Chapter 10

The C* Programming Language

C* is an extension of the C programming language designed to support data parallel programming.

The C* language is based on the standard version of C specified by the American National Standards Institute (ANSI). C programmers will find most aspects of C* code familiar to them. C language constructs such as data types, operators, structures, pointers, and functions are all maintained in C*; new features of Standard C such as function prototyping are also supported. C* extends C with a small set of new features that allow programmers to use the Connection Machine system efficiently.

C* is well suited for applications that require dynamic behavior, since it allows the size and shape of parallel data to be determined at run time. In addition, it provides programmers with all the standard benefits of C, such as block structure, access to low-level facilities, string manipulation, and recursion. C* also provides a straightforward method for calling CM Fortran subroutines from a C* program.

10.1 Structuring Parallel Data

In C*, data is allocated on the processing nodes only when it is tagged with a *shape*. A shape is a way of logically configuring parallel data. C* includes a new construct called *left indexing* that is used in declaring a shape. The left index specifies the number of dimensions (or *axes*) in the shape and the number of *positions* along each dimension. Positions correspond to processors (or virtual processors). For example,

```
shape [25][51]s;
```


Copyright © 1993 Thinking Machines Corporation

declares a shape **s** that is laid out as a 25 x 51 grid on the processing nodes.

This shape is considered to be *fully specified*, since the number of dimensions and positions are provided at compile time. Shapes may also be partially specified or fully unspecified. C* lets the programmer dynamically allocate and specify shapes, thus providing flexibility in the way they can be used.

Once a shape has been fully specified, one can declare *parallel variables* of that shape. Parallel variables have both a Standard C data type and a shape. For example, the code

```
shape [16384]t;
int:t parallel_int1, parallel_int2;
float:t parallel_float1;
```

declares three parallel variables of shape **t**; each consists of 16384 *elements*, laid out along one dimension. Parallel variables interact most efficiently when they are of the same shape. In addition to the above method, parallel variables can also be allocated dynamically.

C* also provides parallel versions of arrays and structures. For example, the code

```
shape [16384]t;
int:t parray[16];
```

declares a parallel array, **parray**, which consists of 16 parallel **int**s of shape **t**. The code

```
shape [16384]t;
struct scalar_struct {
    int a;
    float b;
};
struct scalar_struct:t pstruct;
```

declares a parallel structure, **pstruct**, that consists of the Standard C structure **scalar_struct** replicated in each of the 16384 positions of shape **t**.

C* includes pointers to both shapes and parallel variables. As in Standard C, C* pointers are fast and powerful.

November 1993
Copyright © 1993 Thinking Machines Corporation

10.2 Computing in Parallel

Parallel Use of Standard C Operators

C* extends the use of Standard C operators, through overloading, to apply to parallel data as well as scalar data. For example, if **p1**, **p2**, and **p3** are all parallel variables of the same shape, the statement

```
p3 = p2 + p1;
```

performs a separate addition of the values of **p1** and **p2** *in each position of the shape* and assigns the result to the element of **p3** in that position. The additions take place in parallel. If **p1** or **p2** were not a parallel variable, it would first be promoted to parallel, with its value replicated in every element. Note that this line of code looks exactly like Standard C; the result differs, however, depending on whether the variables are parallel or scalar.

The with and where Statements

C* adds new statements to Standard C that allow operations on parallel data.

The **with** statement selects a current shape. In general, parallel variables must be of the current shape before parallel operations can take place on them. For example, code like the following is actually required to perform a parallel addition like the one shown above:

```
shape [16384]t;
int:t p1, p2, p3;

with (t)
    p3 = p2 + p1;
```

C* also adds a **where** statement to restrict the set of positions on which operations are to take place; the positions to be operated on are called *active*. Selecting the active positions of a shape is known as *setting the context*. The **where** statement in the following example ensures that division by 0 is not attempted:

```
with (t)
    where (p1 != 0)
         p3 = p2 / p1;
```

Serial code always executes, no matter what the context.

Copyright © 1993 Thinking Machines Corporation

Programs may contain nested **where** statements; these cumulatively shrink the set of active positions. The context is passed into functions called within the scope of a **where** statement and is correctly reestablished when returning to an outer level as a result of a **break**, **continue**, **goto**, or **return** statement. Note that the context does not affect the flow of control of a program. One can still use Standard C statements such as **if** and **while** to manipulate flow of control.

C* extends the Standard C **else** statement for use in conjunction with the **where** statement; using **else** after a **where** reverses the set of active positions. The new **everywhere** statement makes all positions active.

