Chapter 10

Communication

This chapter describes methods you can use to perform communication among parallel data. For example:

- Sending values of parallel variable elements to other elements of the same or a different shape.
- Getting values of parallel variable elements that are of the same or a different shape.

C* provides two methods of communication:

- *General communication*, in which the value of any element of a parallel variable can be sent to any other element, whether or not the parallel variables are of the same shape. You can use *parallel left indexing* to perform general communication. Parallel left indexing is described in Section 10.1.
- *Grid communication,* in which parallel variables of the same shape can communicate in regular patterns by using their coordinates. We use the term "grid communication" since the coordinates can be thought of as locating positions on an *n*-dimensional grid. Grid communication is faster than general communication. You can use the **`pcoord`** function, combined with parallel left indexing, to perform grid communication. The **`pcoord`** function is described in Sections 10.2 and 10.3.

In addition to the methods described in this chapter, C* includes a library of functions that provide an alternative way of performing grid and general communication; these functions are discussed in Part III of this manual. There are some differences in what you can accomplish using the different methods, but for most purposes the choice between the methods depends on individual preference.

10.1 Using a Parallel Left Index for a Parallel Variable

By now you should be familiar with the left indexing of a parallel variable to specify an individual element. For example, **`[0]p1`** specifies the first element of the 1-dimensional parallel variable **`p1`**. Similarly, if **`s1`** and **`s2`** are scalar variables, their values determine which element is specified by the 2-dimensional parallel variable **`[s1][s2]d1`**. But we have not yet covered the case in which a parallel variable is used as a left index for another parallel variable. If **`p0`** and **`p1`** are both 1-dimensional parallel variables, what does **`[p0]p1`** mean? If **`d0`**, **`d1`**, and **`d2`** are all 2-dimensional parallel variables, what does **`[d0][d1]d2`** mean?

Basically, a parallel left index rearranges the elements of the parallel variable, based on the values stored in the elements of the index; the index must be of the current shape. The example discussed below will help show how this works.

Note to users of CM-200 C*: This and other examples in this chapter do not represent valid shapes in the CM-200 implementation, because there are too few positions; we use these small shapes to make it easier to visualize what happens when you use a parallel left index.

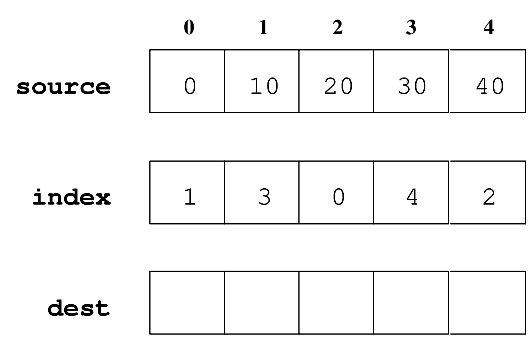

Figure 32. Three parallel variables.

10.1.1 A Get Operation

Given the situation shown in Figure 32, what is the result of the statement below?

```
dest = [index]source;
```

Let's look first at what goes into element 0 of **dest**. The value in element [0] of **index** is 1. This value is used as an index into the elements of **source**. The value in element 1 of **source** is 10. Therefore, element 0 of **dest** gets assigned the value 10. The way to think of this is that the LHS variable gets a value of the RHS variable, based on the value of the corresponding element of the index variable; we refer to this as a *get operation*. In C* code, what happens is this:

```
[0]dest = [1]source;
```

For element 1 of **dest**, the value of the **index** variable is 3. Therefore, element 1 of **dest** gets the value of element 3 of **source**, which is 30. In C* code:

```
[1]dest = [3]source;
```

And for the remaining elements:

```
[2]dest = [0]source;
[3]dest = [4]source;
[4]dest = [2]source;
```

It's important to note the difference between parallel left indexing and these serial statements. Parallel left indexing causes these assignments to occur *at the same time,* in parallel. In the serial statements, the result of an earlier statement could affect the result of a later one; this does not happen when all the statements are executed at the same time.

Figure 33 shows the results of the assignment statement for all elements of **dest**; the arrows show the process by which a value is assigned to **[0]dest**. The value of **[0]index** is 1, which causes **[0]dest** to get the value in **[1]source**.

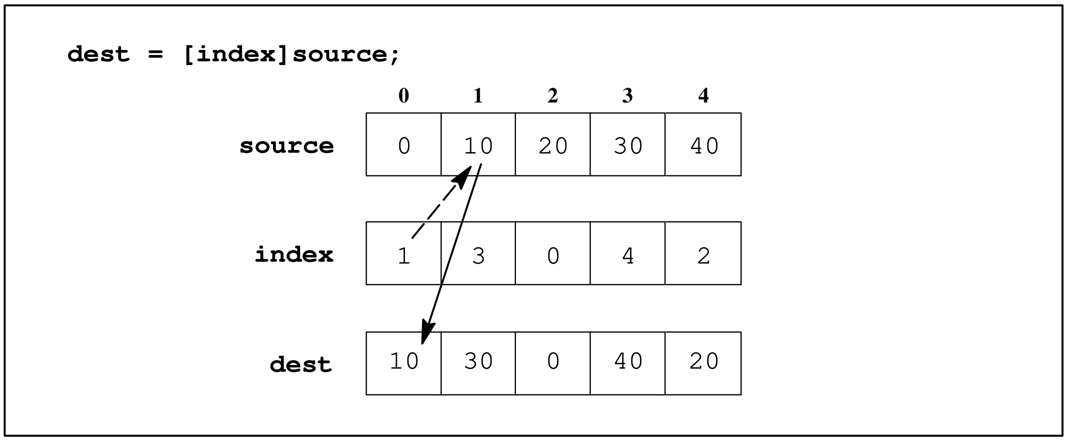

Figure 33. Parallel left indexing of a parallel variable — a get operation.

10.1.2 A Send Operation

Here is another assignment statement that uses the data in Figure 32:

```
[index]dest = source;
```

In this case, **`index`** is being used as an index for **`dest`**. In statements of this form, the RHS variable *sends* a value to the LHS variable, based on the value of the corresponding element of the index variable; we refer to this as a *send operation*.

