Chapter 5

Communication

In the previous chapter we talked about how C* performs operations on individual parallel variables, or on corresponding elements of parallel variables of the current shape. Many problems, however, require more complex interactions — for example, they require data to be shifted along an axis of a parallel variable, or between parallel variables of different shapes. We refer to these kinds of interactions as *communication*. This chapter describes some of the ways in which C* lets parallel variables communicate.

5.1 Kinds of Communication

C* provides two kinds of communication for parallel variables:

- *General communication*, in which the value of any element of a parallel variable can be sent to any element of any other parallel variable, whether or not the parallel variables are of the same shape.
- *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.

Both kinds of communication can be expressed through the syntax of the language, as well as through functions provided in the C* communication library. This chapter briefly describes how to use C* syntax. For complete information on the C* communication library, see the *C* Programming Guide*. In addition to functions that perform grid and general communication, the library also provides functions that perform certain useful transformations of parallel data. For example:

- The **scan** function calculates running results for various operations on a parallel variable.
- The **spread** function spreads the result of a parallel operation into elements of a parallel variable.
- The **rank** function produces a numerical ranking of the values of parallel variable elements; this is useful for sorting parallel data.

5.2 General Communication

General communication involves the concept of *parallel left indexing*. So far, we have seen only constants in left indexes — for example, **[0]p1**; you can also use a scalar variable in a left index. But what if you use a parallel variable as an index for another parallel variable? If **p0** and **p1** are both 1-dimensional parallel variables, what does **[p0]p1** mean?

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, but the parallel variable it is indexing need not be.

In the example below, **source**, **dest**, and **index** are all parallel variables of the current shape. We want to assign values of **source** to **dest**, but reversing their order. To do this, we use **index** as a parallel left index for **dest**; the values in **index** control where **source** is to send its values. The statement looks like this:

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

Figure 12 shows some sample data, and the result of the assignment. The arrows point out what happens for the value of **[0]source**. The value in the corresponding element of **index** is 4; therefore, the value in **[0]source** is assigned to **[4]dest**.

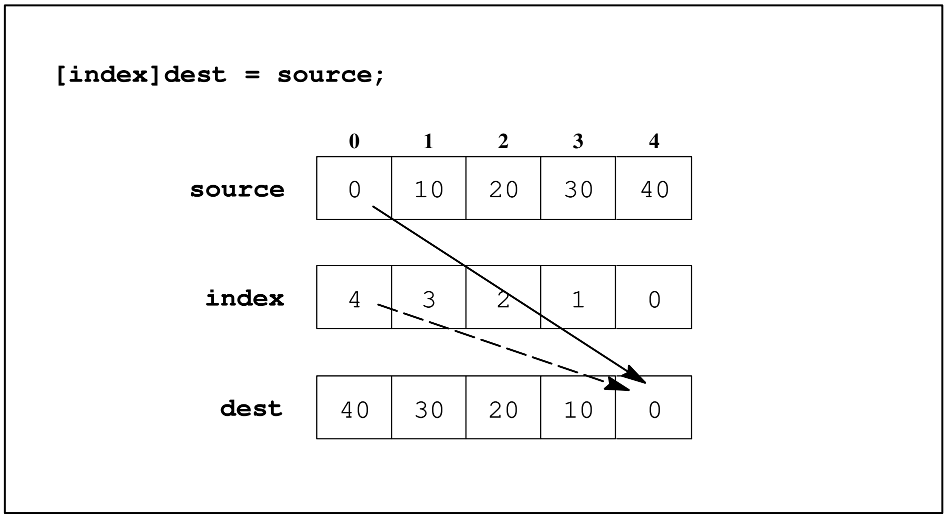

Figure 12. Parallel left indexing — a send operation.

Note that **dest** need not have been of the current shape.

If a parallel variable's shape has more than one dimension, there must be an index for each axis of the shape. For example:

```
[index0][index1]dest = source;
```

Figure 12 above is an example of a *send operation*. In a send operation, the index parallel variable is applied to the destination parallel variable on the left-hand side.

There is a special use of reduction operators with send operations. It is possible that the values in elements of an index variable could be the same. This would cause more than one value to be sent to the same element of the destination parallel variable. You can use a reduction operator to specify what is to be done if this occurs. For example,

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

specifies that the values are to be added to the value of the element of **dest**. If you do a simple assignment, and multiple values are being sent to the same element, the compiler picks one of the values and assigns it to the element.

In addition to the send operation, C* also has a *get operation*. In a get operation, the index parallel variable is applied to the source parallel variable on the right-hand side. For example:

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

Figure 13 shows some sample data for a get operation. In the figure, the arrows show how **[0]dest** gets its value. The value in the corresponding element of **index** is 1; therefore, **[0]dest** gets its value from **[1]source**.

