from previous sections.

For example, let's take the data from Figure 51 and use **from_torus_dim** instead of **from_grid_dim**. The goal is the same: **dest** elements are to get the values of **source** elements that are one position lower along axis 0:

```
dest = from_torus_dim(&source, 0, -1);
```

Note that **from_torus_dim** does not require a **valuep** argument, since values wrap from the other side of the grid. The results of this statement are shown in Figure 58. The arrows in the figure show the movement for two elements of **source**: **[0][0]dest** wraps around to get the value of **[3][0]source**, and **[2][3]dest** gets the value of **[1][3]source**.

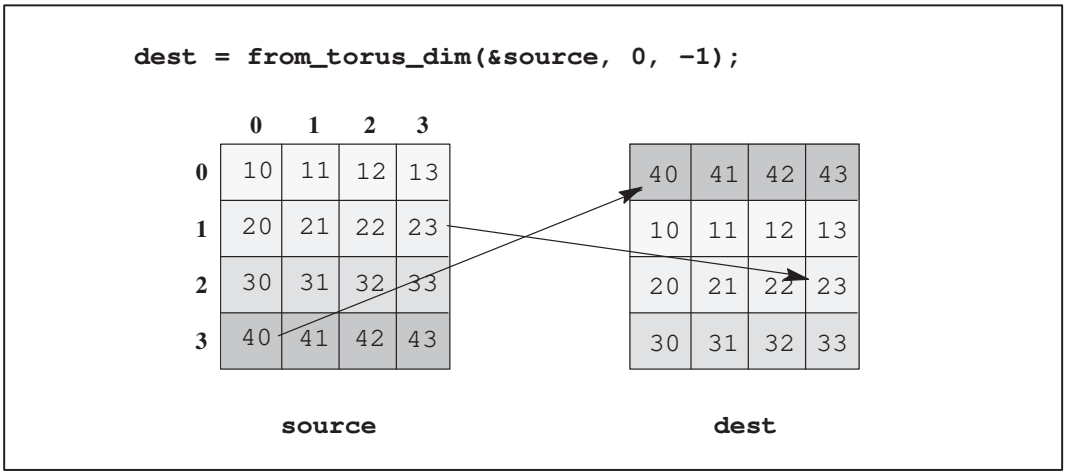

Figure 58. An example of the from_torus_dim function.

Compare the results shown in Figure 58 with those for the equivalent **from_grid_dim** call, shown in Figure 52. The differences are only in the **dest** elements that are at position [0][*n*]. **from_grid_dim** puts the value of the corresponding element of **fill** into the **dest** element. **from_torus_dim** wraps around to the other side of the grid and has the **dest** elements get the values of the **source** elements at position [3][*n*].

Copyright © 1990–1995 Thinking Machines Corporation

Similarly, using the same **source** data, this **from_torus** call:

```
dest = from_torus(&source, -1, 2);
```

produces the results shown in Figure 59. Compare these results with those shown in Figure 53, which are the results for the two **from_grid_dim** calls. Once again, **dest** elements that previously were assigned values of **fill** now get values of **source** elements from the other side of the grid. In Figure 59, the arrows show where the value of **[0][3]source** ends up: After the movement along axis 0, **[1][3]dest** gets it, and after the movement along axis 1, it ends up wrapping around to **[1][1]dest**.

