Chapter 7

Pointers

C* has three kinds of pointers:

- the Standard C pointer
- a scalar pointer to a shape
- a scalar pointer to a parallel variable

As in C, C* pointers are fast and powerful.

7.1 Scalar-to-Scalar Pointers

C* supports the Standard C pointer. For example,

```
int *ptr;
```

declares **`ptr`** to be a scalar pointer to an **`int`**. If **`s1`** is a scalar variable,

```
ptr = &s1;
```

puts the address of **`s1`** in **`ptr`**, and

```
s2 = *ptr;
```

puts the value of **`s1`** into **`s2`**.

7.2 Scalar Pointers to Shapes

C* introduces a new kind of scalar pointer that points to a shape. For example,

```
shape *ptr;
```

declares the scalar variable **ptr** to be a pointer to a shape, and

```
ptr = &ShapeA;
```

makes **ptr** point to **ShapeA**.

A dereferenced pointer to a shape can be used as a shape-valued expression. For example, if **ptr** points to **ShapeA**,

```
with (*ptr)
```

makes **ShapeA** the current shape.

Scalar pointers to shapes are discussed in more detail in Section 9.1.1, when we introduce arrays of shapes.

7.3 Scalar Pointers to Parallel Variables

C* introduces a new kind of scalar pointer that points to a parallel variable. For example,

```
int:ShapeA *ptr;
```

declares a scalar pointer **ptr** that points to a parallel **int** of shape **ShapeA**.

How can a scalar pointer point to a parallel variable? Clearly the mechanism must be different from that used in C pointers, which store the memory address of the object to which it points; each element of a parallel variable would have a different address. In fact, a pointer to a parallel variable in C* does not store a physical address, but a value that uniquely identifies the entire set of elements of the parallel variable.

Note that scalar pointers to parallel variables aren't necessarily the same size as scalar pointers to scalar values. However, they can still be operated on by the usual C pointer operations: for example, addition or subtraction with scalar values, subtraction of pointers, and comparison to zero. See Sections 7.3.2 and

7.3.3. See also Appendix C for a more in-depth discussion of the implementation of scalar pointers to parallel variables in CM-5 C*.

If **p1** is a parallel variable of shape **ShapeA**,

```
ptr = &p1;
```

stores this value for **p1** in the scalar pointer **ptr**. **p1** need not be of the current shape.

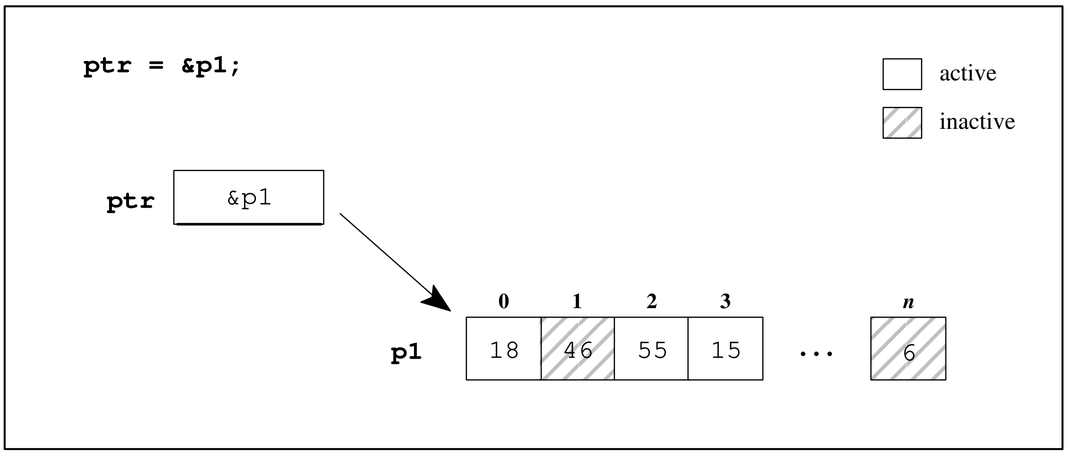

Figure 26. A scalar-to-parallel pointer.

Once the above statement has been executed, a program can reference the parallel variable **p1** via the pointer stored in **ptr**. For example,

```
(*ptr)++;
```

increments the value in each active element of **p1**, as shown in Figure 27.

