Chapter 2

Using C*

This chapter presents a simple C* program that illustrates some basic features of the language. At this point we are not going to describe these features in detail; the purpose is simply to give a feel for what C* is like. After the program has been presented, we briefly describe how to compile and execute it.

The program sets up three parallel variables, each of which consists of 65,536 individual data points called *elements*. It then assigns integer constants to each element of these parallel variables and performs simple arithmetic on them.

```
#include <stdio.h>

/*
============================================================
 *  1. Declare the shape and the variables
 */

shape [2][32768]ShapeA;
int:ShapeA p1, p2, p3;
int sum = 0;
main()
{

/* ============================================================
 * 2. Select the shape
 */
    with (ShapeA){

/*  ============================================================
 *  3. Assign values to the parallel variables
 */
```


Copyright © 1990–1995 Thinking Machines Corporation

```
        p1 = 1;
        p2 = 2;
/*  ============================================================
 *  4. Add them
 */

        p3 = p1 + p2;

/*  ============================================================
 *  5. Print the sum in one element of p3
 */

        printf ("The sum in one element is %d.\n", [0][1]p3);

/*  ============================================================
 *  6. Calculate and print the sum in all elements of p3
 */

        sum += p3;
        printf ("The sum in all elements is %d.\n", sum);
    }
}
```

Its output is

```
The sum in one element is 3.
The sum in all elements is 196608.
```

Before we go through the program, notice the file extension, **`.cs`**, in the program's name. C* source files must have this **`.cs`** extension.

2.1 Step 1: Declaring Shapes and Parallel Variables

2.1.1 Shapes

The initial step in dealing with parallel data in a C* program is to declare its *shape* — that is, the way the data is to be organized. In Step 1 of **`add.cs`**, the line

```
shape [2][32768]ShapeA;
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

declares a shape called **ShapeA**. **ShapeA** consists of 65,536 *positions*, as shown in Figure 1.

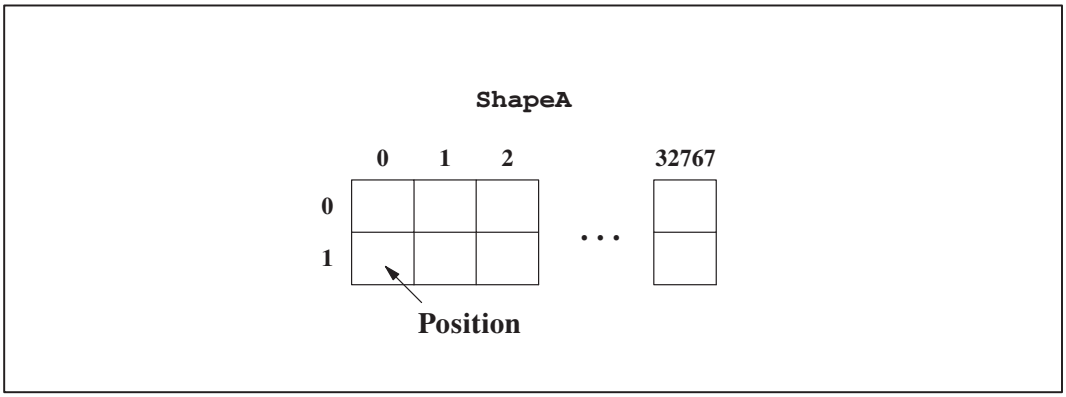

Figure 1. The shape ShapeA.

ShapeA has two dimensions; you can also declare shapes with other numbers of dimensions. The choice of two dimensions here is arbitrary. The appropriate shape depends on the data with which your program will be dealing.

2.1.2 Parallel Variables

Once you have declared a shape, you can declare *parallel variables* of that shape. In **add.cs**, the line