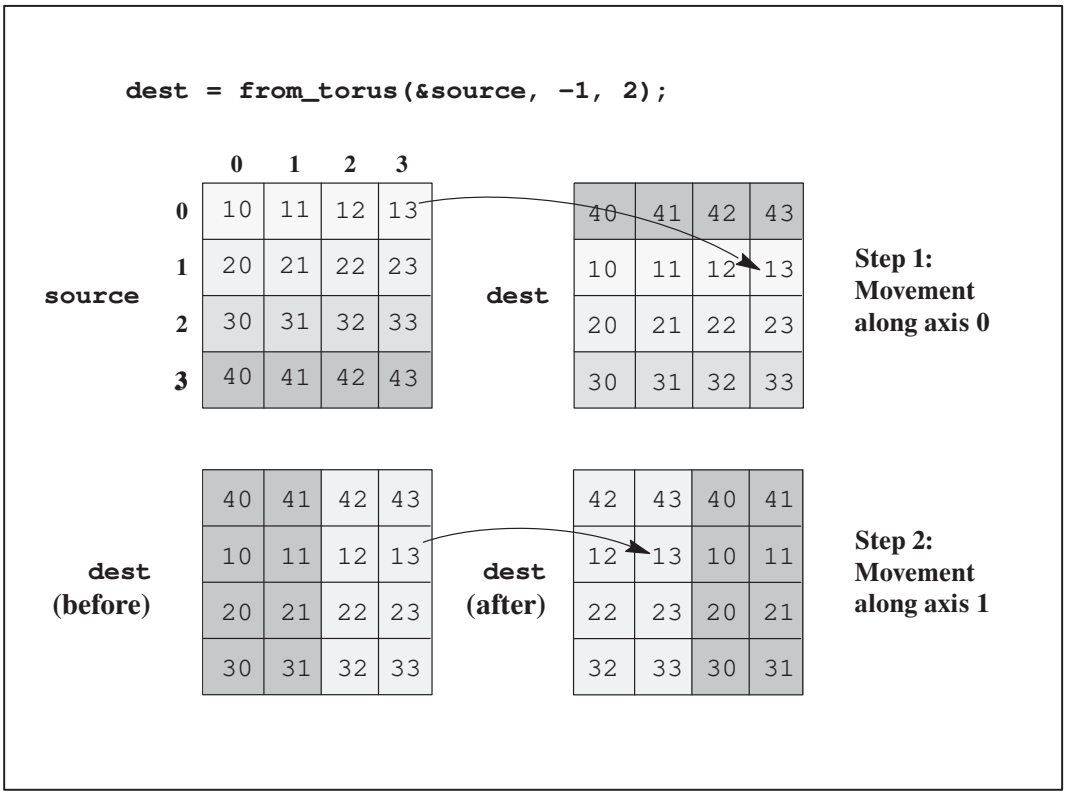

Figure 59. An example of the from_torus function.

from_torus and **from_torus_dim** are both get operations, so their handling of inactive positions is the same as that of **from_grid** and **from_grid_dim**.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

12.5.2 With Parallel Data of Any Length

The **from_torus** and **from_torus_dim** functions also have overloaded versions that can be used with parallel data of any length. Their definitions are

```
void from_torus(
    void:current *destp,
    void:current *sourcep,
    int length,
    int distance_along_axis_0, ... );

void from_torus_dim (
    void:current *destp,
    void:current *sourcep,
    int length,
    int axis,
    int distance);
```

Note that these definitions are the same as those for **from_grid** and **from_grid_dim**, except that a **valuep** argument is not required, since values wrap when they go beyond the border of the grid.

12.6 The to_torus and to_torus_dim Functions

The **to_torus** and **to_torus_dim** functions are send operations that provide grid communication with wrapping to the other side of the grid. As with the other grid communication functions, the **_dim** version provides communication along one axis only, while the more general version provides communication along all axes. Both functions have overloaded versions for all arithmetic and aggregate types.

12.6.1 With Arithmetic Types

The **to_torus** and **to_torus_dim** functions have these definitions when used with an arithmetic type:

```
void to_torus (
    type:current *destp,
```


Copyright © 1990–1995 Thinking Machines Corporation

```
    type:current source,
    int distance_along_axis_0, ... );

void to_torus_dim (
    type:current *destp,
    type:current source,
    int axis,
    int distance);
```

where

destp is a scalar pointer to the parallel variable to which values are to be sent. This parallel variable can be of any arithmetic type; it must be of the current shape.

source is a parallel variable from which values are to be sent; it must be of the current shape and have the same arithmetic type as the parallel variable pointed to by **destp**.

distance_along_axis_0
(for **to_torus**) specifies how many positions along axis 0 the values of **source** are to travel. If the distance is 2, for example, **source** sends its value to the destination element whose position is two greater along axis 0. Include a distance argument for each dimension in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0. The distance can be a negative number, which reverses the direction in which the data is to travel.

axis (for **to_torus_dim**) specifies the number of the axis along which the values of **source** are to be sent.

distance (for **to_torus_dim**) specifies how many positions along the axis the values of **source** are to be sent, as discussed in the description of **distance_along_axis_0**.