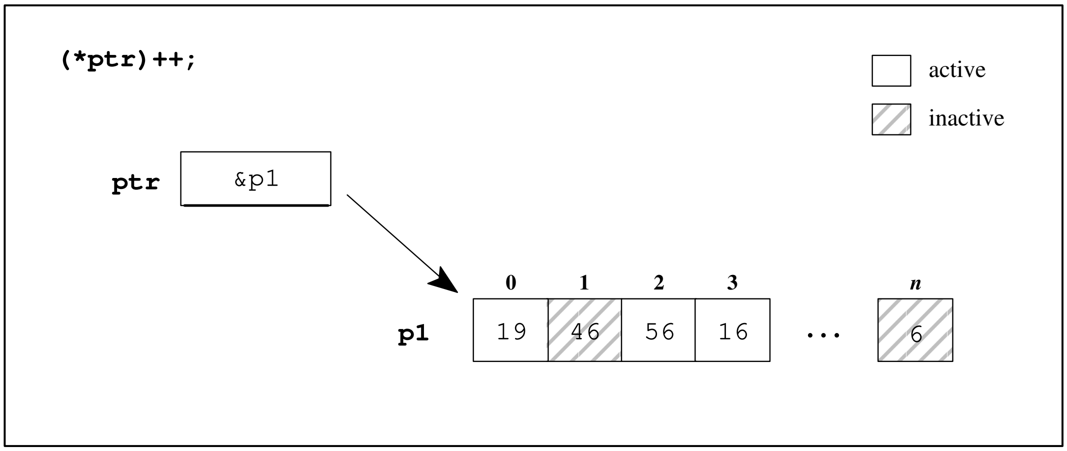

Figure 27. Dereferencing the scalar-to-parallel pointer shown in Figure 26.

If **s1** is a scalar variable,

```
s1 += *ptr;
```

sums the values of the active elements of **p1**, and adds the result to **s1**.

The constraints that apply to dealing directly with a parallel variable also apply to dealing with it via a scalar pointer. For example, **ShapeA** must be the current shape for the above statement to be executed.

7.3.1 Alternative Declaration Syntax Not Allowed

Recall from Chapter 3 that there are two ways of declaring a parallel variable:

```
int:ShapeA p1;
```

and

```
int p1:ShapeA;
```

C* does not allow the latter syntax for declaring scalar-to-parallel pointers, however:

```
int *ptr:ShapeA;  /* This is wrong */
```

In this case, the compiler interprets the shape name as applying to the pointer, and parallel-to-scalar pointers do not exist in the language.

7.3.2 Arrays

The close relationship between arrays and pointers is maintained in C*. For example,

```
int:ShapeA A1[40];
```

declares a parallel array of 40 **int**s of shape **ShapeA**, and **A1** points to the first element of the array. (Recall that an element of a parallel array is a parallel variable.)

7.3.3 Pointer Arithmetic

C* allows arithmetic on scalar pointers to parallel variables; it is similar to the Standard C arithmetic on pointers to scalar variables. For example, given these declarations,

```
shape [65536]ShapeA;
int:ShapeA A1[40], *ptr1, *ptr2;
```

we can do the following:

```
ptr1 = &A1[7];
ptr2 = ptr1 + 2;
printf("%d\n", ptr2 - ptr1);
```

- The first statement sets **ptr1** equal to the address of the eighth element of the parallel array.
- The second statement puts the address of the tenth element of the array into **ptr2**.
- The **printf** statement prints 2, the result of subtracting **ptr1** from **ptr2**.

Note that these statements do not have to be within the body of a **with** statement, since the pointers are scalar variables.

As described above, we don't need to declare separate pointers into the array. We can also do this:

```
shape [65536]ShapeA;
int:ShapeA A1[40], p2, p3;
```

```
main()
{
    with (ShapeA) {
         p2 = *(A1 + 9);
         p3 = A1[9];    /* These two statements are
                           equivalent. */
    }
}
```

Each parallel variable element of both **p2** and **p3** is assigned the value of the corresponding parallel variable element of the tenth array element of **A1**.

Here is something we *can't* do:

```
shape [65536]ShapeA;
int:ShapeA A1[40], p2, p3, *ptr1, *ptr2;

ptr1 = &A1[7];
ptr2 = ptr1 + p2;      /* This is wrong */
p3 = *(ptr1 / p2);     /* This is wrong too */
```

It is illegal to perform arithmetic operations with a parallel variable and a scalar-to-parallel pointer as operands — except as discussed below.

7.3.4 Parallel Indexes into Parallel Arrays

C* lets you use a parallel index into a parallel array. The result is essentially a new parallel variable that contains elements from the existing parallel variables that make up the array. This is referred to as *parallel right indexing*.

Consider the data shown in Figure 28. A parallel array, **A**, and a parallel variable, **i**, have been allocated in a 1-dimensional shape, **S**.

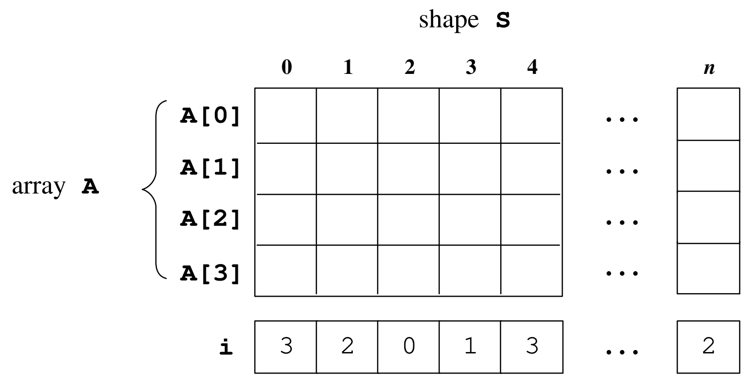

Figure 28. **A parallel array and an index parallel variable.**

C* allows the expression **`A[i]`**. The expression says: *In each position, use the value of* **`i`** *as an index for choosing a parallel variable element.* For example, in position [0] the value of **`i`** is 3; therefore, the element of parallel variable **`A[3]`** in that position is chosen. In position [1], the value of **`i`** is 2; therefore, the element of **`A[2]`** in that position is chosen. The result is a "jagged" parallel variable consisting of parallel variable elements taken from the different parallel variables that make up the parallel array. Figure 29 shows the results.

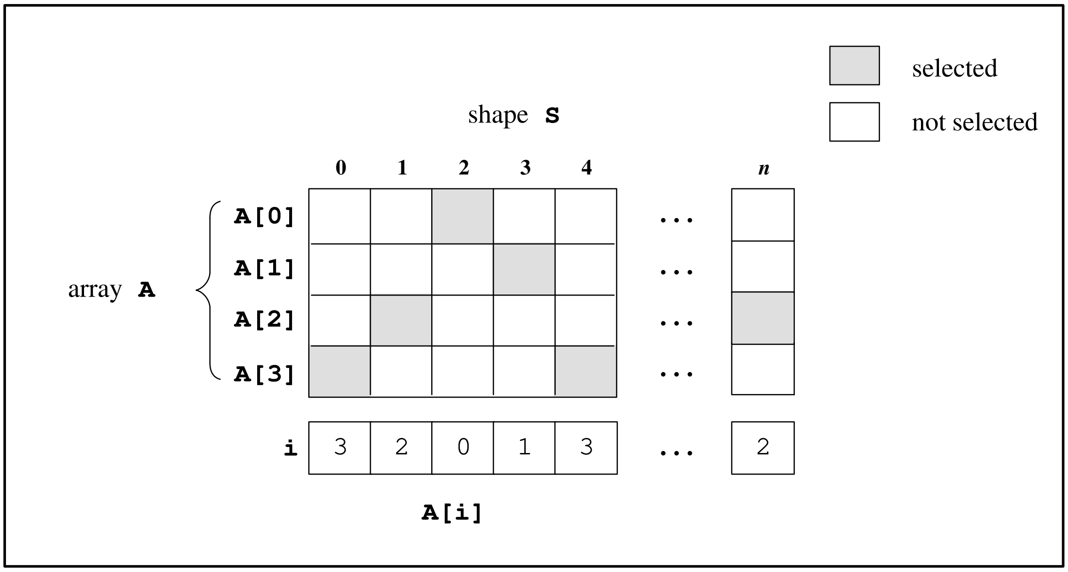

Figure 29. Indexing a parallel array by a parallel variable.

The values of the index parallel variable should be less than the number of parallel variables in the parallel array; otherwise, the index chooses an element outside the array, and the result is undefined. For example, if an element of **i** had a value of 17, the result would be undefined, because **i** is indexing an array of four parallel variables.

Adding a Parallel Variable to a Pointer to a Parallel Variable

The equivalence between arrays and pointers holds for parallel right indexing as well. In other words, **A[i]** is equivalent to ***(A+i)**. Note that ***(A+i)** is a legal example of an arithmetic operation involving a parallel variable and a scalar pointer to a parallel variable.

You can also subtract a parallel variable from a pointer to a parallel variable. For example, you might have a pointer point to the end of an array rather than the beginning. You could then subtract a parallel index from that pointer to choose parallel variable elements within the array. Once again, such an index must cause elements to be chosen from within an array; otherwise, the result is undefined.

Limitations

C* limits what you can do with parallel right indexing. You can dereference these expressions, but you cannot take their address. You can add a parallel variable to a pointer to a parallel variable, or subtract it from the pointer, but in each case the expression is legal only if it is immediately dereferenced. (The problem is that otherwise the expression would represent a parallel pointer to a parallel variable, and this kind of pointer does not exist in the language.) Thus, given these declarations:

```
shape [8192]S;
int:S A[4], i, p1, p2, *ptr;
int s1;
```

these statements are legal:

```
p1 = A[i];     /* In all cases, i should index parallel
                  variable elements within the array */
A[i]++;
p1 = *(A+i);
p1 = *(ptr - i);   /* Pointer should point into
                      an array */
```

and these statements are illegal:

```
s1 = &(A[i]);      /* Can't take the address */
s1 = (A+i);        /* Creates invalid pointer type */
p1 = ptr + p2;     /* Can't perform an operation without
                      dereferencing */
p1 = *(ptr / i);   /* Can only add or subtract */
```