New Operators

C* adds a few new operators to Standard C. For example, the **<?** and **>?** operators are available to obtain the minimum and maximum of two variables (either scalar or parallel). The corresponding compound assignment operators **<?=** and **>?=** are also included. The operator **%%** provides a true modulus operation (as compared to the remainder operator **%**).

Parallel Functions

Functions in C* can pass and return parallel variables and shapes. If it is not known what the current shape will be when the function is called, you can use the new keyword **current** in place of a specific shape name within the function declaration; **current** always means the current shape.

A useful feature of C* is *overloading* of functions. C* allows you to declare more than one version of a function with the same name — for example, one version for scalar data and another for parallel data. The compiler automatically chooses the right version.

November 1993
Copyright © 1993 Thinking Machines Corporation

10.3 Communicating in Parallel

C* provides two methods of parallel communication: as part of the syntax of the language and via an extensive library of functions. Both allow communication in regular patterns within shapes and in irregular patterns both within and between shapes.

Regular Communication

C* uses the intrinsic function **pcoord** to provide a self-index for a parallel variable along a specified axis of its shape. For example, if **p1** is of a one-dimensional shape with 16384 positions (and the shape is current), **pcoord** initializes **p1** as shown in Figure 18.

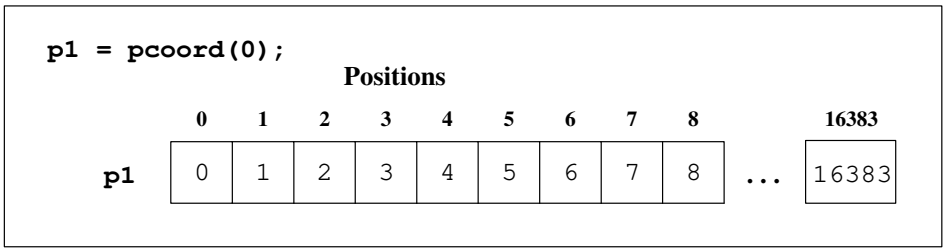

Figure 18. The use of pcoord with a one-dimensional shape.

The **pcoord** function is typically used to provide regular communication — called *grid communication* in C* — along the axes of a shape. For example, the following code sends values of **source** to the elements of **dest** that are one coordinate higher along axis 0:

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

In the common case where **pcoord** is called within a left index expression, and the argument to **pcoord** specifies the axis indexed by the left index, C* allows a shortcut: the call to **pcoord** can be replaced by a period. Thus, for a two-dimensional shape, the following provides grid communication along both axis 0 and axis 1:

```
[.+1][.-2]dest = source;          (A chess knight's move)
```


Copyright © 1993 Thinking Machines Corporation

Wrapping from one end of an axis to the other is provided by a Standard C* programming idiom that involves the use of **pcoord** along with the new modulus operator **%%** and the **dimof** intrinsic function, which returns the number of positions along an axis of a shape.

Library functions are also available to perform grid communication. For example, the **to_grid_dim** and **to_grid** functions can be used in place of the statements above.

Irregular Communication

C* uses the concept of left indexing to provide communication between different shapes, as well as within-shape communication that does not necessarily occur in regular patterns.

A left index can be applied to a parallel variable. If the index itself is a parallel variable, the result is a rearrangement of the values of the parallel variable being indexed, based on the values in the index. If the index is of one shape and the parallel variable being indexed is of another shape, the result is a remapping of the parallel variable into the shape of the index. Thus, in the assignment

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

the parallel variable **dest** gets values from **source**; the values in **index** indicate which element of **source** is to go to which element of **dest**. The variables **dest** and **index** must be of the current shape; **source** can be of any shape. This is known as a *get operation*. Putting the index variable on the left-hand side specifies a *send operation*. Sends are roughly twice as fast as gets. The operations can also be performed with the **send** and **get** functions in the C* communication library.

10.4 Transforming Parallel Data

C* provides operators and library functions that enable programmers to easily perform common transformations of parallel data.

C* overloads the meaning of several Standard C compound assignment operators to provide a succinct way of expressing global reductions of parallel data. For example, **+=**, when applied as a unary operator to a parallel variable, sums the

November 1993
Copyright © 1993 Thinking Machines Corporation

values of all active elements of the parallel variable. The resulting value can be treated the same way as the result of a serial operation. Similarly, the **|=** operator performs a bitwise OR of all elements of a parallel variable. The **reduce** and **global** library functions provide similar capabilities for various operations.

The C* communication library contains many functions that perform other 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 ranking can be used to rearrange the elements into sorted order.

Copyright © 1993 Thinking Machines Corporation