The behavior of inactive positions for **to_torus** and **to_torus_dim** is the same as it is for **to_grid** and **to_grid_dim**: Elements of **source** at inactive positions cannot send values, but **source** can send values to elements at inactive positions.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

Examples

The code below uses the **source** data also used in previous figures; it sends values of **source** to **dest** elements that are one position lower along axis 0:

```
to_torus_dim(&dest, source, 0, -1);
```

The results are shown in Figure 60. Compare these results to those for the comparable call to **from_torus_dim**, shown in Figure 58. The arrows in the figure show the movement of two elements of **source**: **[0][3]source** wraps around and sends its value to **[3][3]dest**; **[3][0]source** sends its value to **[2][0]dest**.

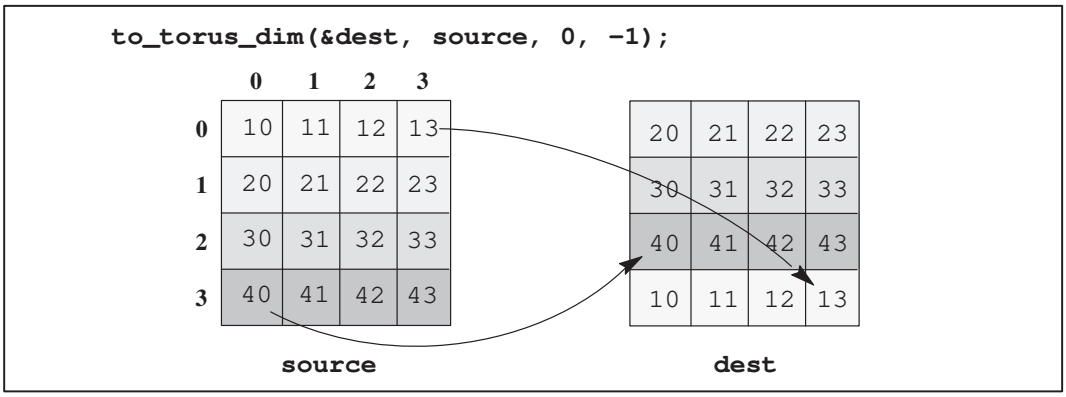

Figure 60. An example of the to_torus_dim function.

to_torus is similar to **to_torus_dim**, except that you must specify the data movement for each axis, as you do for **from_torus** and **from_grid**. This code uses the same **source** data used in previous figures:

```
to_torus(&dest, source, -1, 2);
```

The results are shown in Figure 61. Compare these results to those for the comparable call to **from_torus**, shown in Figure 59. The arrows in the figure show where **[0][3]source** ends up after the movement along axis 0 and axis 1.