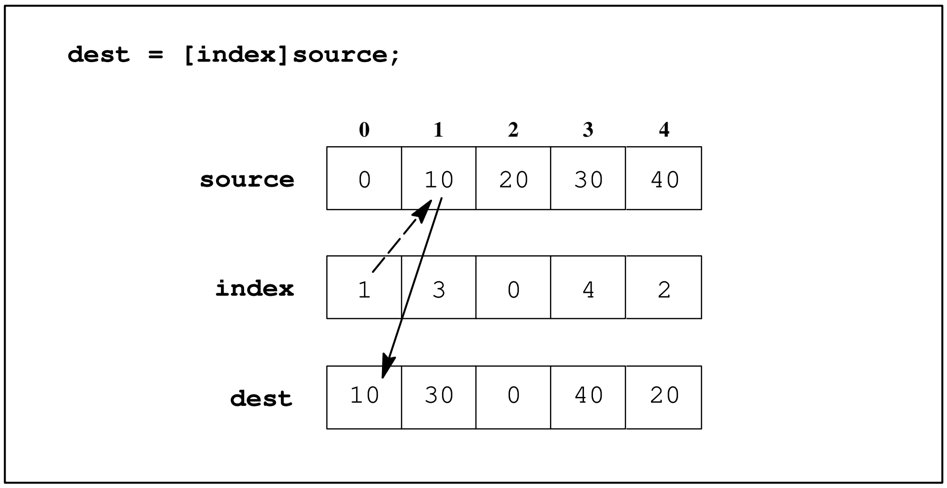

Figure 13. Parallel left indexing of a parallel variable — a get operation.

In a get operation, the destination and index parallel variables must be of the current shape, but the source parallel variable can be of any shape.

When There Are Inactive Positions

The element of the parallel variable that initiates an operation must be active. That is:

- In a send operation, an inactive position cannot send a value, but active positions can send to it.
- In a get operation, an inactive position cannot get a value, but active positions can get from it.

Figure 14 shows an example for a send operation.

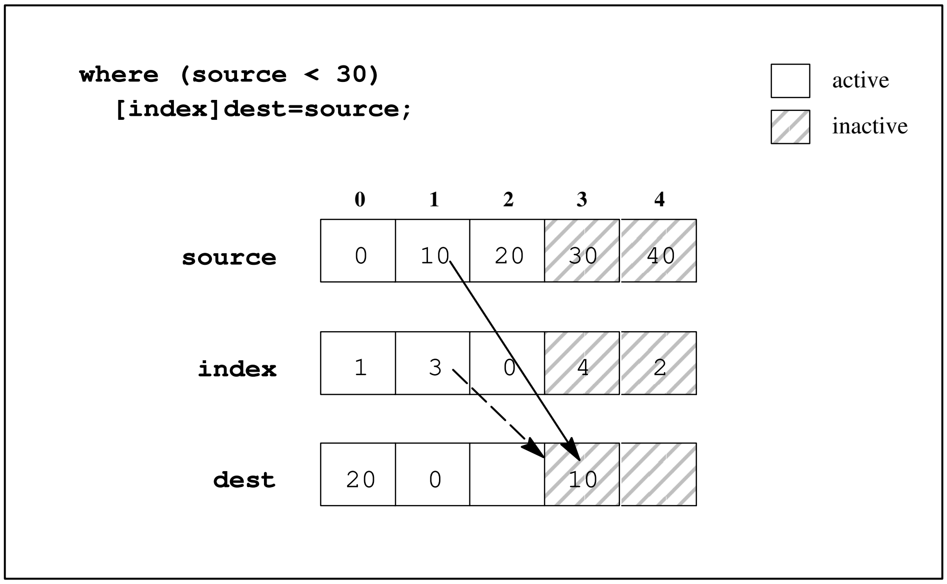

Figure 14. A send operation with inactive positions.

As the arrows in the example show, **`[1]source`** sends its value to **`[3]dest`**, even though position [3] is inactive. However, **`[4]source`**, for example, does not send its value to **`[2]dest`**, because position [4] is inactive.

5.3 Grid Communication

Grid communication in C* involves left indexing and the C* library function **`pcoord`**.

The pcoord Function

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 15.

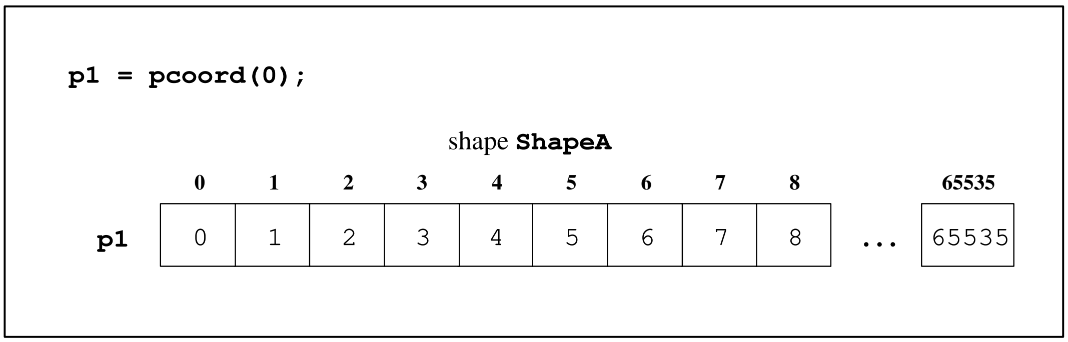

Figure 15. The use of pcoord with a 1-dimensional shape.

The argument to **pcoord** is the axis along which the indexing is to take place; for example, in a 2-dimensional shape, **pcoord(1)** initializes the elements to their coordinates along axis 1.

The pcoord Function and Left Indexing

The **pcoord** function provides a quick way of creating a parallel left index for mapping a parallel variable onto another shape. For example, if **dest** is of the current shape and **source** is of some other (1-dimensional) shape,

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

maps **source** onto the current shape, and assigns the values of the elements of **source** to **dest**, based on this mapping.

Now let's assume that **dest** and **source** are both of the current (1-dimensional) shape. Consider this statement:

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

The statement says: *Send the value of the* **source** *element to the* **dest** *element whose coordinate is one position higher.* This syntax provides grid communication along an axis of a shape. You can add a scalar value to, or subtract a scalar value from, **pcoord** in a left index. Which operation you choose determines the direction of the communication; the value added or subtracted specifies how many positions the values are to travel along the axis.

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 axis in the statements shown above provides redundant information when you are performing grid communication. 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***)**. Thus, this statement is equivalent to the one above:

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

This use of the period is a common idiom in C* programs.

Grid Communication without Wrapping

The left-indexed **pcoord** statements shown so far are not useful by themselves, because they do not specify what happens when elements try to get from or send to positions that are beyond the border of an axis; this behavior is undefined in C*.

One way of solving this problem is to first use a **where** statement to restrict the context to those positions that do *not* get or send beyond the border of an axis. For example, if you want **dest** to get from **source** 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 make positions 0 and 1 inactive, because elements in these positions would otherwise attempt to get nonexistent values. The following code accomplishes this:

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

Figure 16 is an example; the arrow shows **`[2]dest`** getting a value from **`[0]source`**.

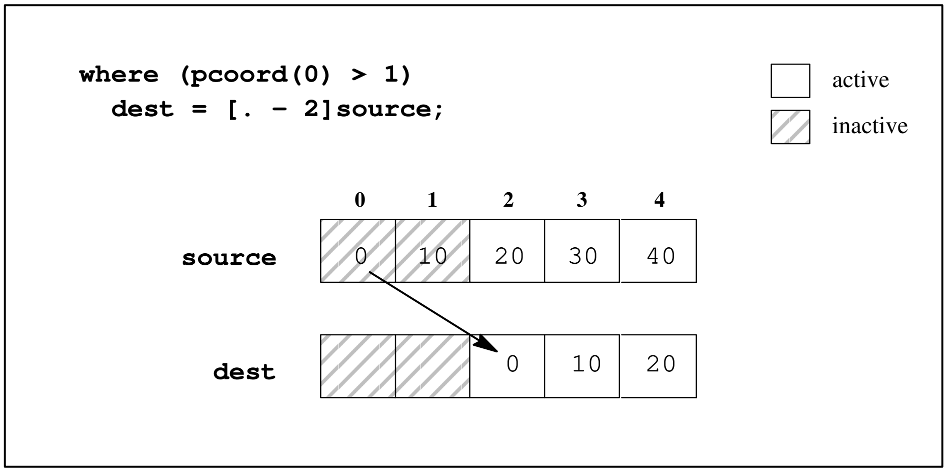

Figure 16. Grid communication without wrapping.

If you want to get from a parallel variable two coordinates higher along axis 0, you can use the C* intrinsic function **`dimof`** to determine the number of positions along the axis. **`dimof`** takes two arguments: a shape and an axis number; it returns the number of positions along the specified axis. For example:

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

Grid Communication with Wrapping

The examples in the previous section solve the problem of getting or sending beyond the border of an axis by making the troublesome positions inactive. In some situations, however, you might prefer to have values wrap back to the other side of the axis. To do this, we once again use the **`dimof`** function, along with the new C* modulus operator **`%%`**. (The **`%%`** operator gives the same answers as the Standard C operator **`%`** when both operands are positive. It can differ when one or both operands are negative; see the *C* Programming Guide* for complete details.)

Consider this statement:

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

The expression in brackets does the following:

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. In the example above, assume that axis 0 of shape **ShapeA** contains 1024 positions; therefore, **dimof(ShapeA, 0)** returns 1024. The result of **(502 %% 1024)** is 502, for example, so **[500]dest** gets from **[502]source**. When Step 1 returns a value equal to or greater than the number of coordinates along the axis, however, Step 2 achieves the desired wrapping. For example, **[1022]dest** attempts to get from element **[1024]source**, which is beyond the border of the axis. 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.