```
int:ShapeA p1, p2, p3;
```

declares three parallel variables: **p1**, **p2**, and **p3**. They are of type **int** and of shape **ShapeA**. This declaration means that each parallel variable is laid out using **ShapeA** as a template, with memory allocated for one *element* of the variable in each of the 65,536 positions specified by **ShapeA**. Figure 2 shows the three parallel variables of shape **ShapeA**.

Copyright © 1990–1995 Thinking Machines Corporation

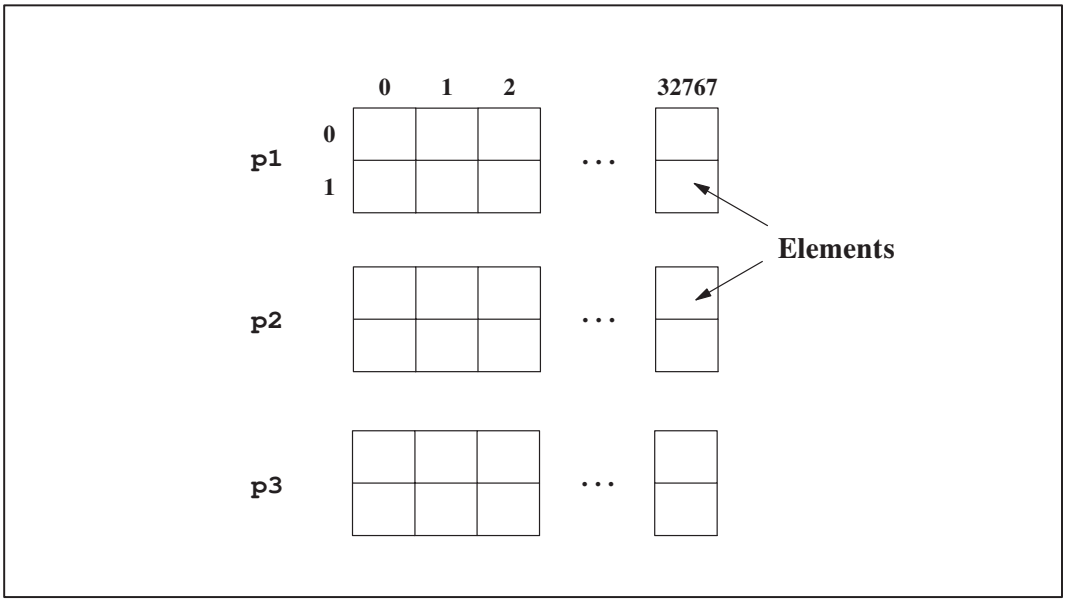

Figure 2. Three parallel variables of shape `ShapeA`.

With C*, you can perform operations on all elements of a parallel variable at the same time, on a subset of these elements, or on an individual element.

2.1.3 Scalar Variables

In Step 1, the line

```
int sum = 0;
```

is Standard C code that declares and initializes a C variable. These C variables are called *scalar* in this guide to distinguish them from C* parallel variables. In the CM-5 implementation of C* (when the program is not compiled with the **`-node`** option), memory for Standard C variables is allocated on the partition manager.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

2.2 Step 2: Selecting a Shape

In **add.cs**, the line

```
with (ShapeA)        /* Step 2 */
```

tells C* to use **ShapeA** in executing the code that follows. In other words, the **with** statement specifies that only the 65,536 positions defined by **ShapeA** are *active*. In C* terminology, this makes **ShapeA** the *current shape*. With some exceptions, the code following the **with** statement can operate only on parallel variables that are of the current shape, and a program can execute most parallel code only within the body of a **with** statement.

2.3 Step 3: Assigning Values to Parallel Variables

Once a shape has been selected to be the current shape, the program can include statements that perform operations on parallel variables of that shape. Step 3 in **add.cs** is a simple example of this:

```
p1 = 1;      /* Step 3 */
p2 = 2;
```

The first statement assigns the constant 1 to each element of **p1**; the second statement assigns 2 to each element of **p2**. After these two statements have been executed, **p1** and **p2** are initialized as shown in Figure 3.

Copyright © 1990–1995 Thinking Machines Corporation

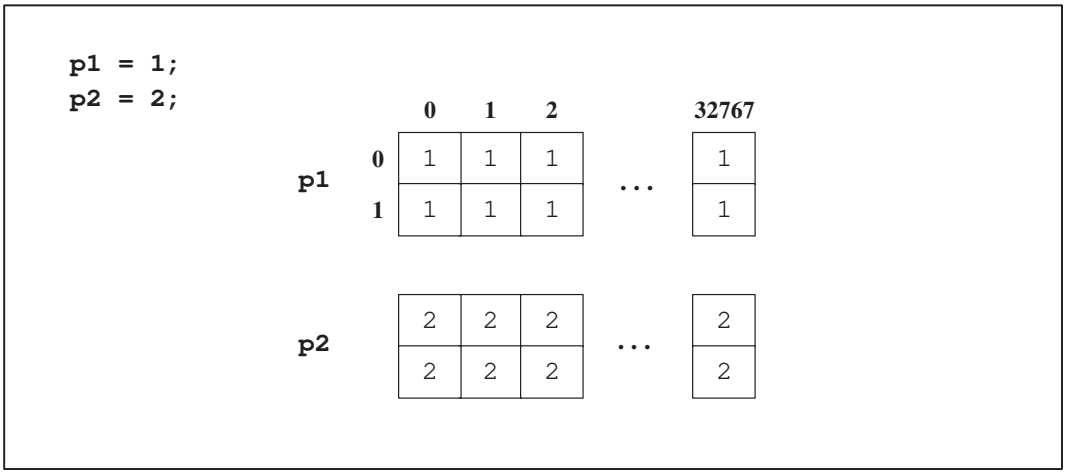

Figure 3. Initialized parallel variables.

Note that the statements in Step 3 look like simple C assignment statements, but the results are different (although probably what you would expect) because **p1** and **p2** are parallel variables. Instead of one constant being assigned to one scalar variable, one constant is assigned simultaneously to each element of a parallel variable.

2.4 Step 4: Performing Computations Using Parallel Variables

Step 4 in **add.cs** is a simple addition of parallel variables:

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

In this statement, each element of **p1** is added to the element of **p2** that is in the same position, and the result is placed in the element of **p3** that is also in the same position. Figure 4 shows the result of this statement.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

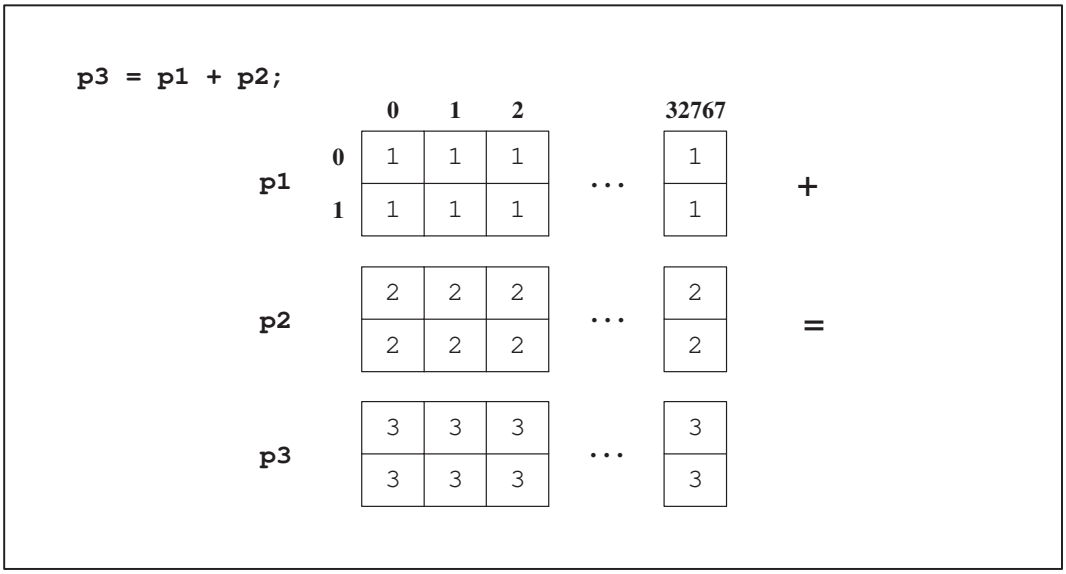

Figure 4. Addition of parallel variables.

Like C* assignment statements, C* parallel arithmetic operators look the same as the standard C arithmetic operators, but work differently because they use parallel variables.

2.5 Step 5: Choosing an Individual Element of a Parallel Variable

In Step 5 of **`add.cs`** we print the sum in one element of **`p3`**. Step 5 looks like a standard C **`printf`** statement, except for the variable whose value is to be printed:

```
[0][1]p3
```

`[0][1]` specifies an individual element of the parallel variable **`p3`**. Elements are numbered starting with 0, and you must include subscripts for each dimension of the parallel variable. Thus, **`[0][1]p3`** specifies the element in row 0, column 1 of **`p3`**, and the **`printf`** statement prints the value contained in this element.

Copyright © 1990–1995 Thinking Machines Corporation

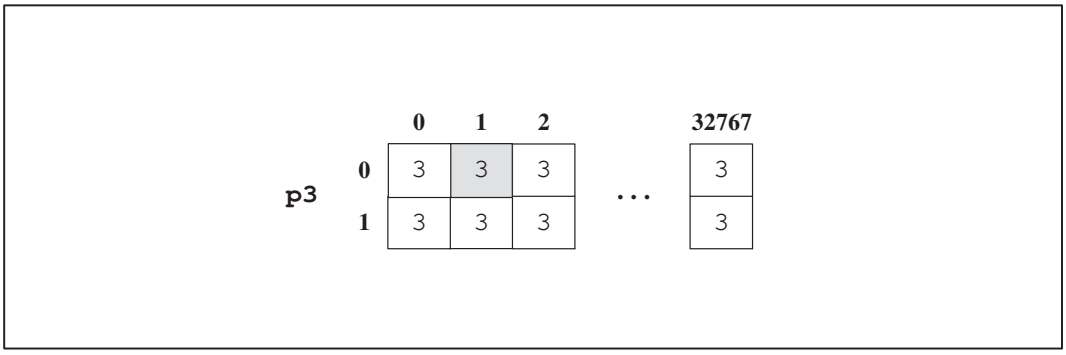

Figure 5. Element [0][1] of `p3`.

Note that this **`printf`** statement would be incorrect:

```
printf ("The sum in one element is %d.\n", p3); /* wrong */
```

Different elements of **`p3`** could have different values (even though they are all the same in the sample program), so **`printf`** would not know which one to print.

2.6 Step 6: Performing a Reduction Assignment of a Parallel Variable

So far, **`add.cs`** has demonstrated assignments to parallel variables and addition of parallel variables. This line in the program:

```
sum += p3;  /* Step 6 */
```

is an example of a *reduction assignment* of a parallel variable. In a reduction assignment, the variable on the right-hand side must be parallel, and the variable on the left-hand side must be scalar. The **`+=`** reduction assignment operator sums the values in all elements of the parallel variable (in this case, **`p3`**) and adds this sum to the value in the scalar variable (in this case, **`sum`**); see Figure 6. (Note that the value of the scalar variable on the left-hand side is included in the addition; that is why **`add.cs`** initializes **`sum`** to 0 in Step 1.)

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

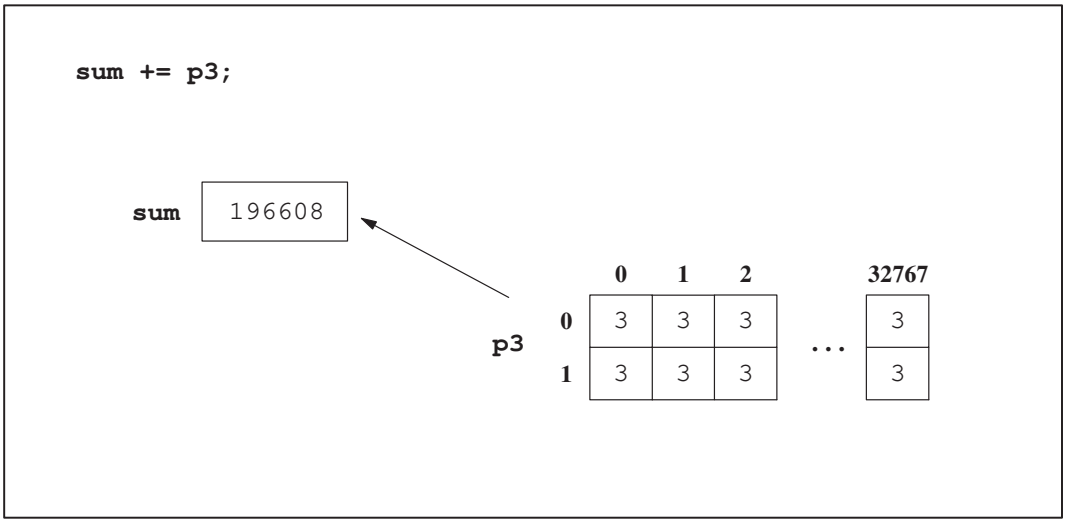

Figure 6. The reduction assignment of parallel variable `p3`.

The final statement of the program simply prints in standard C fashion the value contained in **`sum`**.

Note the first closing brace, on the line after the final **`printf`** statement. This brace ends the block of statements within the scope of the **`with`** statement in Step 2.

2.7 Compiling and Executing the Program

2.7.1 Compiling

You compile a C* program using the command **`cs`** on a computer on which the C* compiler is installed. To compile the program **`add.cs`**, type

```
% cs add.cs
```

Use the **`-cm5`** or **`-cmsim`** option to specify the hardware for which the program is to be compiled (there is also a site-specific default). On a CM-5, specify the **`-sparc`** or **`-vu`** option to specify whether you are compiling to run on the processing nodes or vector units.

As with the C compiler command **`cc`**, this command produces an executable load module, placed by default in the file **`a.out`**.

Copyright © 1990–1995 Thinking Machines Corporation

2.7.2 Executing

On a CM-5, you can execute the resulting load module from a partition manager as you would any program or UNIX command. For example,

```
% a.out
```

For more information on how to compile and execute a C* program on a CM-5, see the *CM-5 C* User's Guide*.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation