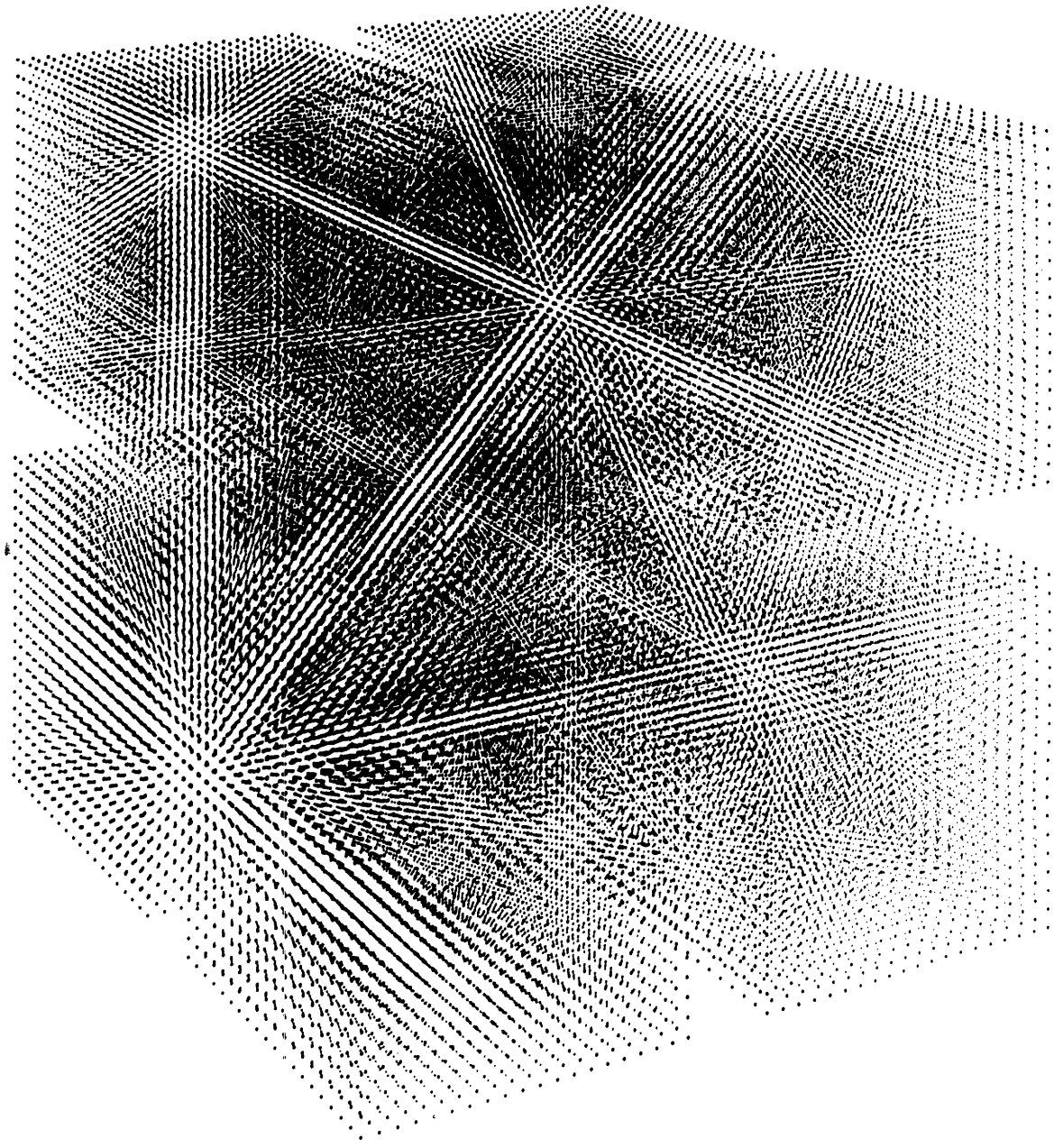
Thinking Machines Corporation

Getting Started in C*



The
Connection Machine
System

Getting Started in C*

May 1993

Thinking Machines Corporation
Cambridge, Massachusetts

Second edition, May 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

Contents

Chapter 1 What Is C*? 1

1.1 C* and C 1

1.2 C* Implementations 2

1.3 Developing a C* Program 2

Chapter 2 A Simple Program 3

2.1 Step 1: Declaring Shapes and Parallel Variables 5

Shapes 5

Parallel Variables 5

Scalar Variables 6

2.2 Step 2: Selecting a Shape 7

2.3 Step 3: Assigning Values to Parallel Variables 7

2.4 Step 4: Performing Computations Using Parallel Variables 8

2.5 Step 5: Choosing an Individual Element of a Parallel Variable 9

2.6 Step 6: Performing a Reduction Assignment of a Parallel Variable 10

2.7 Compiling and Executing the Program 11

Compiling 11

Executing 12

Chapter 3 Shapes and Parallel Variables 13

3.1 Shapes 13

Declaring Shapes 14

3.2 Parallel Variables 15

Declaring Parallel Variables 15

3.3 Other Kinds of Parallel Data 16

Parallel Structures 17

Parallel Arrays 18

3.4 Pointers to Shapes and Parallel Variables 18

Pointers to Shapes 19

Pointers to Parallel Variables 19
3.5 Choosing a Shape: The **with** Statement 20
3.6 Setting the Context: The **where** Statement 21
The everywhere Statement 22
3.7 Dynamically Allocating Shapes and Parallel Variables 22
The **allocate_shape** and **deallocate_shape** Functions 23
The **palloc** and **pfree** Functions 24

Chapter 4 Parallel Operations 25

4.1 Standard C Operators 25
Unary Operators 25
Binary Operators with a Scalar Operand and a Parallel Operand 26
Assignment with a Scalar LHS and a Parallel RHS 26
Binary Operators with Two Parallel Operands 27
The Conditional Expression 27
4.2 Reduction Operators 28
4.3 Functions 29
Passing by Value and Passing by Reference 30
Overloading Functions 30

Chapter 5 Communication 33

5.1 Kinds of Communication 33
5.2 General Communication 34
When There Are Inactive Positions 36
5.3 Grid Communication 37
The **pcoord** Function 37
The **pcoord** Function and Left Indexing 38
Grid Communication without Wrapping 39
Grid Communication with Wrapping 40

Chapter 6 Sample Programs 43

6.1 Julia 43
CM-200 Version 44
CM-5 Version 47
Output 49

6.2 Prime Number Sieve 50
6.3 Shuffle 52
CM-200 Version 52
CM-5 Version 55
Output 57

Chapter 7 Performance Hints **59**

Index 61

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Telephone:	(617) 234-4000

Chapter 1

What Is C*?

C* (pronounced "sea-star") is an extension of the C programming language designed to help users program massively parallel distributed-memory computers.

1.1 C* and C

The C* language is based on the standard version of C specified by the American National Standards Institute. (We assume in this guide that you are familiar with C.) 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 to aid in writing programs for massively parallel computers. For example:

- C* provides a **`shape`** keyword that lets you describe the size and shape of parallel data. You can then declare *parallel* variables by tagging them with a shape.
- C* lets you operate on parallel variables by providing a few new operators, and new meanings for standard C operators.
- C* adds a **`with`** statement that lets you choose the shape to be used for parallel operations, and a **`where`** statement to restrict operations to certain data points of the shape.
- C* extends Standard C structures, pointers, and functions so that they work with parallel data.

- C* provides ways for parallel variables to interact with other parallel variables, even if the parallel variables are of different shapes.

1.2 C* Implementations

This guide refers to two implementations of the C* language:

- *CM-200 C** — Use this implementation to run your program on a CM-200, CM-200a, CM-2, or CM-2a Connection Machine system.
- *CM-5 C** — Use this implementation to run your program on all models of the CM-5 Connection Machine system.

1.3 Developing a C* Program

C* uses its own compiler, run-time libraries, and header files, as well as some Standard C libraries. Files containing C* source code must have the suffix `.cs`.

Within your C* program, you can include calls to library routines such as those in the CMFS library. You can also call subroutines written in CM Fortran.

On a CM-2, CM-2a, CM-200, or CM-200a, you execute the program by attaching to the CM via the **cmattach** command, or by submitting the executable load module as a batch request to a queue in the NQS batch system.

On a CM-5, you execute the program just as you would any UNIX executable load model; or you can submit it as a batch request to NQS.

You can use Prism, the CM's graphical programming environment, to help you develop, debug, and analyze the performance of your C* program.

Chapter 2

A Simple Program

This chapter uses a simple C* program to show some basic features of the language. Subsequent chapters discuss these features in more detail. See Chapter 6 for examples of more advanced C* programs.

