Chapter 4

Parallel Operations

Operations in C* that involve only scalar variables work exactly as they do in Standard C. As we mentioned in the previous chapter, these operations can take place anywhere in the program; you don't have to worry about the current shape or the context.

This chapter gives an overview of how to perform operations that involve parallel variables. It also describes the use of parallel variables and shapes in functions.

4.1 Standard C Operators

Unary Operators

You can use Standard C unary operators with parallel variables of the current shape. Each active element of the parallel variable performs the operation separately, at the same time. For example, if **p1** is a parallel variable of the current shape,

```
p1++;
```

increments the value in every active element of **p1**.

Binary Operators with a Scalar Operand and a Parallel Operand

You can use Standard C binary operators when one of the operands is parallel and one is scalar; the parallel variable must be of the current shape. In this case, the scalar value is first *promoted* to a parallel value of the shape of the parallel operand, and this parallel value is used in the operation. For example, if **p1** is a parallel variable of the current shape,

```
p1 = 12;
```

assigns the value 12 to each active element of **p1**. If **s1** is scalar,

```
p1 = s1;
```

assigns the value of **s1** to each active element of **p1**. Similarly,

```
p2 = p1 + s1;
```

adds the value of **s1** to each active element of **p1**, and assigns the result to the corresponding element of **p2**. (Corresponding elements are discussed in Chapter 3.) Both **p1** and **p2** must be of the same shape.

Assignment with a Scalar LHS and a Parallel RHS

A scalar variable is not promoted when it is on the left-hand side of an assignment statement. If **s1** is scalar and **p1** is parallel, this statement is illegal in C*:

```
s1 = p1;  /* This is wrong */
```

To assign a parallel variable to a scalar variable, you must explicitly *demote* the parallel variable to a scalar variable, by casting it to the type of the scalar variable. Thus, if **s1** is an **int**, this code works:

```
s1 = (int)p1;   /* This works */
```

In this case, C* simply chooses one value from the active elements of the parallel variable and assigns that value to the scalar variable. C* does not specify which of the values will be chosen; it could be different for different implementations of the language.

Binary Operators with Two Parallel Operands

Standard binary C operators work with two parallel operands if both are of the current shape. These operations once again bring in the concept of corresponding elements. For example, if **p1** and **p2** are parallel variables of the current shape,

```
p2 = p1;
```

simply assigns the value in each active element of **p1** to the corresponding element of **p2**, as shown in Figure 11.

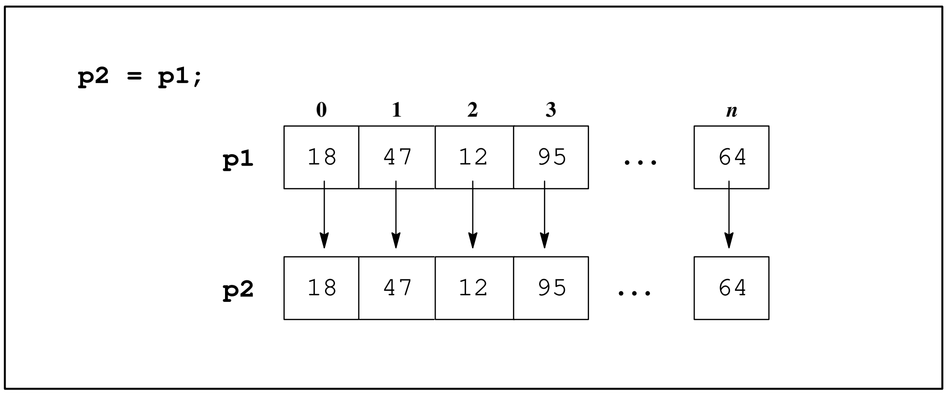

Figure 11. Assignment of a parallel variable to a parallel variable.

Similarly,

```
p3 = (p1 >= p2);
```

assigns to **p3**, for each corresponding active element of **p1**, 1 if it is greater than or equal to the corresponding element of **p2**, and 0 if it is not.

The Conditional Expression

The conditional expression **?:** operates in slightly different ways depending on the mix of parallel and scalar variables in the expression. Consider the statement below, where **s1** is scalar, and **p1**, **p2**, and **p3** are parallel variables of the current shape:

```
p1 = (s1 < 5) ? p2 : p3;
```

If **s1** is less than 5, this statement assigns the value of the corresponding element of **p2** to each active element of **p1**; otherwise, **p1** is assigned the value of the corresponding element of **p3**.