Let's look at element 0 of **`source`**. The value in element 0 of the index variable **`index`** is 1; this value is used as an index into **`dest`**. The value in element 0 of **`source`**, 0, is sent to element 1 of **`dest`**. In C* code:

```
[1]dest = [0]source;
```

For element 1 of **`source`**, in the corresponding element, the value of **`index`** is 3; therefore, the value in element 1 of **`source`**, 10, is sent to element 3 of **`dest`**. In C* code:

```
[3]dest = [1]source;
```

The serial C* statements for the rest of the elements are:

```
[0]dest = [2]source;
[4]dest = [3]source;
[2]dest = [4]source;
```

Note once again, however, that parallel left indexing causes all these statements to be executed at the same time. The results are shown in Figure 34; the arrows show the process by which the value in **`[0]source`** is assigned to an element of **`dest`**. The value in **`[0]index`** is 1; therefore, **`[0]source`** sends its value to **`[1]dest`**.

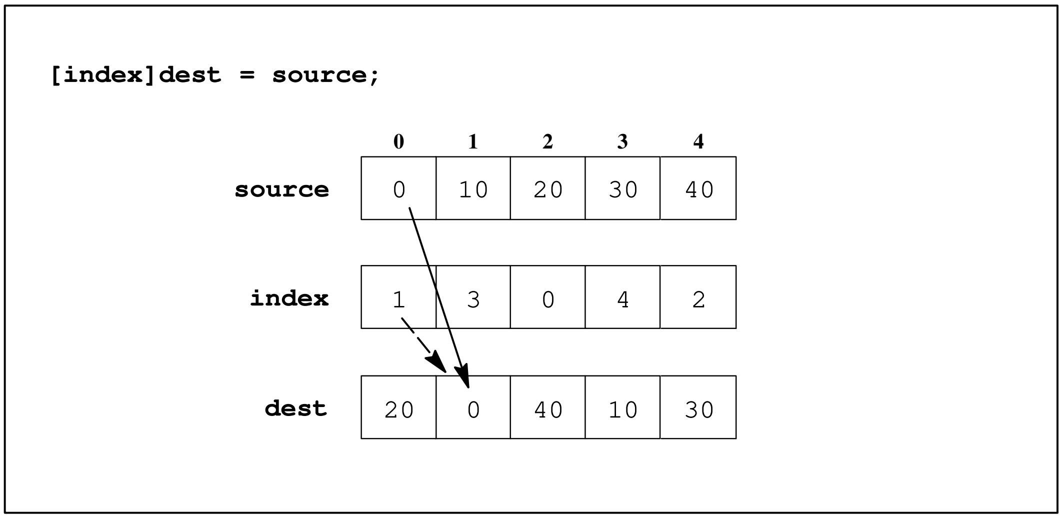

Figure 34. Parallel left indexing of a parallel variable — a send operation.

10.1.3 Use of the Index Variable

The index variable would typically contain values that cause a meaningful rearrangement of the parallel variable it indexes. For example, if we use the values shown in Figure 35,

```
dest = [index]source;
```

causes **`dest`** to contain the **`source`** values in reverse order; the arrows show the process by which **`[0]dest`** gets its value, based on the index in **`index`**.

The index variable cannot reference nonexistent elements of a parallel variable. For example, an index value of 5 in Figure 35 creates unpredictable results.

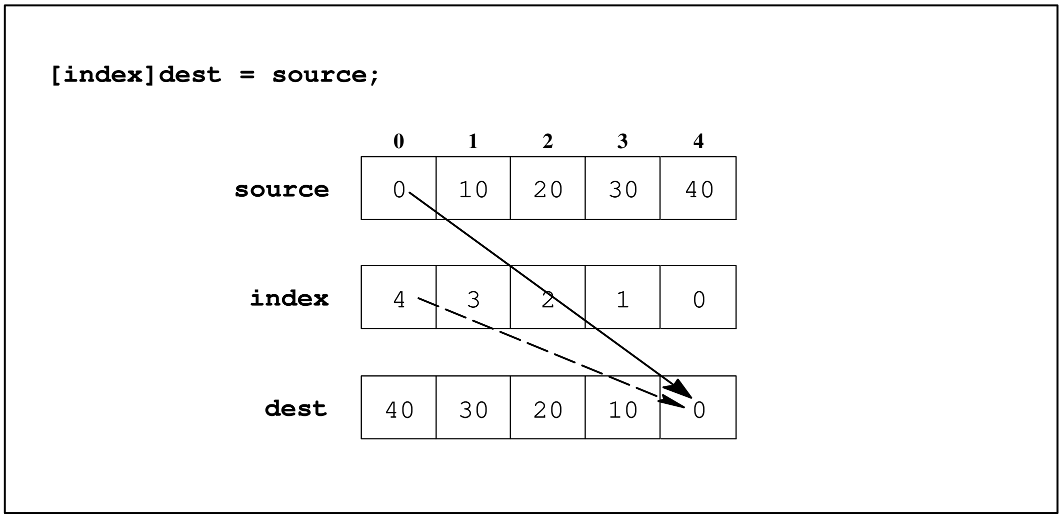

Figure 35. An index that reverses the order of a parallel variable.

10.1.4 If the Shape Has More Than One Dimension

Parallel left indexing can be used if the parallel variable is of a shape with more than one dimension. In this case, however, you need to specify a left index for each axis of the shape. For example:

```
shape [128][512]ShapeA;
int:ShapeA dest, index0, index1, source;

main()
{
    with (ShapeA)
         dest = [index0][index1]source;
}
```

In this case, **source** is of the 2-dimensional shape **ShapeA**. Therefore, it requires two left indexes to specify the values to be assigned to **dest**. **index0** is used as the index for axis 0 of **source**, and **index1** is used as the index for axis 1 of **source**.

If one of the indexes is parallel and one or more are scalar, the scalar indexes are promoted to parallel in the current shape.

10.1.5 When There Are Potential Collisions

In the examples of parallel left indexing shown so far, the index variable, **`index`**, has had different variables in each element. Let's consider a situation, shown in Figure 36, where this is not true.

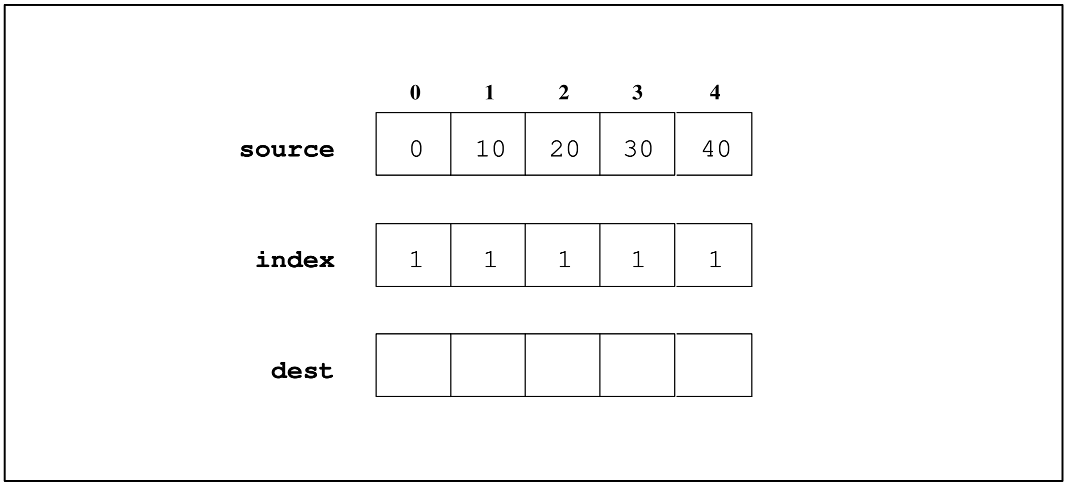

Figure 36. An index with the same value in each element.

For a Get Operation

Using the data in Figure 36, the result of this get operation is straightforward:

```
dest = [index]source;
```

For each element of **`dest`**, the **`index`** index into **`source`** is 1. This means that the value in element 1 of **`source`**, 10, is assigned to each element of **`dest`**, as shown in Figure 37.

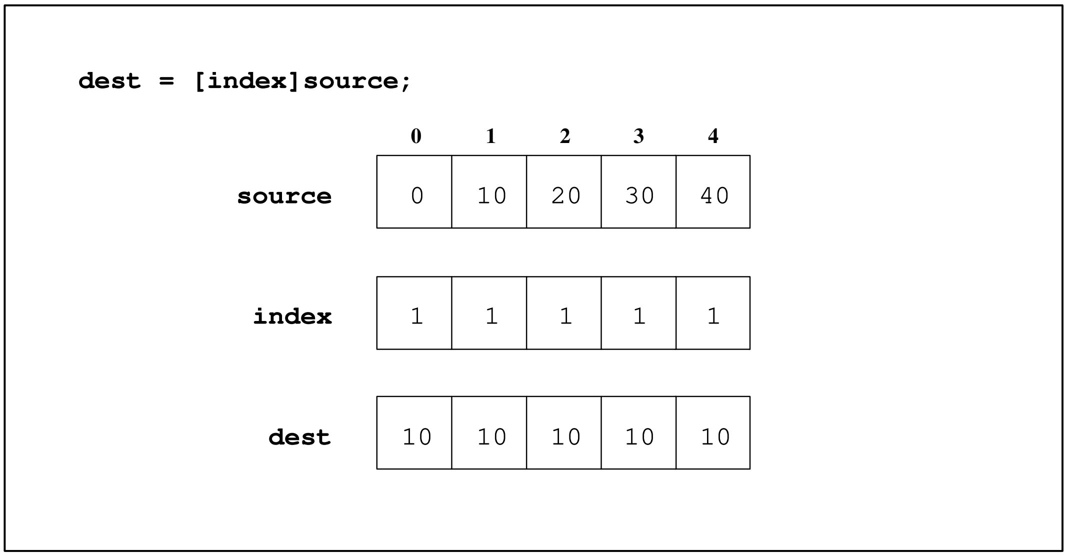

Figure 37. A get operation where the index has the same value in each element.

It is equivalent to this C* code:

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

except that all operations are carried out at the same time, in parallel.

For a Send Operation

If we try this, however:

```
[index]dest = source;
```

we have a problem. For each element of **source**, the index into **dest** is 1. This means that all the values of all the elements of **source** attempt to write into element 1 of **dest**. In serial C* code:

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

This is an example of potential *collisions*, which could occur when more than one element tries to write into the same element at the same time. To avoid the collisions, C* chooses one of the **source** elements to assign to **[1]dest**. How it chooses the element is defined by the implementation.

You can use any C* reduction assignment operator in this situation. For example, we could specify this:

```
[index]dest += source;
```

This statement says: *If there is going to be a collision of* **source** *values assigned to any of the elements of* **dest**, *add the values of the* **source** *elements that would otherwise collide, then add this result to the value of the* **dest** *element.*

In cases where there are no collisions, the value of the **source** element is simply added to the value of the **dest** element. In the example, all the values of **source** are summed, and the result is assigned to element 1 of **dest**, as shown in Figure 38. (Note that if you *knew* that all the index values were the same, it would be more efficient to use a simple unary reduction operator instead of doing parallel left indexing.)

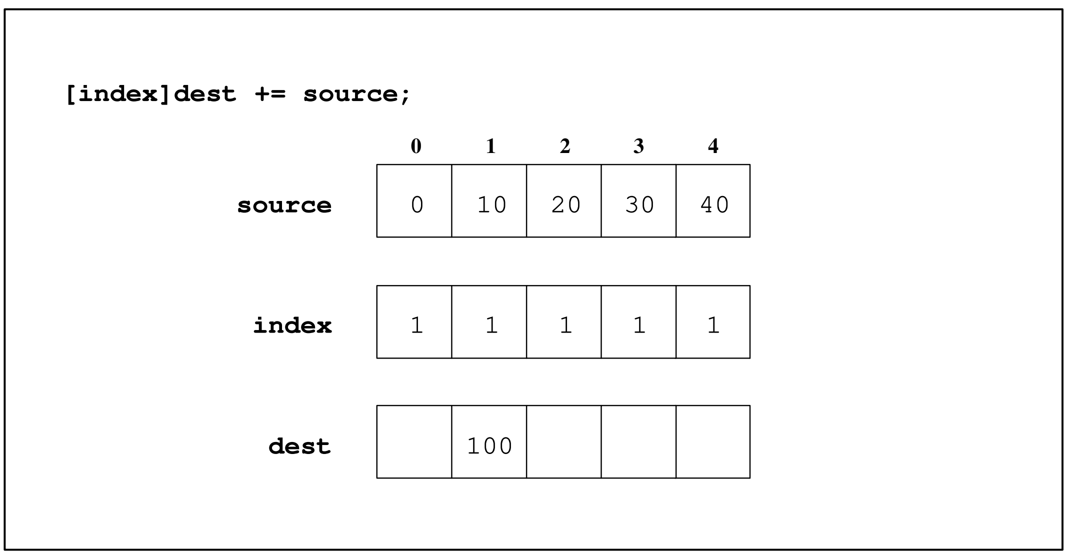

Figure 38. A reduction assignment when the parallel left index is on the LHS.