Copyright © 1990–1995 Thinking Machines Corporation

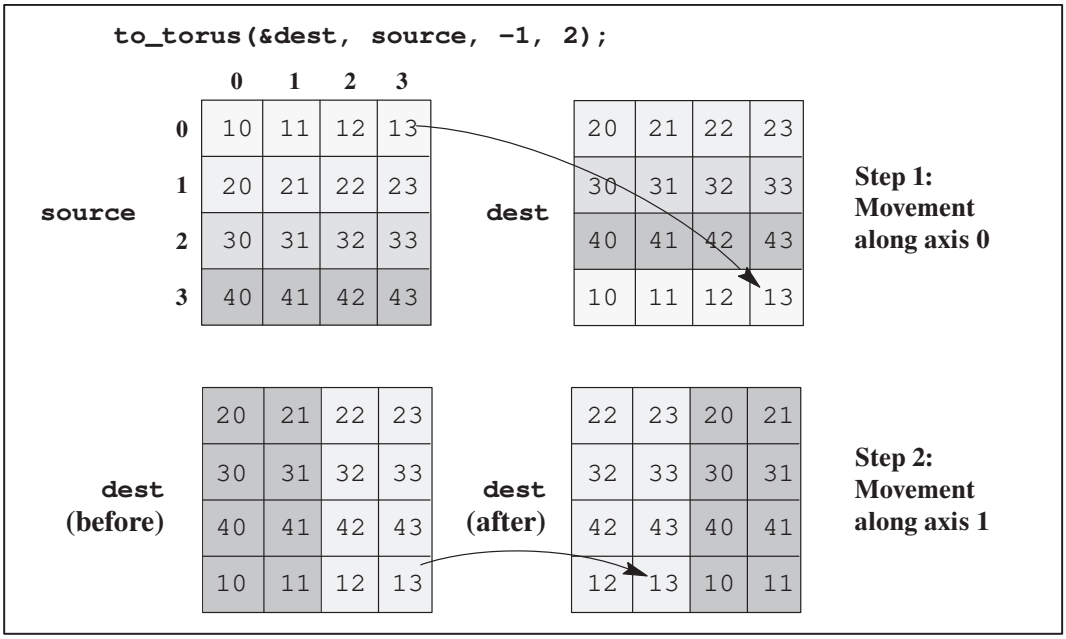

Figure 61. An example of the `to_torus` function.

In this example, we make a position inactive and call **`to_torus_dim`**:

```
where (source != 7)
  to_torus_dim(&dest, source, 0, 1);
```

Figure 62 shows the results for some sample data.

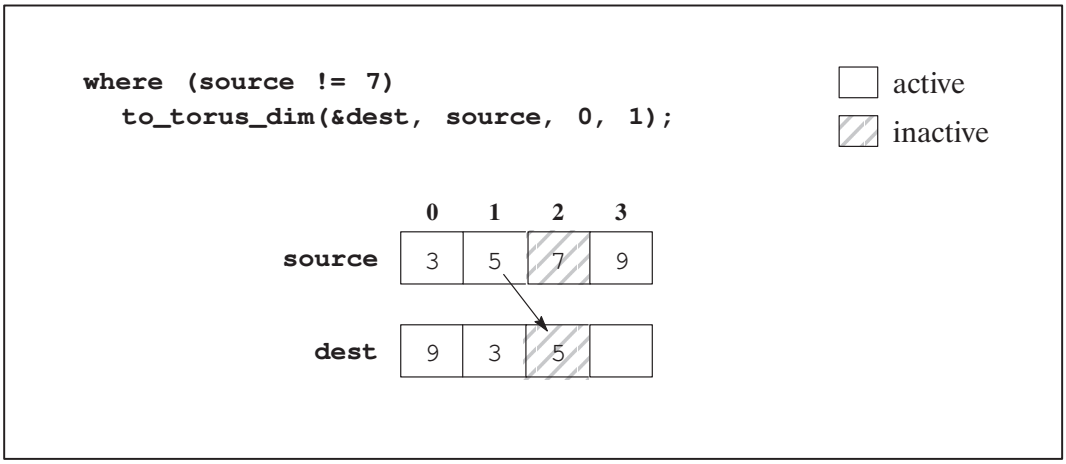

Figure 62. An example of `to_torus_dim` when a position is inactive.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

Note how the rules for send operations with inactive positions are applied in Figure 62:

- **[1]source** sends a value to **[2]dest**, even though position [2] is inactive.
- Position [2] is inactive, so **[2]source** doesn't send a value to **[3]dest**, which keeps its original value from before the call.

12.6.2 With Parallel Data of Any Length

The **to_torus** and **to_torus_dim** functions also have overloaded versions that can be used with parallel arrays or parallel structures. Their definitions are

```
void to_torus(
    void:current *destp,
    void:current *sourcep,
    int length,
    int distance_along_axis_0, ... );

void to_torus_dim (
    void:current *destp,
    void:current *sourcep,
    int length,
    int axis,
    int distance);
```

Note that these definitions are the same as those for **from_torus** and **from_torus_dim**. But, as with the versions that use arithmetic types, the distance arguments are interpreted differently, and the behavior of inactive positions is different.

Copyright © 1990–1995 Thinking Machines Corporation