The program, which we'll call **add.cs**, declares three parallel variables of the same shape; each of the parallel variables 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 two of the parallel variables.
 */
      p1 = 1;
      p2 = 2;

/*
 * ============================================================
 *  4. Add them and assign the result to the third parallel
 *     variable.
 */

      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);
   }
}
```

The output of **add.cs** is shown below:

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

2.1 Step 1: Declaring Shapes and Parallel Variables

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;
```

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

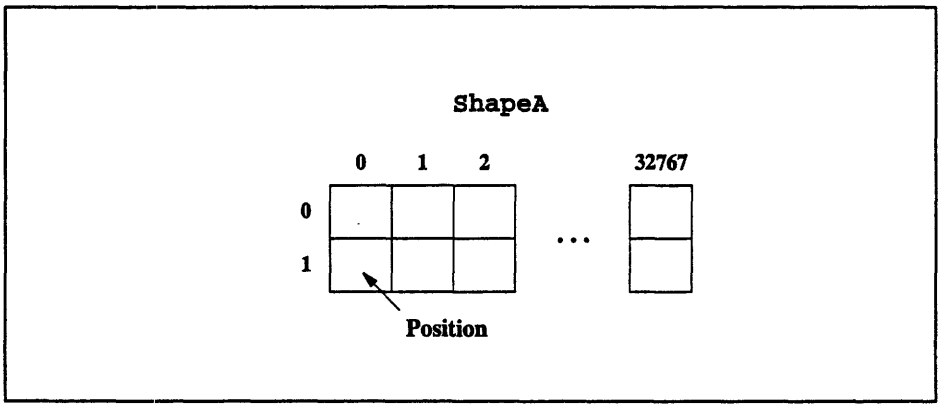


Figure 1. The shape ShapeA.

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.

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`**.

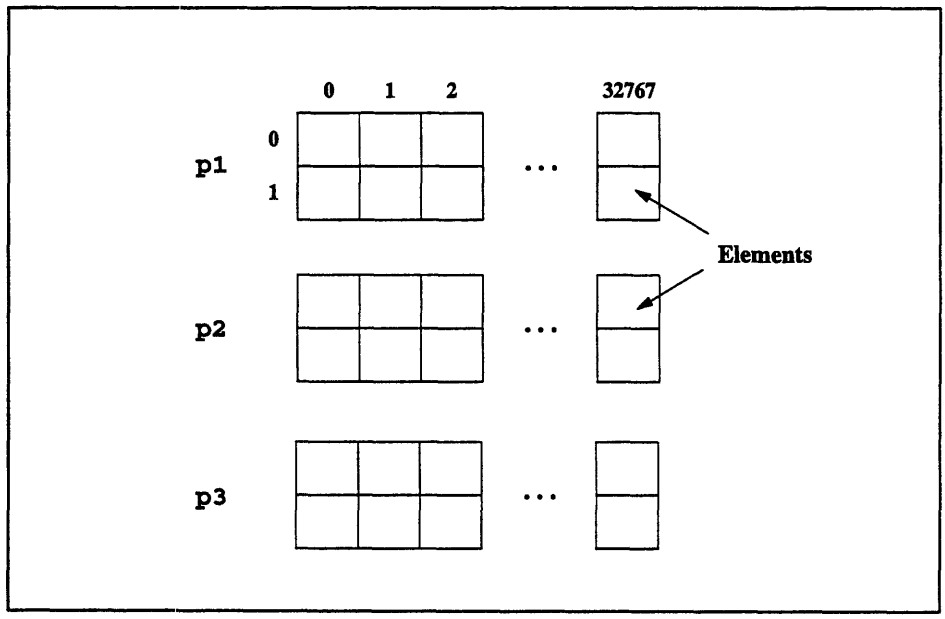


Figure 2. Three parallel variables of shape `ShapeA`.

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.

Scalar Variables

In Step 1, the line

```
int sum = 0;
```

is Standard C code that declares and initializes a Standard C variable. These C variables are called *scalar* to distinguish them from C* parallel variables. In the CM-200 and CM-5 implementations of C*, memory for Standard C variables is allocated on the serial computer (for example, front end or partition manager) rather than on the CM processors.

2.2 Step 2: Selecting a Shape

In Step 2 of **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. 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 current, 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.

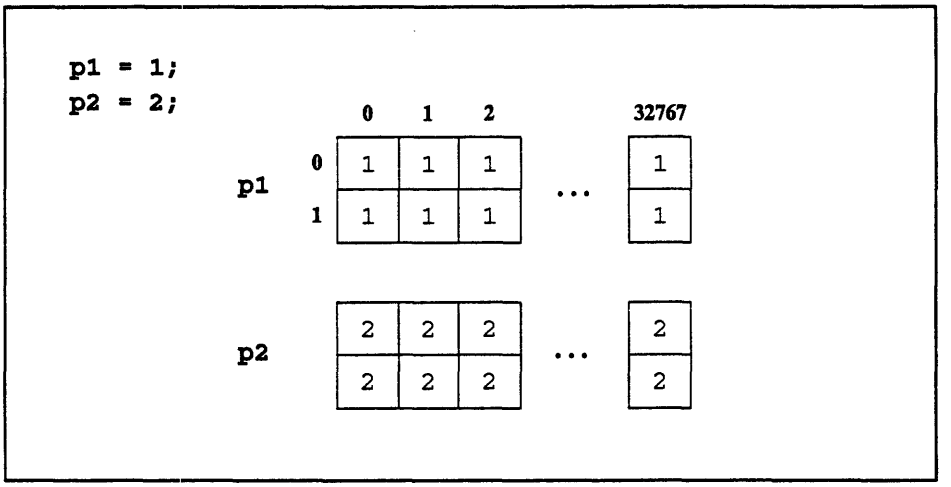


Figure 3. Initialized parallel variables.

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.

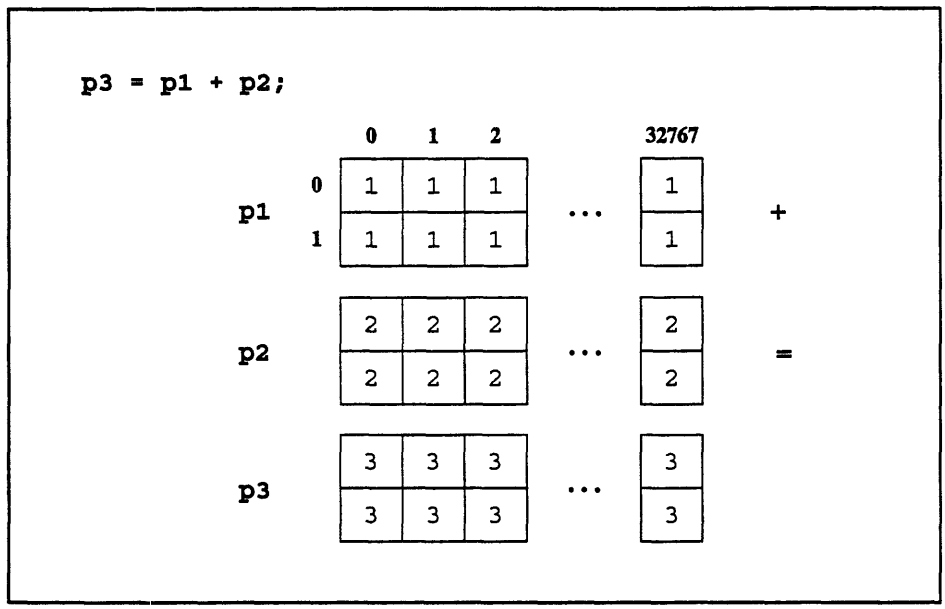


Figure 4. Addition of parallel variables.

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 expression 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.

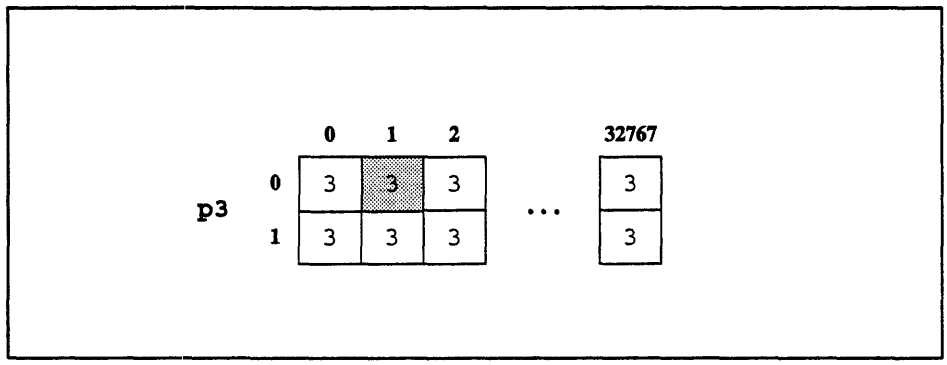


Figure 5. Element [0][1] of `p3`.