The kind of reduction assignment operator you use specifies the way the colliding elements are combined. For example, the **>?=** operator selects the maximum value of the elements.

Note that the reduction occurs only for elements that would otherwise collide. Given the examples shown in the previous section, for example, the type of reduction assignment you use would not matter, because there are no possible collisions. This is consistent with the way parallel-to-scalar reduction operators work, because all values of the parallel variable will collide when they are assigned to a scalar variable; therefore, all must be included in the specified reduction operation.

To sum up:

- In a get operation, you don't have to consider using a reduction assignment operator, because there are no potential collisions.
- In a send operation, there may be potential collisions. If you simply use **=** instead of a reduction assignment operator, and there is a potential collision, C* picks one of the colliding values and assigns it to the element.

10.1.6 When There Are Inactive Positions

The examples of parallel left indexing shown so far have assumed that all positions are active. What happens when a **where** statement makes some positions inactive?

For a Get Operation

Consider this get operation:

```
where (source < 30)
    dest = [index]source;
```

In this situation, the **where** statement deselects positions [3] and [4], using the data shown in Figure 39, but it deselects them only for *getting* purposes. Parallel variable elements in these positions cannot get values; however, elements in active positions can get values from them. The serial C* code would therefore be:

```
[0]dest = [1]source;
[1]dest = [3]source;
[2]dest = [0]source;
```

except that all operations occur at the same time. Figure 39 shows the results; the arrows show how **[1]dest** gets its value.

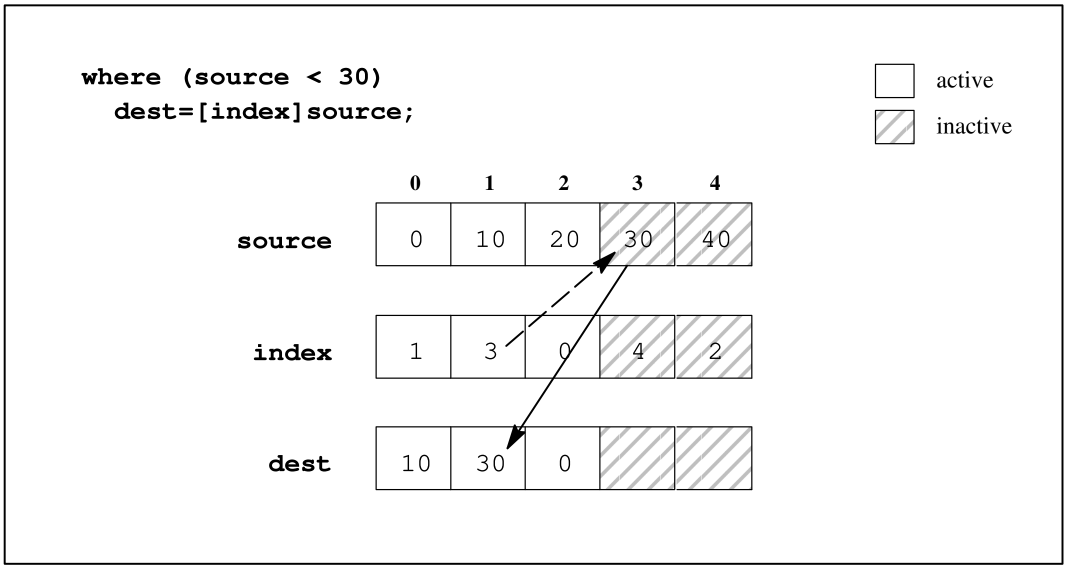

Figure 39. A get operation with inactive positions.

Note these results:

- **[1]dest** gets a value from **[3]source**, even though position [3] is inactive.
- **[4]dest** does not get a value from **[2]source**, because position [4] is inactive.

For a Send Operation

Send operations work similarly:

```
where (source < 30)
    [index]dest = source;
```

The **where** statement "turns off" positions 3 and 4, as shown in Figure 40. But it turns them off only for *sending* purposes. Elements in inactive positions cannot send values, but elements in active positions can send to them. Thus, the serial C* version of this statement would be:

```
[1]dest = [0]source;
[3]dest = [1]source;
[0]dest = [2]source;
```

The results are shown in Figure 40; the arrows show how the value in **[1]source** is sent to **[3]dest**.

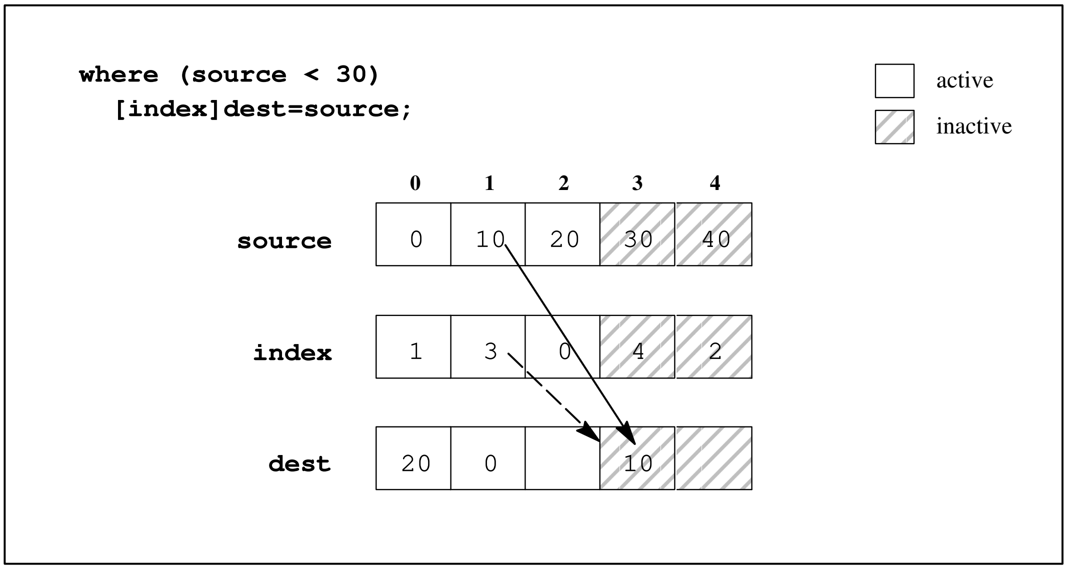

Figure 40. A send operation with inactive positions.

Note these results:

- **[1]source** sends its value to **[3]dest**, even though position [3] is inactive, because position [1] is still active.
- **[4]source** does not send its value to **[2]dest** because position [4] is inactive.

One way to look at the concept of inactive positions in these situations is that the parallel variable without the parallel left index is the one doing the work (sending or getting). When a position is made inactive, it can't do work, but it can have work done to it. Thus:

- In a send operation, the inactive position can't send, but other positions can send to it.
- In a get operation, the inactive position can't get, but other positions can get from it.

Send and Get Operations in Function Calls

As we mentioned in Section 8.2, you should be careful about passing a parallel variable by value to a function that involves the parallel variable in a send or get operation. If there are inactive positions when the function is called, the results may not be what you expected.

For example, suppose we define this function:

```
int:current get_op(int:current source, int:current index)
{
    return ([index]source);
}
```

If we use the data and the context from Figure 39, we get the results shown in Figure 41.

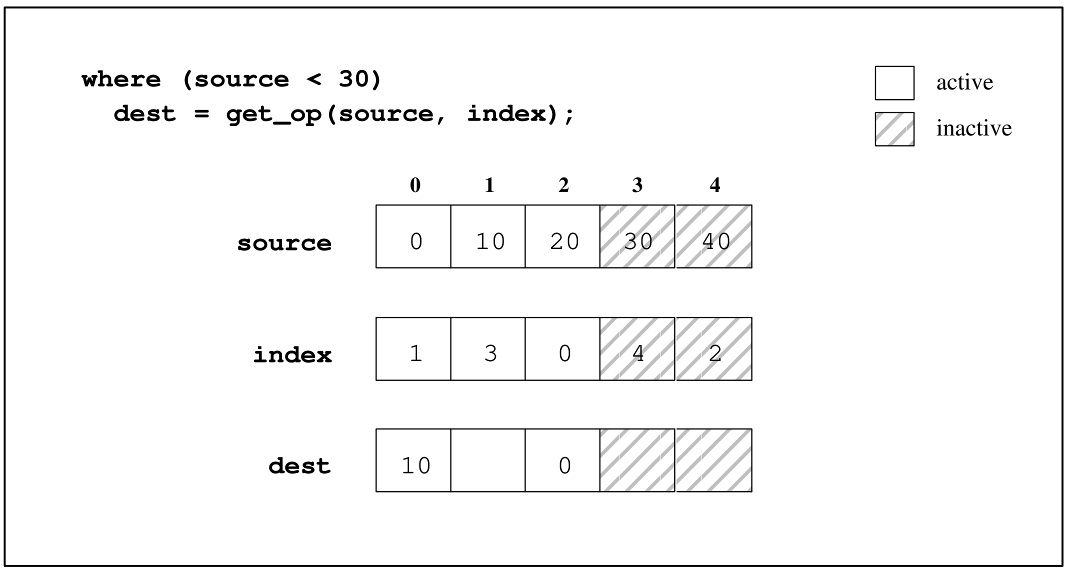

Figure 41. A function that includes a get operation.

Note the difference in results between Figure 39 and Figure 41: In Figure 39, **`[1]dest`** got its value from **`[3]source`**, even though position [3] was inactive. In Figure 41, **`[1]dest`** receives an undefined value. This happens because the compiler makes a copy of a parallel variable when it is passed by value, and elements at inactive positions receive undefined values.

The solution is to pass **source** by reference. In that case, the compiler does not make a copy of the parallel variable, and the function can gain access to values at inactive positions.

Note that in send operations it is the **dest** parallel variable that should be passed by reference, since positions can send to an inactive destination.

10.1.7 Mapping a Parallel Variable to Another Shape

One use of the parallel left index is to map a parallel variable into another shape. Consider the situation shown in Figure 42.

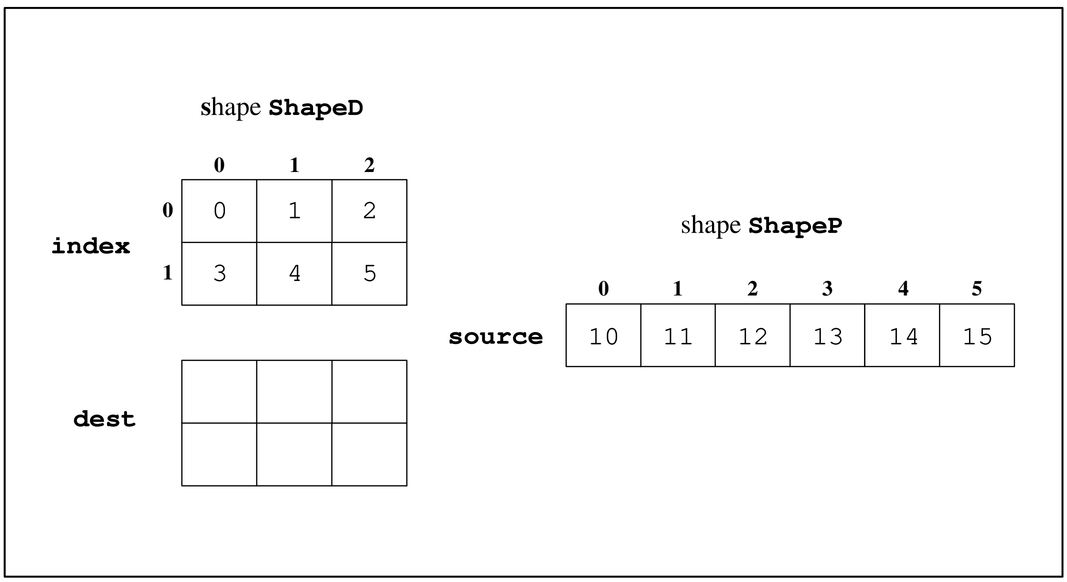

Figure 42. Two shapes.