The behavior is different if all the operands are parallel variables of the current shape. For example:

```
p1 = (p2 < 5) ? p3 : p4;
```

In this case, each active element of **p2** is evaluated separately. If the value in **p2** is less than 5 in a given element, the value of **p3** is assigned to **p1** for the corresponding element; otherwise, the value of **p4** is assigned to **p1**. If either **p3** or **p4** (or both) were scalar in this example, they would be promoted to parallel variables of the current shape, and the expression would be evaluated as described above.

4.2 Reduction Operators

Standard C has several compound assignment operators, such as **+=**, that perform a binary operation and assign the result to the left-hand side. Many of these operators can be used with parallel variables in C* to perform reductions — that is, they *reduce* the values of all elements of a parallel variable to a single scalar value. C* reduction operators provide a quick way of performing operations on all elements of a parallel variable.

Reduction operators can be either unary or binary. For example, if **p1** is a parallel variable of the current shape,

```
+= p1
```

sums the values of all active elements of parallel variable **p1**.

```
s1 += p1;
```

sums the values of all active elements of **p1**, and adds them to the value of scalar variable **s1**.

When used with two parallel variables, these operators perform simple elemental binary operations. For example, if **p1** and **p2** are both parallel variables of the current shape,

```
p1 += p2;
```

adds the value of each active element of **p2** to the value of the corresponding element of **p1**.

C* introduces two new reduction operators: the minimum operator, **<?=**, and the maximum operator, **>?=**. For a parallel variable, **<?=** returns the minimum value among its active elements, and **>?=** returns the maximum value.

4.3 Functions

C* adds support for parallel variables and shapes to Standard C functions. Parallel variables and shapes can be passed as arguments to and returned from functions. For example, this function takes a parallel variable of type **int** and shape **ShapeA** as an argument:

```
void print_sum(int:ShapeA x)
{
   printf ("The sum of the elements is %d.\n", +=x);
}
```

Note the use of the unary reduction operator, discussed above. Note also, however, that this function has limited usefulness, since it requires that the parallel variable be of a specific shape. C* provides more general methods for specifying shapes in functions.

Use the C* keyword **current** to specify that the parallel variable is of the current shape; **current** always refers to the current shape. By substituting **current** for **ShapeA** in the function above, we generalize the function so that it works for any current shape.

You cannot pass a parallel variable that is not of the current shape. However, you can pass a pointer to a parallel variable of a shape that isn't current. To allow you to specify a pointer to a parallel variable of *any* shape, C* extends the use of the Standard C keyword **void**. You can use **void** instead of a shape name to specify a shape without indicating its name; the actual shape is determined by the parallel variable that is ultimately assigned to the pointer.

For a function to perform most operations on a pointer to a parallel variable not of the current shape, the function must contain its own **with** statement to make the parallel variable's shape current. To do this, you can use the new C* intrinsic function **shapeof**, which takes a parallel value and returns the shape of that parallel value. For example, this function returns the sum of the active elements of a parallel variable of any shape:

```
int sum(int:void *ptr)
{
   with (shapeof(*ptr))
      return (+= *ptr);
}
```

Like a dereferenced pointer to a shape, **shapeof** is a shape-valued expression; you can use it anywhere you can use a shape name.

Passing by Value and Passing by Reference

As we have shown, you can pass parallel variables by value or by reference. In deciding which method to use, you must take into account the effect of positions made inactive by the **where** statement. (See Chapter 2 for a discussion of **where**.)

When you pass a variable by value, the compiler makes a copy of it for use in the function. If the variable is parallel and positions are inactive, elements in those positions have undefined values in the copy. This is not a problem if the function does not operate on the inactive positions; if it does, however, passing by value can produce unexpected results. The function can operate on the inactive positions if, for example, it contains an **everywhere** statement to widen the context, and then operates on a parallel variable you pass. In such a situation, you should define the function so that it passes by reference rather than by value.

Overloading Functions

It may be convenient for your program to have more than one version of a function with the same name — for example, one version for scalar data and another for parallel data. This is known as *overloading*. C* allows overloading of functions, provided that the functions differ in the type of at least one of their arguments or in the total number of arguments. For example, these versions of function **f** can be overloaded:

```
void f(int x);
void f(int x, int y);
void f(int:current x);
```

Use the **`overload`** statement to specify the names of the functions to be overloaded. For example, this statement specifies that there may be more than one version of the function **`f`**:

```
overload f;
```

The **`overload`** statement must precede an overloaded declaration. Typically, it is put at the beginning of the file that contains the declarations of the functions.