Note that this `printf` statement would be incorrect:

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

The `printf` statement is looking for one value to print. 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 */
```

demonstrates 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 adds 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.)

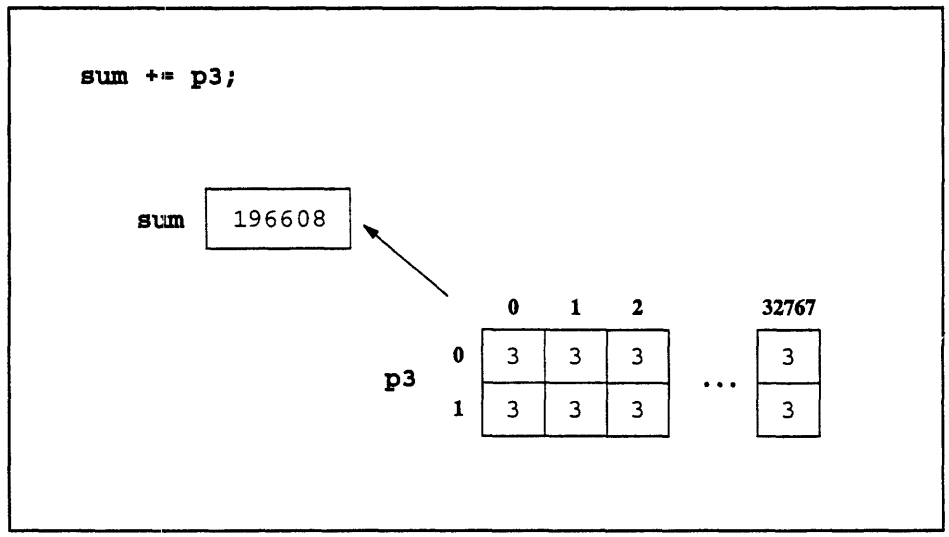


Figure 6. The reduction assignment of parallel variable `p3`.

The final statement of the program prints in Standard C fashion the value contained in `sum`.

2.7 Compiling and Executing the Program

Compiling

You compile a C* program using the compiler command `cs`. To compile the program `add.cs`, issue this command:

```
% cs add.cs
```

As with the C compiler command `cc`, this command produces an executable load module, placed by default in the file `a.out`. You can use the `-o` option to specify a different file.

The `cs` command has various other options you can specify, many of which, like `-o`, are the same as standard `cc` options.

Executing

To execute the resulting load module, simply type its name at the UNIX prompt of a front end or partition manager, just as you would any executable program:

```
% a.out
```

For complete information on compiling and executing C* programs, see the *C* User's Guide* for the C* implementation you will be using.

Chapter 3

Shapes and Parallel Variables

As the sample program in Chapter 2 shows, shapes and parallel variables are central to the way C* extends C to support data parallel programming. This chapter describes shapes and parallel variables in more detail.

3.1 Shapes

A shape is a template for parallel data. In C*, you must specify the shape of the data before you can define data of that shape. A shape is specified by:

- How many dimensions it has. This is referred to as the shape's *rank.*
- The number of *positions* in each of its dimensions. A position is an area that can contain values of parallel data.

The total number of positions in a shape is the product of the number of positions in each of its dimensions. For example, an 8-by-4 shape has 32 positions.*

Typically, the choice of a shape reflects the natural organization of the data. For example, a graphics program might use a shape representing the 2-dimensional images that the program is to process. If your program works on different data sets, you can have a different shape for each one.

---

* In the CM-200 implementation of C*, the number of positions in each dimension of a shape must be a power of 2, and the total number of positions in the shape must be some multiple of the number of physical processors in the section of the CM that the C* program is using.

Declaring Shapes

Use the new C* keyword **shape** to declare a shape, as in this example:

```
shape [16384]employees;
```

This statement declares a shape called **employees**. It has one dimension (a rank of 1) and 16384 positions. A dimension is also referred to as an *axis*.

Note the position of the brackets, to the left of the shape name. *Left indexing* is an important new concept in C*. A shape can have multiple axes; specify each in brackets to the left of the shape name. For example, here is a 2-dimensional shape:

```
shape [256][512]image;
```

The left-most axis is referred to as axis 0; the next axis to the right is axis 1, and so on.

A program can include many shapes. You can use a single shape statement to declare multiple shapes. For example:

```
shape [16384]employees, [256][512]image;
```

The shapes we have looked at so far have been *fully specified.* You don't need to specify a shape fully when you declare it. For example,

```
shape data;
```

declares a shape called **data**, without specifying its rank or the number of positions in it. You can also specify the shape's rank without specifying its size. For example,

```
shape []data;
```

declares a 1-dimensional shape of unspecified size.

Declaring a shape that is not fully specified is useful if, for example, the size of the shape is to come from user input. However, a shape's rank and size must be fully specified before you can use it. Section 3.7 describes how to specify a shape fully.

3.2 Parallel Variables

As Chapter 2 explained, a parallel variable is similar to a Standard C variable, except that it has a shape in addition to its type and storage class. The shape defines how many *elements* of a parallel variable exist, and how they are organized. Each element occupies one position within the shape and contains a single value. If a shape has 16384 positions, a parallel variable of that shape has 16384 elements, one for each position.

Each element of a parallel variable can be thought of as a single scalar variable. But a C* program can also carry out operations on all elements (or any subset of elements) of a parallel variable at the same time.

Declaring Parallel Variables

Before declaring a parallel variable, you must fully specify the shape that the parallel variable is to take. For example, once you have declared shape **employees** as in the example above, you can declare a parallel variable of that shape:

```
unsigned int:employees employee_id;
```

This statement declares a parallel variable named **employee_id**; it is an **unsigned int** of shape **employees**. Figure 7 shows this parallel variable.

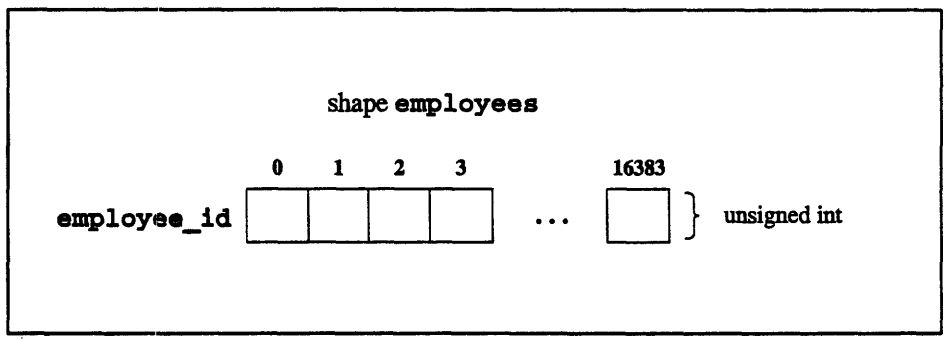


Figure 7. A parallel variable of shape employees.

Once the parallel variable has been declared, you can use left indexing to specify an individual element of it. For example, **[2]employee_id** refers to the third

element of **employee_id** in Figure 7. **[2]** is referred to as the *coordinate* for this element.

You can declare many parallel variables of the same shape. If they are of the same type, you can declare them in the same statement. For example:

```
unsigned int:employees employee_id, age, salary;
```

All three parallel variables are of shape **employees**. As shown in Figure 8, elements of different parallel variables that are in the same position of a shape are referred to as *corresponding elements.*

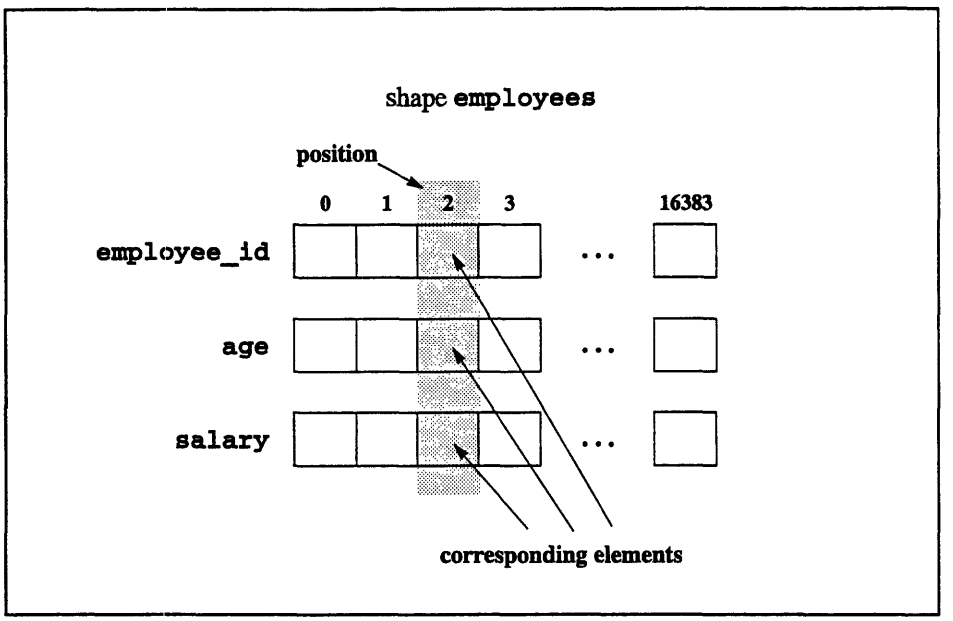


Figure 8. Three parallel variables of shape employees.

3.3 Other Kinds of Parallel Data

In addition to parallel variables, C* provides parallel versions of C aggregate types. In this section, we look at parallel structures and parallel arrays.

Parallel Structures

You can declare an entire structure as a parallel variable. For example, if you have declared this shape and structure:

```
shape [16384]employees;
struct date {
   int month;
   int day;
   int year;
   };
```

you can declare a parallel structure as follows:

```
struct date:employees birthday;
```

birthday is of type **struct date** and of shape **employees**. It is shown in Figure 9.

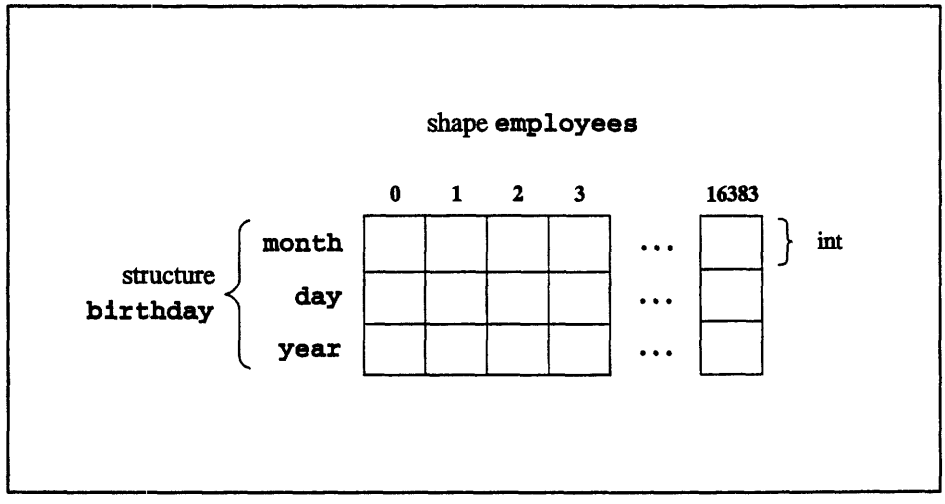


Figure 9. A parallel structure of shape employees.

Each element of the parallel structure contains a scalar structure, which in turn will contain the birthday of an employee.

Accessing a member of a parallel structure works the same way as accessing a member of a scalar structure. For example, **birthday.day** specifies all elements of structure member **day** in the parallel structure **birthday**.

C* does not allow shapes, pointers, or parallel variables inside a parallel structure.

Parallel Arrays

You can also declare an array of parallel variables. For example,

```
shape [16384]employees;
int:employees ratings[3];
```

declares an array of three parallel variables of shape **employees**, as shown in Figure 10.

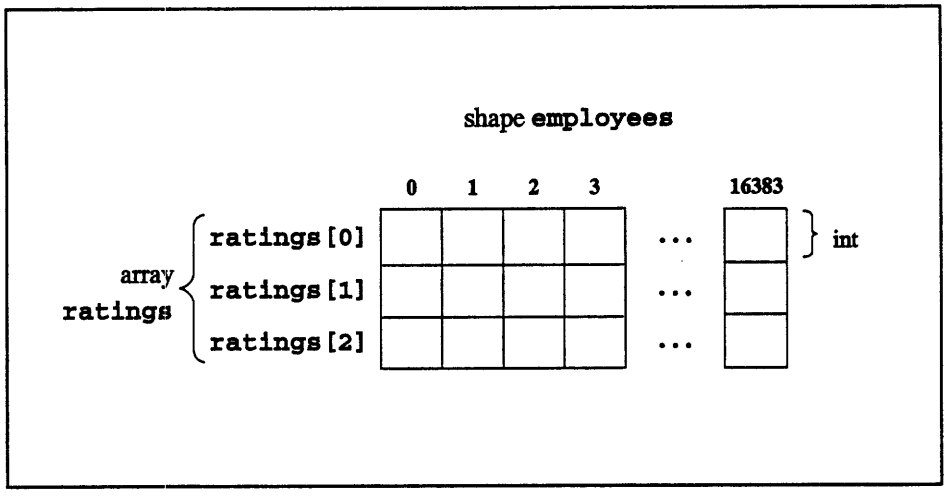


Figure 10. A parallel array of shape employees.

3.4 Pointers to Shapes and Parallel Variables

In addition to Standard C pointers, C* provides pointers to shapes and parallel variables. As with C pointers, you can use the pointer in place of the object being pointed to. In functions, you can use pointers to pass by reference instead of by value.

Pointers to Shapes

This statement declares the scalar variable **shape_ptr** to be a pointer to a shape:

```
shape *shape_ptr;
```

And this statement makes **shape_ptr** point to shape **ShapeA**:

```
shape_ptr = &ShapeA;
```

A dereferenced pointer to a shape can be used as a *shape-valued expression*; that is, you can use it wherever you would use a shape name. For example,

```
with (*shape_ptr)
```

makes **ShapeA** the current shape.

Pointers to Parallel Variables

The statement:

```
int:ShapeA *pvar_ptr;
```

declares a scalar pointer **pvar_ptr** that points to a parallel **int** of shape **ShapeA**. If **p1** is a parallel variable of shape **ShapeA**, then

```
pvar_ptr = &p1;
```

makes **pvar_ptr** a pointer to **p1**. You can then reference **p1** via the pointer **pvar_ptr**.

You can declare a pointer to a parallel variable of a shape that is not fully specified, even though you cannot declare a parallel variable of that shape. For example:

```
shape data;
int:data *data_ptr;
```

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

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

declares a parallel array of 40 **ints** of shape **ShapeA**, and **A1** points to the first element of the array — that is, its type is a scalar pointer to a parallel **int** of shape **ShapeA**.

3.5 Choosing a Shape: The with Statement

Before you can carry out most operations on parallel variables, they must be of the *current shape*. You designate the current shape by using the new C* **with** statement. For example:

```
shape [16384]employees;
unsigned int:employees employee_id, age, salary;

main()
{
   with (employees)
      /* Operations on parallel variables of shape
         employees go here. */
}
```

Most operations on parallel variables of shape **employees** can occur only within the scope of the **with** statement. We will discuss what constitutes an operation on a parallel variable in the next chapter. For now, note that the **with** statement does not restrict operations on scalar expressions — that is, expressions, like Standard C expressions, that refer to a single data point. This includes parallel variables that are left-indexed so that they specify only one element of the parallel variable — for example, **[4]age** is considered a scalar expression. Dereferenced pointers to parallel variables, however, share the same restrictions as parallel variables.

You can have many **with** statements in a program, making different shapes current at different times. You can also nest **with** statements. When the program returns from the nested **with** statement, the previous shape once again becomes current, as in this example:

```
with (ShapeA) {
   /* Put operations on parallel variables of shape
      ShapeA here. */
```

```
    with (ShapeB) {
        /* Operations on variables of shape ShapeB. */
    }
    /* Operations on variables of shape ShapeA
        once again. */
}
```

3.6 Setting the Context: The where Statement

To perform an operation on a subset of the elements of a parallel variable, use the new C* **where** statement to restrict the context in which the operation is performed. A **where** statement specifies which positions in a shape remain *active*; code in the body of a **where** statement operates only on elements in active positions.

For example, the code below restricts parallel operations to positions of shape **employees** where the value of parallel variable **age** is greater than 35:

```
shape [16384]employees;
unsigned int:employees employee_id, age, salary;

main()
{
    with (employees)
        where (age > 35)
            /* Parallel code in restricted context
               goes here. */
}
```

The controlling expression that **where** evaluates to set the context must operate on a parallel variable of the current shape. It evaluates to 0 (false) or non-zero (true) separately for each position that is currently active. Positions in which the expression is false are made inactive. If no positions are active, code is still executed, but an operation on a parallel variable of the current shape has no result. Initially, all positions in all shapes are active.

You can nest **where** statements; the effect is to cumulatively shrink the set of active positions of the current shape. For example, the two **where** statements below restrict the context to positions of shape **employees** in which **age** is greater than 35 and **salary** is greater than 50000:

```
with (employees)
   where (age > 35) {
      /* Parallel code in restricted context
         goes here. */
      where (salary > 50000) {
         /* Parallel code in more restricted context
            goes here. */
      }
   }
```

Like the **if** statement in Standard C, the **where** statement can include an **else** clause. The **else** clause reverses the set of active positions; that is, those positions that were active when the **where** statement was executed are made inactive, and those that were made inactive are made active. In the example below, it causes parallel operations to be carried out on positions of shape **employees** where **age** is less than or equal to 35 (assuming that all positions are initially active):

```
with (employees)
   where (age > 35)
      /* Parallel code in one restricted context. */
   else
      /* Parallel code in the opposite context. */
```

The everywhere Statement

C* also provides an **everywhere** statement. The **everywhere** statement makes all positions of the current shape active. Parallel code within the scope of an **everywhere** statement operates on all positions of the current shape, no matter what context has been set by previous **where** statements. After the **everywhere** statement, the context returns to what it was before the **everywhere**.

3.7 Dynamically Allocating Shapes and Parallel Variables

One of the powerful features of C* is that it allows you to allocate (and deallocate) shapes and parallel variables dynamically. The functions

allocate_shape and **deallocate_shape** do this for shapes; **palloc** and **pfree** do it for parallel variables.

The allocate_shape and deallocate_shape Functions

Use the C* intrinsic function **allocate_shape** to allocate a shape dynamically. The **allocate_shape** function has two formats. In the first, it takes as arguments a pointer to a shape, the rank of this shape, and the number of positions in each axis. It returns a description of the shape. For example,

```
allocate_shape(&new_shape, 3, 2, 2, 4096);
```

allocates a 3-dimensional shape that is 2-by-2-by-4096.

In the alternative format, you can use an array to specify the rank and the number of positions in each axis. This format is useful if the program will not know the rank until run time, and therefore can't use the variable number of arguments required by the previous syntax.

You can use **allocate_shape** either to allocate a totally new shape, or to complete the specification of a shape that was not fully specified when you declared it.

Use the C* library function **deallocate_shape** to deallocate a shape that was allocated via **allocate_shape**; include the header file **<stdlib.h>** when calling this function. Its argument is a pointer to the shape to be deallocated. For example,

```
deallocate_shape(&new_shape);
```

deallocates the shape allocated above.

You might want to deallocate a shape if you have reached the limit on the number of shapes imposed by your CM system, or if you want to reuse a partially specified shape.

For certain programs, you may be able to improve performance by using the intrinsic function **allocate_detailed_shape**. See the *C* Programming Guide* for information.

The palloc and pfree Functions

Use the C* library function **palloc** to explicitly allocate storage for a parallel variable; use the function **pfree** to free this storage. In both cases, include the header file **<stdlib.h>**.

The **palloc** function takes two arguments: a shape and a size. It allocates space of that size and shape, and returns a pointer to the beginning of the space.

The unit of size in **palloc** is **bool**s. **bool** is a new data type in C* that stores either a 0 or a 1. (The actual size of a **bool** can be different in different implementations.)

To obtain the exact size of a variable or data type in units of **bool**s, use the new C* operator **boolsizeof**. For example,

```
s1 = boolsizeof(int:ShapeA);
```

returns the number of **bool**s that must be allocated for a single instance of a parallel **int**.

The **pfree** function takes as its argument the pointer returned by **palloc**.

The example below allocates a shape and a parallel variable of that shape, using **allocate_shape** and **palloc**, then deallocates both using **pfree** and **deallocate_shape**:

```
#include <stdlib.h>

shape S;
double:S *p;

main()
{
   allocate_shape(&S, 2, 4, 8192);
   p = palloc(S, boolsizeof(double:S));

   /* ... */

   pfree(p);
   deallocate_shape(&S);
}
```

Note the use of **boolsizeof** to obtain the size, in **bool**s, of a parallel **double**.

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.

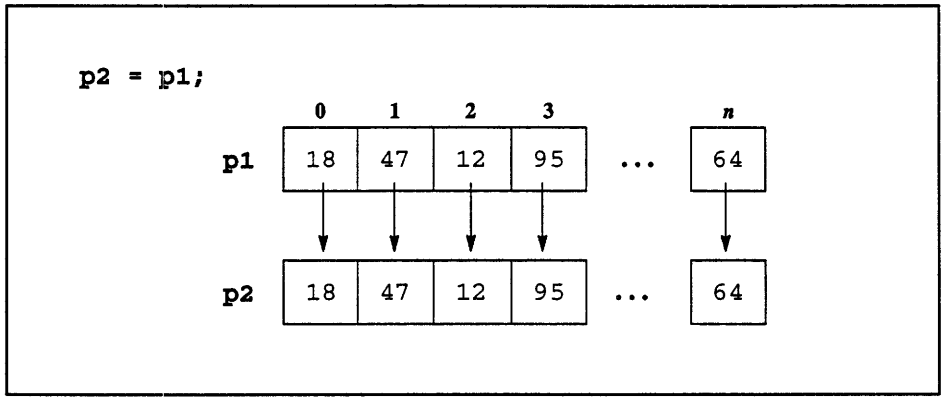


Figure 11. Assignment of a parallel variable to a parallel variable.

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.

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`.

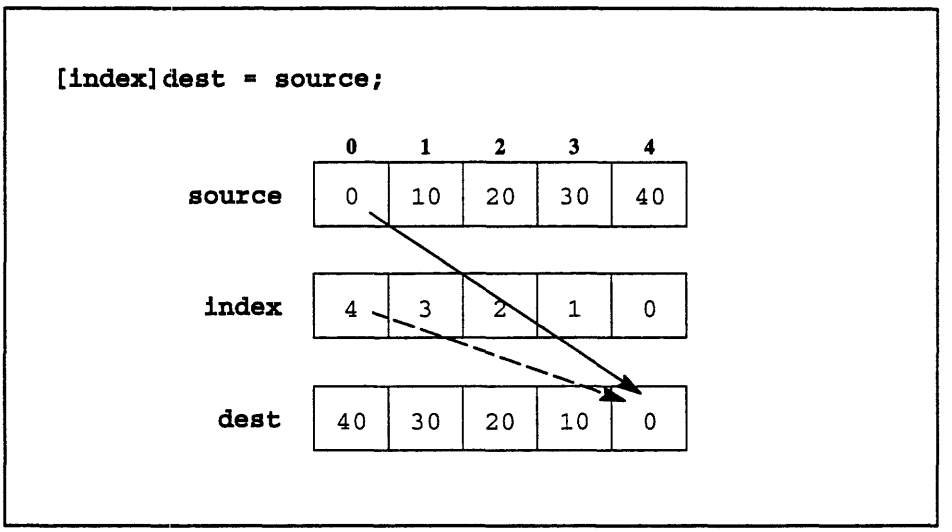


Figure 12. Parallel left indexing — a send operation.

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`.

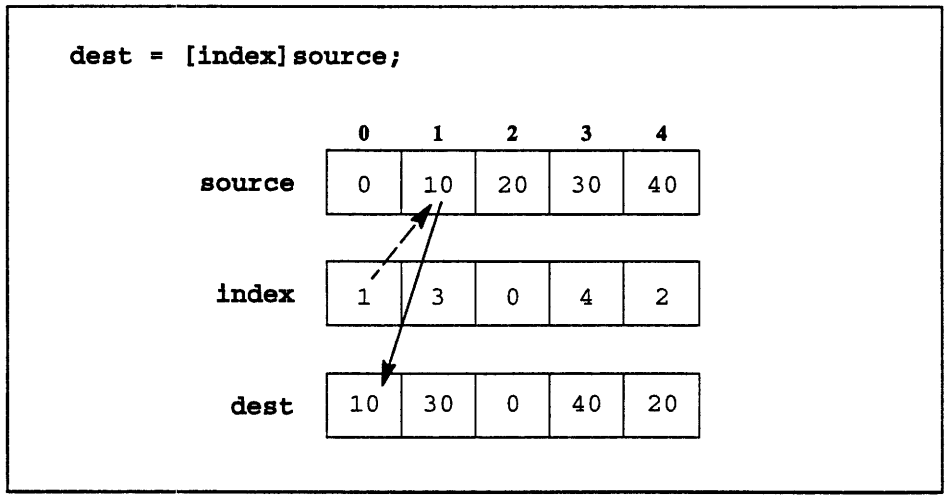


Figure 13. Parallel left indexing of a parallel variable — a get operation.

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.

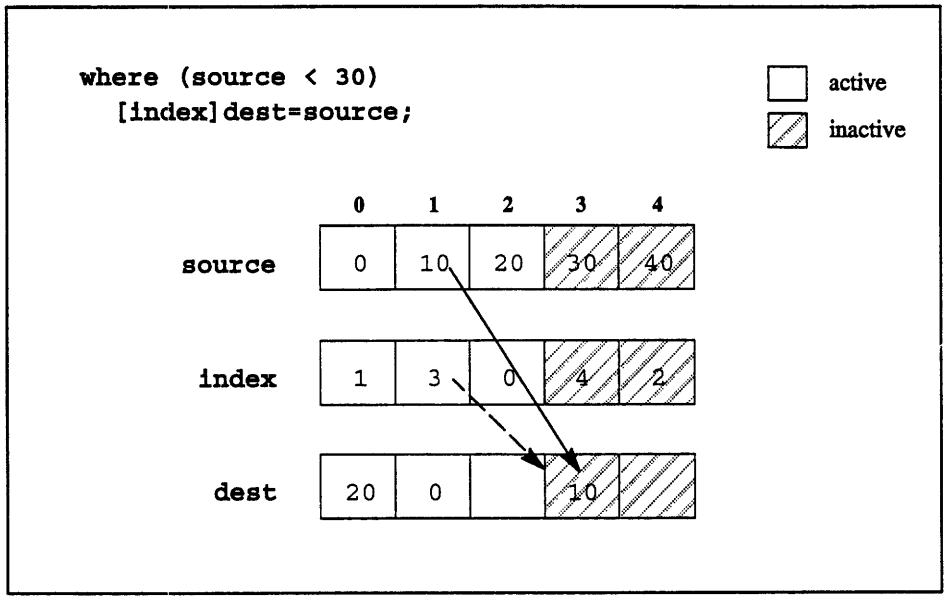


Figure 14. A send operation with inactive positions.

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.

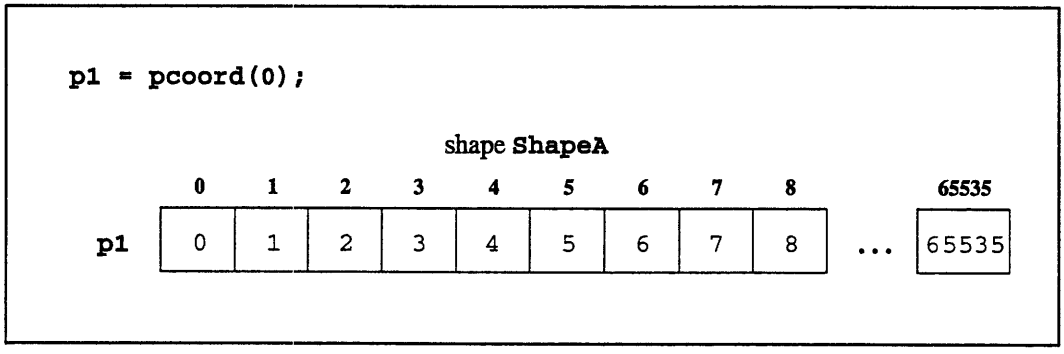


Figure 15. The use of pcoord with a 1-dimensional shape.

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`.

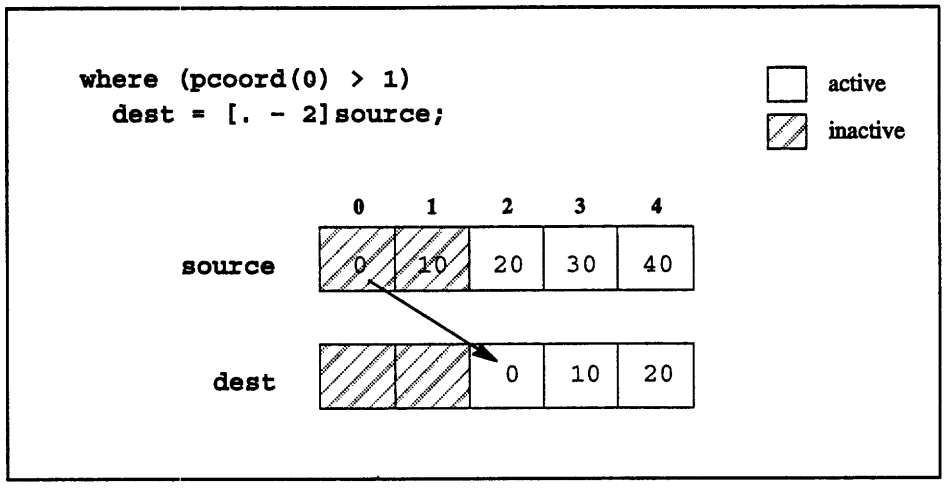


Figure 16. Grid communication without wrapping.

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.

Chapter 6

Sample Programs

This chapter presents three programming examples in C*. If you have C* installed at your site, these programs may be available in the C* examples directory. Check with your system administrator for the location of this directory.

6.1 Julia

This program displays a Julia set. A Julia set is a kind of fractal, obtained by iteratively applying a function at points in a complex plane until the function goes off to infinity. The number of iterations at each point determines how the point is displayed.

Note these points about the program:

- In the program, the shape `julia` is the 1024-by-1024 plane in which the set is to be displayed. The points in the plane are obtained by initializing parallel variables using `pcoord`.
- The `where` statement makes a position inactive when the function goes off to infinity for the complex number at that position; in cases where this does not happen, the `for` loop stops the iterations at 500.
- The program requires no communication among parallel variables.
- The CM-200 version includes calls to the CM graphics library to display the Julia set. When compiling the program, you must link to the `cmsr` and `X11` libraries; the comments at the beginning of the program show how to do this. The program uses the CM Generic Display Interface to prompt the user for the device on which the Julia set is to be displayed:

a framebuffer or an X Window System. **NOTE:** You must use Version 2.0 of the graphics library for this program to compile correctly.

- The CM-5 version uses calls to CMX11. Use these commands to compile and link:

```
% cs -vu julia.cs -lcmx_cm5_vu_sp -lXm -lXt -lXext \
 -lX11
```

CM-200 Version

Here is the CM-200 version:

```
/*
 *
 * This program computes a Julia set (a fractal) in the square region of
 * the complex plane defined by the point (LEFT_SIDE,TOP_SIDE) and the
 * side length SI.
 *
 * The fractal is computed on an N x N grid, with each pixel being
 * handled by a separate virtual processor on the CM. Each pixel will be
 * assigned a value in the range 0 to RES.
 *
 *
 * The fractal is computed by iterating the complex function
 * f(Z) = Z*Z + C for each coordinate in the above region of
 * the complex plane and counting the number of iterations
 * required to cause the function's value to become unbounded
 * at each coordinate. The set of iteration counts is then used as
 * the set of values that is displayed on the output device.
 *
 * NOTE: Complex numbers are handled as pairs of real numbers.
 *
 * To compile this example use the following command:
 *
 *          cs -o julia julia.cs  -lcmsr -lX11
 *
 */

#include <stdio.h>
#include <cm/cmsr.h>
#include <cm/display.h>

/*
```

```
 * parameters defining the size, position, and resolution of the image
 * to be computed
 */

#define N           512
#define RES         500
#define SI          3.0f
#define TOP_SIDE    -1.53f
#define LEFT_SIDE   -1.50f

/* the components of the complex constant, C */

#define CR      0.320f
#define CI      0.043f

main()
{
   shape [N][N]julia ;

   float:julia zr,  zi  ;        /* real and imaginary components of z */
   float:julia zrs, zis ;        /* squares of same                    */
   int:julia   ittr     ;        /* number of cycles before cell done  */

   float cell_size ;
   int   i ;

   /* Set up output device */

   initialize_display() ;

   /* for all pixels simultaneously */

   with ( julia )
   {
      ittr = 0     ;
      zrs  = 0.0f  ;
      zis  = 0.0f  ;

      cell_size = SI / N ;

      /* Give each processor the coordinate in the complex plane that
         it should use. */

      zr = pcoord(1) * cell_size + TOP_SIDE;
      zi = pcoord(0) * cell_size + LEFT_SIDE;

      for(i=0; i<RES; i++)
      {
         /* Perform the iteration for those points in the plane where
            the iteration has not produced an unbounded value. There
```

```
            is a result from the theory of complex numbers that states
            that the values produced by the iteration will remain
            bounded so long as the magnitude of the function's value
            does not exceed 2. */

         where ( zrs+zis <= 4.0f )
         {
            /* iterate f(Z) = Z*Z + C */

            zrs = zr * zr ;
            zis = zi * zi ;
            zi  = 2.0f * zr * zi + CI ;
            zr  = zrs - zis + CR ;

            /* Update the iteration count for those processors that
               are still handling unbounded iterations. */

            ittr = i ;
         }
      }

      /* Done. Display the resulting fractal. */

      CMSR_write_to_display(&ittr);
   }

   if (CMSR_display_type()==CMSR_x_display)
   {  printf("Type return to quit");
      getchar();
   }
}

initialize_display()
{
   int zoom ;

   CMSR_select_display_menu(8, N, N);

   if (CMSR_display_type()==CMSR_cmfb_display)
   {
      zoom = (1024/N) - 1;
      CMFB_set_zoom (CMSR_cmfb_display_display_id(), zoom, zoom, 0);
      CMSR_set_display_offset(128/(zoom+1), 0);
   }
}
```

CM-5 Version

Here is the CM-5 version:

```
/*
 *
 * This program computes a Julia set (a fractal) in the square region of
 * the complex plane defined by the point (LEFT_SIDE,TOP_SIDE) and the
 * side length SI.
 *
 * The fractal is computed on an N x N grid, with each pixel being
 * handled by a separate virtual processor on the CM. Each pixel will be
 * assigned a value in the range 0 to RES.
 *
 *
 * The fractal is computed by iterating the complex function
 * f(Z) = Z*Z + C for each coordinate in the above region of
 * the complex plane and counting the number of iterations
 * required to cause the function's value to become unbounded
 * at each coordinate. The set of iteration counts is then used as
 * the set of values that is displayed on the output device.
 *
 * NOTE: Complex numbers are handled as pairs of real numbers.
 */

#include <stdio.h>
#include <cm/cmxdisplay.h>
#include <cm/cmx-cs.h>

/*
 * parameters defining the size, position and resolution of the image
 * to be computed
 */

#define N          512
#define RES        256
#define SI         3.0f
#define TOP_SIDE   -1.53f
#define LEFT_SIDE  -1.50f

/* the components of the complex constant, C */

#define CR      0.320f
#define CI      0.043f

main(argc, argv)
{
    shape [N][N]julia ;
```

```
float:julia zr,  zi  ;      /* real and imaginary components of z */
float:julia zrs, zis ;      /* squares of same                    */
int:julia   ittr     ;      /* number of cycles before cell done  */
CMX_display_t display;
Widget widget;
int depth, mask;
float cell_size ;
int   i ;

/* Set up X display */
CMXSetArgcArgv(argc, argv);      /* Enable X command-line options */
CMXSetXWindowTitle("Julia Set");
display = CMXCreateSimpleDisplay(256, N, N);
widget = CMXGetWidget(display);
depth = CMXGetDepth(widget);
depth = (depth < 8) ? 1 : 8; /* CMXPutImage supports only 1, 8 */
mask = (1 << depth) - 1;

/* For all pixels simultaneously */
with ( julia )
{
   ittr = 0      ;
   zrs  = 0.0f   ;
   zis  = 0.0f   ;

   cell_size = SI / N ;

   /* give each processor the coordinate in the complex plane that
      it should use */

   zr = pcoord(1) * cell_size + TOP_SIDE;
   zi = pcoord(0) * cell_size + LEFT_SIDE;

   printf("Iteration ");
   for(i=0; i<RES; i++)
   {
    /* Perform the iteration for those points in the plane where
       the iteration has not produced an unbounded value. There
       is a result from the theory of complex numbers that states
       that the values produced by the iteration will remain
       bounded so long as the magnitude of the function's value
       does not exceed two. */

       printf("%d ", i);
       fflush(stdout);
       where ( zrs+zis <= 4.0f )
       {
          /* iterate f(Z) = Z*Z + C */

          zrs = zr * zr ;
```

```
            zis = zi * zi ;
            zi  = 2.0f * zr * zi + CI ;
            zr  = zrs - zis + CR ;

         /* update the iteration count for those processors that are
            still handling unbounded iterations */

            ittr = i ;
          }
       /* done. display the low <depth> bits of the iteration count. */
          CMXPutImage(CMXGetDisplay(display), CMXGetDrawable(display),
          CMXGetGC(display), ittr & mask, depth, 0, 0, 0, 0,
          CMXGetWidth(widget), CMXGetHeight(widget));
       }
       printf("\n");

       }

       printf("Type return to quit");
       getchar();
}
```

Output

Figure 17 shows what the output looks like.

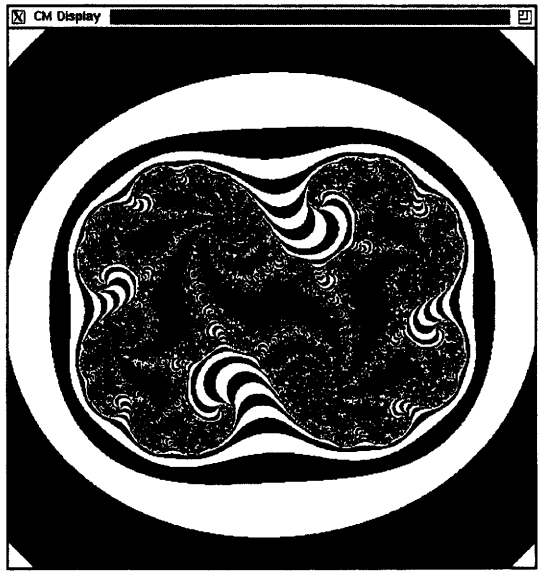


Figure 17. Output of the Julia program.

6.2 Prime Number Sieve

This program finds the prime numbers in a set of numbers. Note these points about the program:

- The function **find_primes** uses the keyword **current** for its parallel variables so that it can be used with any current shape.
- The program uses the **<?=** reduction operator and **pcoord** to obtain the lowest coordinate among active positions of the current shape.
- It uses the parallel conditional expression to initialize the parallel variable **is_candidate**.
- It contains nested **where** statements. Inside the first **where** statement, the program finds the minimum value still active; the second **where** statement makes all multiples of this minimum value active and removes them as candidates to be prime numbers.
- It uses scalar left indexing to select individual elements of parallel variables.

Here is the code:

```
#define MAXIMUM_PRIME 16384

#define FALSE 0
#define TRUE 1
#define FIRST_PRIME 2
/*                                                              */
/*              Function to find prime numbers                  */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*     A pointer, "is_prime_p," to a one-dimensional parallel   */
/*     char that will have non-zero elements in all positions   */
/*      where the index is a prime number                       */
/*                                                              */
/*  Side effects:                                               */
/*                                                              */
/*     find_primes alters the one-dimensional parallel bool     */
/*      that is pointed to by "is_prime_p."                     */
/*                                                              */
/*  Calling constraints:                                        */
/*                                                              */
/*     The shape of the parallel bool pointed to by             */
/*     "is_prime_p" must be the current shape and all           */
```

```
/*      positions must be active                                    */
/*                                                                  */
/*  Algorithm:                                                      */
/*                                                                  */
/*      This function will use the Sieve of Eratosthenes to         */
/*      find the prime numbers.  That is, it will iterate           */
/*      through all numbers which are indices to the one-           */
/*       dimensional parallel bool                                  */
/*                                                                  */
void find_primes(bool:current *is_prime_p) {
   bool:current is_candidate;
   int minimum_prime;

   *is_prime_p = FALSE;

   is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;

   do
      where(is_candidate) {
         minimum_prime = <?= pcoord(0);
         where(!(pcoord(0) % minimum_prime))
            is_candidate = FALSE;
         [minimum_prime]*is_prime_p = TRUE;
         }
   while(|= is_candidate);
}

main() {
   shape [MAXIMUM_PRIME]s;

   bool:s is_prime;
   int i;

   with(s)
      find_primes(&is_prime);
   for(i=0; i<MAXIMUM_PRIME; i++)
      if([i]is_prime)
         printf("The next prime number is %d\n", i);
}
```

The program's output begins:

```
The next prime number is 2
The next prime number is 3
The next prime number is 5
The next prime number is 7
```

... and so on.

6.3 Shuffle

This program provides a perfect shuffle of a deck of cards — that is, the deck is split into two halves, and the cards from each half are alternated. The program continues to shuffle the deck until the original order is obtained again.

Note these points about the program:

- The program uses general communication to rearrange the deck into its new order after each shuffle.
- The CM-200 version uses the shape **physical** — this is a predeclared shape name in C*. The shape **physical** is always of rank 1; its number of positions is the number of physical processors to which the program is attached when it runs on a CM. The program then restricts the context to those positions whose coordinates are less than or equal to **DECK_SIZE**. This is a typical way of changing the number of positions in your shape in CM-200 C* programs, if you don't want a power of 2. Note that the CM-5 version doesn't require a **where** statement, since it can use a shape of 52 positions.

CM-200 Version

Here is the version for CM-200 C*:

```
#include <stdio.h>
#define DECK_SIZE 52
/*                                                              */
/*               Function to print a deck of cards              */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*      A parallel int, "deck," of physical shape, the first    */
/*      DECK_SIZE entries of which contain card numbers.        */
/*                                                              */
/*  Side effects:                                               */
/*                                                              */
/*      The contents of "deck" are printed (allowing three      */
/*      columns per int) followed by a new line.                */
/*                                                              */
/*  Calling constraints:                                        */
/*                                                              */
/*      The first DECK_SIZE entries of physical shape should    */
/*      be active (because "deck" is passed by value) and       */
```

```
/*      DECK_SIZE should be less than or equal to              */
/*      dimof(physical, 0).                                    */
/*                                                             */
/*  Algorithm:                                                 */
/*                                                             */
/*      Self-evident                                           */
/*                                                             */
void print_deck(int:physical deck)
{
   int i;

   for(i = 0; i < DECK_SIZE; i++)
      printf("%3d", [i]deck);
   printf("\n\n");
}

/*                                                             */
/*             Main function to shuffle a deck of cards        */
/*                                                             */
/*  Parameters:                                                */
/*                                                             */
/*      None                                                   */
/*                                                             */
/*  Description:                                               */
/*                                                             */
/*      This program takes a pseudo deck of cards and          */
/*      repeatedly performs the perfect shuffle transformation */
/*      on the deck until it is back in its original order.    */
/*                                                             */
/*      The perfect shuffle is performed by cutting the deck   */
/*      in the middle and then interleaving cards from the     */
/*      two half decks.  For example, if the original deck     */
/*      contained the cards 0, 1, 2, 3, 4, and 5, the first cut */
/*      deck would contain 0, 1, and 2, and the second cut     */
/*      deck would contain 3, 4, and 5.  Interleaving these    */
/*      two decks results in 0, 3, 1, 4, 2, and 5.             */
/*                                                             */
/*  Side effects:                                              */
/*                                                             */
/*      The program performs output.                           */
/*                                                             */
/*  Program constraints:                                       */
/*                                                             */
/*      The number of cards in the deck, DECK_SIZE, should be  */
/*      less than or equal to dimof(physical, 0).              */
/*                                                             */
/*  Algorithm:                                                 */
/*                                                             */
/*       A "send" is used to perform the shuffle.              */
/*                                                             */
```

```
main()
{
   int:physical original_deck, deck, shuffling_order;

   /* offset is the half-way point in the deck (for cutting purposes) */
   int offset = (DECK_SIZE+1)/2, number_shuffles = 0;

with(physical)
   /* only positions in the deck are left active */
   where((deck = original_deck = pcoord(0)) < DECK_SIZE) {
      printf("original deck:");
      print_deck(original_deck);

      /* generate the perfect shuffle transformation: positions in the
      first half of the deck are to be sent
      to consecutive even positions
      in the shuffled deck; whereas positions in the second half of the
      deck are to be sent to consecutive odd positions in
      the shuffled deck*/

      shuffling_order = (2*deck < DECK_SIZE) ? (2*deck)
                        : (2*(deck-offset)+1);

      printf("shuffle order:");
      print_deck(shuffling_order);

      do {
         /* perform the shuffle */
         [shuffling_order]deck = deck;

         /* print the shuffled deck and an incremented
         sequence number */
         printf("%3d:", ++number_shuffles);
         print_deck(deck);
      /* continue to shuffle until the deck is in its original order */
      } while(|=(deck != original_deck));

      /* print the number of shuffles required */
      printf("\nNumber of shuffles = %d\n", number_shuffles);
   }
}
```

CM-5 Version

Here is the version for CM-5 C*:

```
#include <stdio.h>
#define DECK_SIZE 52
shape [DECK_SIZE]deck_shp;

/*                                                              */
/*               Function to print a deck of cards              */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*      A parallel int, "deck," of physical shape, the first    */
/*      DECK_SIZE entries of which contain card numbers.        */
/*                                                              */
/*  Side effects:                                               */
/*                                                              */
/*      The contents of "deck" are printed (allowing three      */
/*      columns per int) followed by a new line.                */
/*                                                              */
/*  Calling constraints:                                        */
/*                                                              */
/*      The first DECK_SIZE entries of physical shape should    */
/*      be active (because "deck" is passed by value) and       */
/*      DECK_SIZE should be less than or equal to               */
/*      dimof(physical, 0).                                     */
/*                                                              */
/*  Algorithm:                                                  */
/*                                                              */
/*      Self-evident                                            */
/*                                                              */
void print_deck(int:deck_shp deck)
{
   int i;

   for(i = 0; i < DECK_SIZE; i++)
      printf("%3d", [i]deck);
   printf("\n\n");
}

/*                                                              */
/*            Main function to shuffle a deck of cards          */
/*                                                              */
/*  Parameters:                                                 */
/*                                                              */
/*      None                                                    */
/*                                                              */
/*  Description:                                                */
```

```
/*                                                                  */
/*      This program takes a pseudo deck of cards and               */
/*      repeatedly performs the perfect shuffle transformation      */
/*      on the deck until it is back in its original order.         */
/*                                                                  */
/*      The perfect shuffle is performed by cutting the deck        */
/*      in the middle and then interleaving cards from the          */
/*      two half decks.  For example, if the original deck          */
/*      contained the cards 0, 1, 2, 3, 4, and 5, the first cut     */
/*      deck would contain 0, 1, and 2, and the second cut          */
/*      deck would contain 3, 4, and 5.  Interleaving these         */
/*      two decks results in 0, 3, 1, 4, 2, and 5.                  */
/*                                                                  */
/*  Side effects:                                                   */
/*                                                                  */
/*      The program performs output.                                */
/*                                                                  */
/*  Program constraints:                                            */
/*                                                                  */
/*      The number of cards in the deck, DECK_SIZE, should be       */
/*      less than or equal to dimof(physical, 0).                   */
/*                                                                  */
/*  Algorithm:                                                      */
/*                                                                  */
/*       A "send" is used to perform the shuffle.                   */
/*                                                                  */

main()
{
   int:deck_shp original_deck, deck, shuffling_order;

   /* offset is the half-way point in the deck (for cutting purposes) */
   int offset = (DECK_SIZE+1)/2, number_shuffles = 0;

with(deck_shp) {
   /* only positions in the deck are left active */
   deck = original_deck = pcoord(0)
   printf("original deck:");
   print_deck(original_deck);

   /* generate the perfect shuffle transformation: positions in the
      first half of the deck are to be sent
      to consecutive even positions
      in the shuffled deck; whereas positions in the second half of the
      deck are to be sent to consecutive odd positions in
      the shuffled deck */

   shuffling_order = (2*deck < DECK_SIZE) ? (2*deck)
                       : (2*(deck-offset)+1);
```

```
    printf("shuffle order:");
    print_deck(shuffling_order);

    do {
        /* perform the shuffle */
        [shuffling_order]deck = deck;

        /* print the shuffled deck and an incremented
           sequence number */
        printf("%3d:", ++number_shuffles);
        print_deck(deck);
    /* continue to shuffle until the deck is in its original order */
    } while(|=(deck != original_deck));

    /* print the number of shuffles required */
    printf("\nNumber of shuffles = %d\n", number_shuffles);
    }
}
```

Output

Here is the output from the program:

```
original deck:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51

shuffle order:  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
38 40 42 44 46 48 50  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33
35 37 39 41 43 45 47 49 51
  1:  0 26  1 27  2 28  3 29  4 30  5 31  6 32  7 33  8 34  9 35 10 36 11
37 12 38 13 39 14 40 15 41 16 42 17 43 18 44 19 45 20 46 21 47 22 48 23
49 24 50 25 51

  2:  0 13 26 39  1 14 27 40  2 15 28 41  3 16 29 42  4 17 30 43  5 18 31
44  6 19 32 45  7 20 33 46  8 21 34 47  9 22 35 48 10 23 36 49 11 24 37
50 12 25 38 51

  3:  0 32 13 45 26  7 39 20  1 33 14 46 27  8 40 21  2 34 15 47 28  9 41
22  3 35 16 48 29 10 42 23  4 36 17 49 30 11 43 24  5 37 18 50 31 12 44
25  6 38 19 51

  4:  0 16 32 48 13 29 45 10 26 42  7 23 39  4 20 36  1 17 33 49 14 30 46
11 27 43  8 24 40  5 21 37  2 18 34 50 15 31 47 12 28 44  9 25 41  6 22
38  3 19 35 51
```

```
  5:  0  8 16 24 32 40 48  5 13 21 29 37 45  2 10 18 26 34 42 50  7 15 23
31 39 47  4 12 20 28 36 44  1  9 17 25 33 41 49  6 14 22 30 38 46  3 11
19 27 35 43 51

  6:  0  4  8 12 16 20 24 28 32 36 40 44 48  1  5  9 13 17 21 25 29 33 37
41 45 49  2  6 10 14 18 22 26 30 34 38 42 46 50  3  7 11 15 19 23 27 31
35 39 43 47 51

  7:  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44
46 48 50  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41
43 45 47 49 51

  8:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51

Number of shuffles = 8
```

Chapter 7

Performance Hints

This chapter gives some tips on how to improve the performance of C* programs. For more information on CM-200 C* performance, see the *C* Programming Guide*. For more information on CM-5 C* performance, see the *CM-5 C* Performance Guide*.

- **Prototype functions.** Using ANSI function prototyping speeds up a program by reducing the number of conversions that the compiler must make.
- **Pass parallel variables by reference.** In function calls, passing a parallel variable by reference rather than by value saves the need to make a temporary copy of the parallel variable.
- **Use the `everywhere` statement in functions when all positions are active.** Enclosing the function's statements in an `everywhere` statement lets the compiler use faster instructions that ignore the context.
- **Do not store scalar data in parallel variables.** If data is scalar, declare it as a Standard C variable, so that it is stored on the scalar computer (front end or partition manager).
- **Declare float constants as floats.** That is, add the final `f` to the value; this reduces the number of conversions that the compiler must make.
- **If possible, put parallel variables that are to communicate in the same shape.** This reduces the amount of communication required.
- **Whenever possible, use grid communication instead of general communication.** As mentioned in Chapter 5, grid communication is faster.
- **Whenever possible, use send operations instead of get operations for general communication.** As mentioned in Chapter 5, send operations are faster.

Index

Symbols

`?:`, 27

`%%`, 40

`+=`, 28

`<?=`, 29, 50

`>?=`, 29

A

active positions, 7

`allocate_detailed_shape`, 23

`allocate_shape`, 23

arrays, parallel, 18

assignment, with a scalar LHS and a parallel RHS, 26

axis, 14

B

binary operators
  with a scalar and a parallel operand, 26
  with two parallel operands, 27

`bool`, 24

`boolsizeof`, 24

C

C*, and C, 1

C* implementations, 2

C* program
  compiling, 11–12
  developing, 2
  executing, 12
  sample, 3, 43

`cmattach`, 2

communication, 33
  See also general communication; grid communication

conditional expression, 27

context, 21

coordinates, 16

`current`, 29, 50

current shape, 7, 20

D

`deallocate_shape`, 23

demotion, parallel to scalar, 26

`dimof`, 40

E

elements, 3, 6, 15
  corresponding, 16

`else`, 22

`everywhere`, 22, 59
  and passing by value, 30

F

functions, 29
  prototyping, 59

G

general communication, 33, 52, 59
  when there are inactive positions, 36

get operations, 35, 59

grid communication, 33, 37, 59
  with wrapping, 40
  without wrapping, 39

L

left indexing, 14
  and **pcoord**, 38
  parallel, 34

M

maximum operator, 29
minimum operator, 29, 50

O

overload, 31
overloading functions, 30

P

palloc, 24
parallel variables, 5, 15
  choosing an individual element of, 9–10
  declaring, 15
  dynamically allocating, 22
  passing to a function, 29
passing by reference, 30, 59
passing by value, 59
  and **where** statement, 30
pcoord, 37, 43, 50
  and left indexing, 38
performance, improving, 59
period, and **pcoord**, 39
pfree, 24
physical predeclared shape name, 52
pointers
  to parallel variables, 19
    passing to a function, 29
  to shapes, 19
positions, 5, 13
  active, 21
  inactive, 21
promotion, scalar to parallel, 26

R

rank, 13, 34
reduction assignment, 10
reduction operators, 28

S

scalar variables, 6
scan, 34
send operations, 35, 59
shape selection, 7
shapeof, 30
shapes, 5, 13
  declaring, 14
  dynamically allocating, 22
  fully specified, 14
spread, 34
<stdlib.h>, 23, 24
structures, parallel, 17

U

unary operators, and parallel variables, 25

V

void, 29

W

where, 21, 43
  and passing by value, 30
  nested, 21, 50
with, 7, 20
  in functions, 29

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1264
(617) 234-1000