The statement:

```
dest = [index]source;
```

has the same interpretation as before: Elements of **dest** get values of **source**, based on the value in the corresponding element of **index**. But in this situation, we are essentially mapping **source** into shape **ShapeD**, based on **index**. **ShapeD** must be the current shape. Since the values in **index** are the same as the coordinates for **ShapeP**, the assignment is straightforward: the value of **index** for position [0][0] is 0; this value is used as an index into the elements of **`source`**. The value of element [0] of **`source`** is 10; therefore, 10 is assigned to element [0][0] of **`dest`**.

The mapping occurs only for the specified operation; it does not permanently affect the parallel variable being mapped. For example, **`source`** remains of shape **`ShapeP`** after the operation above.

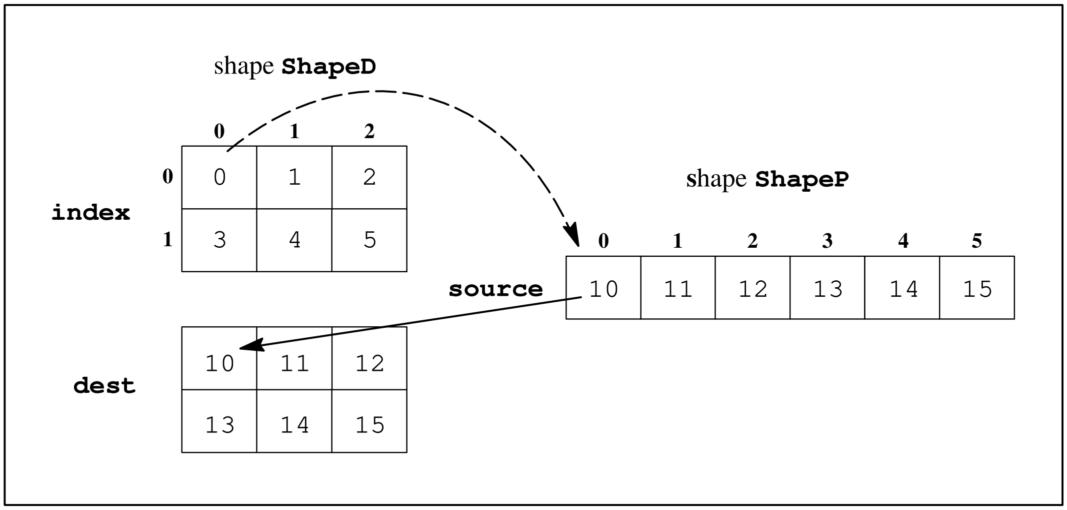

Figure 43. Mapping a parallel variable to another shape.

If a parallel variable is not of the current shape, you can use a parallel left index to map it to the current shape and then operate on it. For example:

```
shape [64][64]ShapeD;
int:ShapeD index, dest;
shape [16384]ShapeP;
int:ShapeP source;

/* Code to initialize variables omitted. */

main()
{
    with (ShapeD) {
         dest = source;    /* This doesn't work--source
                                 is the wrong shape. */
         dest = [index]source;  /* This does work. */
    }
}
```

Only active elements of a parallel left index participate in the indexing. If we add a **where** statement to the code example above and assume the data shown in Figure 42:

```
/* ... */
with (ShapeD) {
    where (index != 0)
          [0][0]dest += [index]source;
}
```

the value of element [0] of **source** is not included in the summation.

10.1.8 Limitation of Using Parallel Variables with a Parallel Left Index

A parallel variable with a parallel left index is a modifiable lvalue; therefore, it can appear as the left operand of assignment operators, as the operand of prefix or postfix **++** or **--**, and in all cases where an rvalue is needed. You cannot, however, take the address of it using the **&** operator. (In general, this would require a parallel pointer handle, which isn't supported in C*.)

10.1.9 What Can Be Left-Indexed

Parallel left indexing follows the general rules about performing parallel operations within the current shape; see Section 4.4. Specifically:

- If an expression is of the current shape, you can always left-index it.
- If an expression is not of the current shape, you can left-index it when it is any of these:
    - A simple identifier.
    - A per-processor array that is not of the current shape, if it is right-indexed by a scalar value. (You cannot left-index an array that is not of the current shape if it has a parallel right index, because that would require a parallel operation on a variable not of the current shape.)
    - A parallel variable with the **&** operator applied to it to take its address.

- A member of a parallel structure or union that is not of the current shape (so long as the member is not an aggregate type, such as another structure or union).

10.1.10 An Example: Adding Diagonals in a Matrix

The example in this section uses a parallel left index and the **+=** reduction assignment operator to add diagonals in a matrix. It uses the data shown in Figure 44.

shape **ShapeA**

source

	0	**1**	**2**	**3**
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

index

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

Figure 44. Two 4-by-4 parallel variables.

The task is to add the values of **source** in the diagonals of the matrix. The code below accomplishes this.

```
shape [4][4]ShapeA;
shape [7]ShapeB;

int:ShapeA source, index;
int:ShapeB dest = 0;
```

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

main()
{
    with (ShapeA)
         [index]dest += source;
}
```

As you can see, the actual computation is quite simple, once the data has been set up properly. Let's look in detail at the statement:

```
[index]dest += source;
```

First, note that the statement is legal, even though **dest** is not of shape **ShapeA**, since **dest** is left-indexed by a parallel variable that *is* of that shape. The statement says: *Use* **index** as *an index into* **dest** for *sending values of* **source***; if there are potential collisions, add the values of* **source**. So, for example, element [0][0] of parallel variable **source** is assigned to element [3] of **dest**, because the value of the corresponding element of **index** is 3. Element [1][1], element [2][2], and element [3][3] are also assigned to element [3] of **dest**. They are all added, thus avoiding collisions.

The other elements of **source** are also assigned to **dest**, based on the value of the corresponding elements of **index**. The result is the addition of the diagonals. Figure 45 shows the results, highlighting the values that go into **[3]dest**.

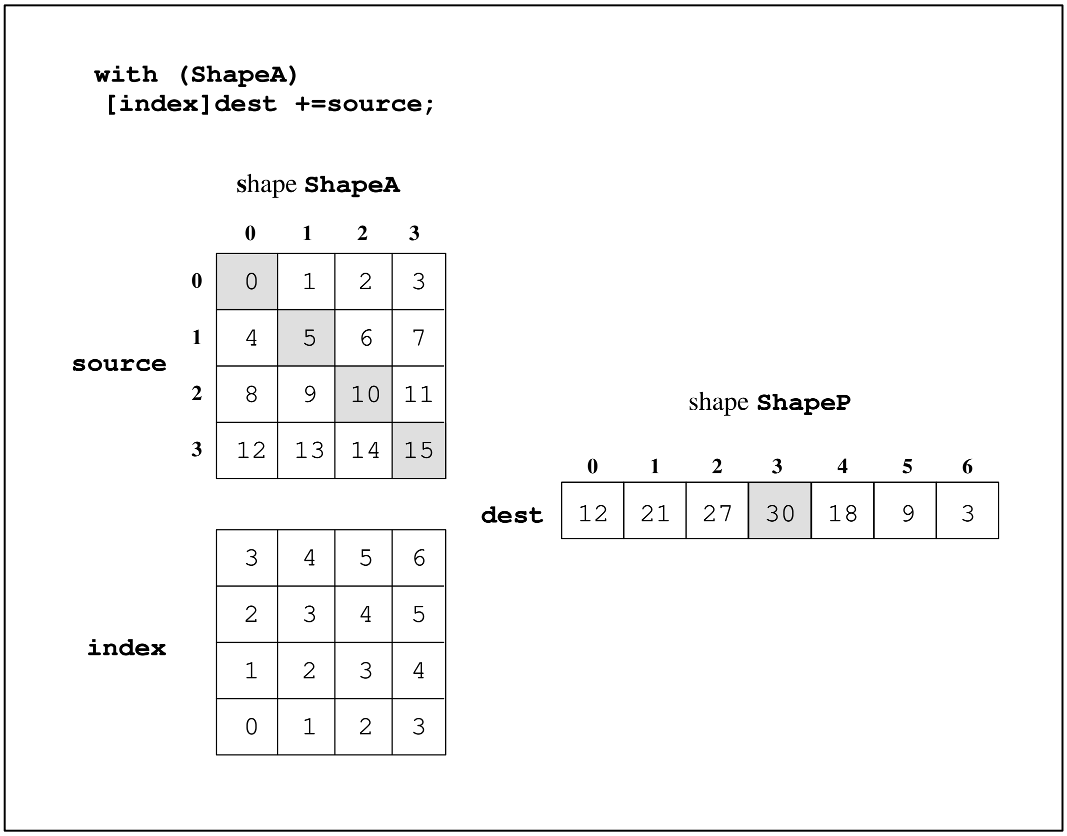

Figure 45. Using parallel left indexing to add the diagonals of a matrix.

10.2 Using the pcoord Function

C* includes a new library function called **pcoord**, which is especially useful when combined with parallel left indexing. Use **pcoord** to create a parallel variable in the current shape; each element in this variable is initialized to its coordinate along the axis you specify as the argument to **pcoord**. For example,

```
shape [65536]ShapeA;
int:ShapeA p1;

main()
{
    with (ShapeA)
```

```
            p1 = pcoord(0);
}
```

initializes **p1** as shown in Figure 46.

p1 = pcoord(0);

shape **ShapeA**

	0	1	2	3	4	5	6	7	8		65535
p1	0	1	2	3	4	5	6	7	8	...	65535

Figure 46. The use of pcoord with a 1-dimensional shape.

Likewise, for a 2-dimensional shape,

```
shape [4][4096]ShapeB;
int:ShapeB p2;

main()
{
    with (ShapeB)
         p2 = pcoord(1);
}
```

initializes **p2** as shown in Figure 47.

p2 = pcoord(1);

		0	1	2	3		4095
p2	0	0	1	2	3	...	4095
	1	0	1	2	3	...	4095
	2	0	1	2	3	...	4095
	3	0	1	2	3	...	4095

Figure 47. The use of pcoord with axis 1 of a 2-dimensional shape.

Similarly,

```
with (ShapeB)
    p2 = pcoord(0);
```

initializes **`p2`** as shown in Figure 48.

`p2 = pcoord(0);`

p2

	0	1	2	3		4095
0	0	0	0	0	...	0
1	1	1	1	1	...	1
2	2	2	2	2	...	2
3	3	3	3	3	...	3

Figure 48. The use of `pcoord` with axis 0 of a 2-dimensional shape.

The **`pcoord`** function provides a quick way of creating a parallel left index for mapping a parallel variable into another shape. For example:

```
shape [16384]ShapeA, [16384][4]ShapeB;
int:ShapeA source;

/* Code to initialize source omitted. */

main()
{
    with (ShapeB) {
         int:ShapeB index, dest;
         index = pcoord(0);
         dest = [index]source;
    }
}
```

Rather than assign the results of **`pcoord`** to a parallel variable, you can simply use it as the parallel left index itself:

```
dest = [pcoord(0)]source;
```

The index of the specified axis of the current shape is generated by **pcoord**. This index is used as an index for selecting elements of a parallel variable of another shape. The values of these elements are assigned to elements of a parallel variable of the current shape.

10.2.1 An Example

This example uses **pcoord** to transpose a matrix — in other words, to turn its rows into columns and its columns into rows. For example, consider the simple 3-by-3 parallel variable called **matrix** shown on the left in Figure 49. The task is to turn it into the new matrix shown on the right.

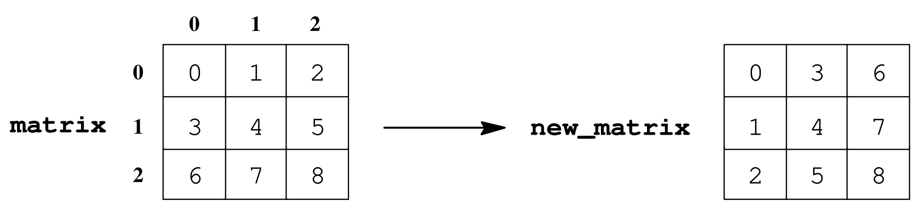

Figure 49. Transposing a 3-by-3 matrix.

This can be done by reversing the axes for the parallel variable **matrix**. For example, **[0][1]matrix** (which contains the value 1) becomes element **[1][0]** of a new parallel variable. To do this for a 256-by-256 matrix, use **pcoord** as follows:

```
Shape [256][256]ShapeA;
int:ShapeA matrix, new_matrix;

main()
{
    with (ShapeA)
         [pcoord(1)][pcoord(0)]new_matrix = matrix;
}
```

The statement

```
[pcoord(1)][pcoord(0)]new_matrix = matrix;
```

says: *Assign each element of* **matrix** *to* **new_matrix**, *but reverse the axis numbering.* Thus, in serial C* code:

```
[0][0]new_matrix = [0][0]matrix;
[0][1]new_matrix = [1][0]matrix;
[0][2]new_matrix = [2][0]matrix;
[1][0]new_matrix = [0][1]matrix;  /* And so on */
```

except that all operations take place at the same time. This algorithm can be generalized for use in a function with any 2-dimensional parallel variable:

```
void transpose(float:current *matrixp,
          float:void *new_matrixp)
{
    [pcoord(1)][pcoord(0)]*new_matrixp = *matrixp;
}
```

Note these points about **transpose**:

- It passes two pointers to parallel variables. **matrixp** is a pointer to a parallel variable of the current shape; we pass a pointer rather than the parallel variable itself to avoid having to make a copy of the variable. **new_matrixp** is a pointer to a parallel variable of a new shape; we *must* pass a pointer in this case because we will be modifying the variable — therefore, it can't be passed by value.

- We use a second shape so that the function can work with a matrix that isn't square. For example, if the current shape is 256 by 512, make **new_matrixp** a pointer to a parallel variable of a shape that is 512 by 256.

- The variable pointed to by **matrixp** is assigned to the variable pointed to by **new_matrixp**, and this variable has its coordinates reversed.

10.3 The pcoord Function and Grid Communication

When used with parallel left indexing, **pcoord** provides the grid communication capabilities we discussed at the beginning of this chapter.

Consider this statement, where both **dest** and **source** are of the current shape:

```
dest = [pcoord(0) + 1]source;
```

This statement says: *Each active element of* **dest** *is to get the value of* **source** *that is in the position one coordinate higher along axis 0.* You can either add a scalar value to or subtract a scalar value from **pcoord** in the left index. Which operation you choose determines the direction of the communication; the value added or subtracted specifies how many positions along the axis the values are to travel. Note, however, that the values must stay within the border of the grid; the behavior is undefined if **dest** tries to get a nonexistent element of **source**.

You can use **pcoord** for a send operation as well as for a get operation; send and get operations are discussed in Section 10.1. For example:

```
[pcoord(0) + 1]dest = source;
```

This statement says: *Send the value of the* **source** *element to the* **dest** *element that is one position higher along axis 0.*

You can use **pcoord** to specify movement along more than one dimension. For example:

```
dest = [pcoord(0) - 2][pcoord(1) + 1]source;
```

Note that specifying the axes in this kind of statement provides redundant information. By definition, the first pair of brackets contains the value for axis 0, the next pair of brackets contains the value for axis 1, and so on. C* therefore lets you simplify the expression by substituting a period for **pcoord**(*axis–number*). The period is position-dependent. If it is in the first pair of brackets, it means **pcoord(0)**; if it appears in the second pair of brackets, it means **pcoord(1)**, and so on. Thus, this statement is equivalent to the statement above:

```
dest = [. - 2][. + 1]source;
```

10.3.1 Grid Communication without Wrapping

As we noted above, behavior is undefined when elements try to get or send beyond the border of the grid. This means that the statements shown so far are not especially useful, because they do not solve this problem. What happens to the elements of **dest** in row 0 when they try to get from **[pcoord(0)-1]** — that is, from beyond the border of the grid?

For this kind of statement to work, you must first use a **where** statement to turn off positions that would otherwise get or send beyond the border of the grid. For example, if you want elements to get from elements two coordinates lower along axis 0 (that is, position 2 gets from position 0, position 3 gets from position 1, and so on), you must turn off positions 0 and 1, because elements in these positions would otherwise attempt to get nonexistent values. The code below accomplishes this:

```
where (pcoord(0) > 1)
    dest = [. - 2]source;
```

If you want to get from a parallel variable two coordinates higher along axis 0 (position 0 gets from position 2, and so on), you can use the **dimof** intrinsic function to determine the number of positions along the axis. For example:

```
where (pcoord(0) < (dimof(ShapeA, 0) - 2))
    dest = [. + 2]source;
```

Note that you must subtract 2 from the result returned by **dimof** to turn off the correct number of positions. If **dimof** returns 1024, the positions are numbered 0 through 1023. To turn off positions 1022 and 1023, you must subtract 2 from 1024 and specify that the result of calling **pcoord** is to be less than this.

10.3.2 Grid Communication with Wrapping

To perform grid communication in which the values "wrap" back to the other side of the grid, we once again need to use the **dimof** intrinsic function. Consider this statement:

```
dest = [(. + 2) %% dimof(ShapeA, 0)]source;
```

The expression in brackets does this:

1. It adds 2 to the coordinate index returned by **pcoord**.
2. For each value returned, it returns the modulus of this number and the number of positions along the axis.

Step 2 does not affect the results as long as step 1 returns a value that is less than the number of coordinates along the axis. For example, if **(. + 2)** is 502 in a 1024-position axis, the result of **(502 %% 1024)** is 502. When step 1 returns a value equal to or greater than the number of coordinates along the axis, step 2 achieves the desired wrapping. For example, element [1022] of **dest** attempts to get from element [1024] of **source**, which is beyond the border of the grid.

But **(1024 %% 1024)** is 0, so instead **[1022]dest** gets from **[0]source**. Thus, the **%%** operator provides the wrapping back to the low end of the axis.

Similarly,

```
dest = [(. - 2) %% dimof(ShapeA, 0)]source;
```

provides wrapping to the high end of the axis. For this statement, let's look at the case where **[0]dest** tries to get a value from the element of **source** that is two lower along axis 0. If there are 1024 coordinates along the axis, this produces the expression **(-2 %% 1024)** for the left index of **source**. Following the procedure for **%%** shown on page 52, we find that the result of this expression is 1022. This is the element of **source** from which **[0]dest** gets its value.

Note that you cannot use the Standard C operator **%** to perform these operations, because different implementations of **%** can give different answers when one or both of its operands is negative. The **%%** operator guarantees that the sign of the answer is the same as the sign of the denominator, which is what is required.