The
Connection Machine
System

C* Programming Guide

Version 6.0
November 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Part II Programming in C*

Part III C* Communication Functions

Appendixes

About This Manual

Objectives of This Manual

This manual is intended to help you learn how to program in the C* data parallel programming language.

Intended Audience

Readers are assumed to have a working knowledge of C programming and a general understanding of the components of the Connection Machine system.

Revision Information

This is a new manual.

Organization of This Manual

Part I Getting Started

These two chapters introduce C* and data parallel programming on the Connection Machine system and provide a step-by-step explanation of a simple program.

Part II Programming in C*

These eight chapters describe how to write programs in C*.

Part III C* Communication Functions

Data parallel programming lets you operate on large multi-dimensional sets of data at the same time. These four chapters describe C* library functions that you can use to transfer values among items in the data set and to perform cumulative operations along any of the dimensions of the data set.

Appendix A Improving Performance

This appendix suggests ways of increasing the performance of a C* program.

There is also a glossary that defines technical terms used in the manual.

Associated Documents

The following document about C* appears in the same volume as this programming guide:

- *C* User's Guide*

In addition, a technical report is available that provides a reference description of the C* language.

Information about related aspects of programming the Connection Machine system is contained in the following volumes of the Connection Machine documentation set:

- *Connection Machine Front-End Systems*
- *Connection Machine I/O Programming*
- *Connection Machine Graphics Programming*
- *Connection Machine Parallel Instruction Set*
- *Connection Machine Programming in C/Paris*

C* is based on the standard version of the C programming language proposed by the X3J11 committee of the American National Standards Institute; this version is referred to as *standard C* in this manual. The standard is available from:

> X3 Secretariat
> Computer and Business Equipment Manufacturers Association
> 311 First Street, N.W.
> Suite 500
> Washington, DC 20001–2178

Related books about standard C include the following:

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition (Englewood Cliffs, New Jersey: Prentice-Hall, 1988)
- Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual*, 2nd edition (Englewood Cliffs, New Jersey: Prentice-Hall, 1987)

Notation Conventions

The table below displays the notation conventions used in this manual:

Convention	Meaning
`bold typewriter`	C* and C language elements, such as keywords, operators, and function names, when they appear embedded in text. Also UNIX and CM System Software commands, command options, and file names.
italics	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`%` **`bold typewriter`** `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in regular `typewriter` font.

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 back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–4000 (617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl-M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Part I
Getting Started

Chapter 1

What Is C*?

C* (pronounced "sea–star") is an extension of the C programming language designed for the Connection Machine data parallel computing system. This chapter introduces C* and data parallel computing on the Connection Machine system.

1.1 Data Parallel Computing

In the data parallel computing model, there are many small processors, each with some associated memory, and all acting under the direction of a serial computer called the *front end*. Each processor stores the information for one data point in its local memory; all processors can then perform the same operation on all the data points at the same time.

Here are some examples of how data parallel computing can be used:

- A graphics program might store pixels one-per-processor and then have each processor calculate the color value for its pixel, all at the same time.
- A text retrieval program might store articles one-per-processor and then have each processor search its article for a keyword.
- A modeling program (for example, one that simulates fluid flow) might create a large number of individual cells, stored one-per-processor. Each cell might have a small number of possible states, which are simultaneously updated at each "tick" of a clock according to a set of rules that are applied to each cell.

Data parallel programming also has the following features:

- A programmer can specify that only a particular subset of the processors is to carry out an operation. In the text retrieval program, for example, the processors that find the initial keyword might be instructed to search further for another keyword, while those that did not find the initial keyword remain idle.
- Processors can pass messages to each other. For example, color shading in a graphic image requires that each processor obtain surface information from surrounding processors to calculate the result for its pixel.

1.2 The Connection Machine System

In the Connection Machine system, the front end is a standard serial computer, such as a Sun-4 or certain models of VAX. A bus interface connects the front end to the CM itself. Programs for the CM reside on and run from the front end. Serial code within a program is executed on the front end in the usual manner; parallel code is executed by the CM processors.

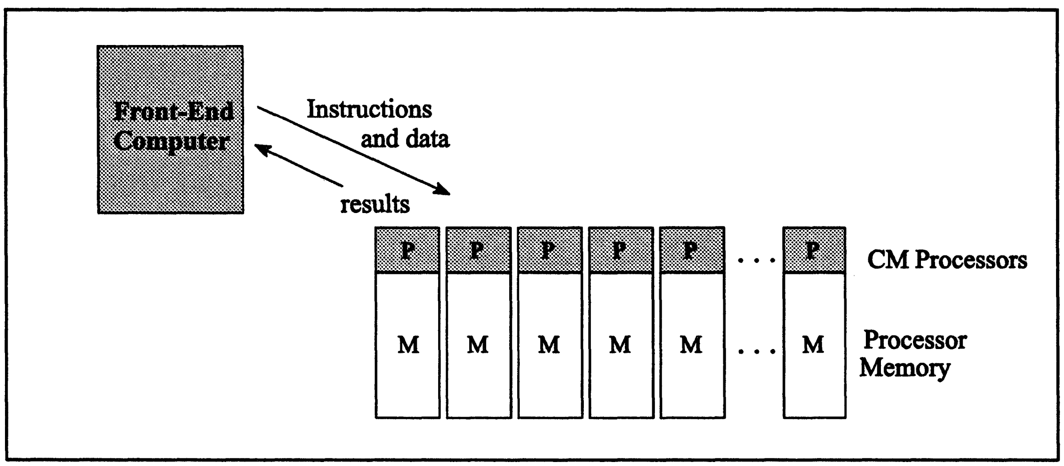

Figure 1. Interactions between front end and CM

1.2.1 Virtual Processors

Different CM models have different numbers of processors. This does not limit the size of the data set that a program can use, however, since the CM system supports *virtual* processors. The CM can divide up the memory associated with each physical processor to create power-of-2 multiples of the entire set of processors. A CM with 16K physical processors, for example, can operate as if it has 32K processors, 64K processors, and so on. The time required to do each operation increases as the number of virtual processors increases.

1.2.2 Communication

CM processors are interconnected by a high-speed communication device called a *router*. The router allows each processor to send a message to any other processor, all at the same time. In addition, the CM system has a faster form of communication called *grid communication*, which allows processors to communicate with their neighbors.

1.2.3 I/O

Different devices can be used to perform I/O to and from CM memory:

- A mass storage system called the DataVault can be attached to the CM via a 64-bit I/O bus. The DataVault provides permanent disk storage for CM data.
- Other I/O devices can be connected to the CM via this same I/O bus or through an interface to a computer with aVMEbus.
- The graphic display system and associated software provides I/O from the CM to a display monitor.

Various user commands and system calls are available to perform Connection Machine I/O. For more information, see the volume *Connection Machine I/O Programming*. In addition, data can be moved between the CM and the front end using C* and calls to Paris, the CM's parallel instruction set. For more information, see the *C* User's Guide*.

1.3 C* and C

C* implements the ANSI standard C language; programs written in standard C compile and run correctly under C* (except when they use one of the words that are newly reserved in C*). In addition, C* provides new features that make possible the kind of data parallel computing described in Section 1.1. These features include the following:

- A method for describing the size and shape of parallel data and for creating parallel variables. Shapes and parallel variables are discussed in Chapters 3, 4, and 9.
- New operators and expressions for parallel data, and new meanings for standard operators that allow them to work with parallel data. Operators are discussed in Chapter 5.
- Methods for choosing the parallel variables, and the specific data points within parallel variables, upon which C* code is to act. These features are discussed in Chapters 4 and 6.
- New kinds of pointers that point to parallel data and to shapes. C* pointers are discussed in Chapter 7.
- Changes to the way functions work so that, for example, a parallel variable can be used as an argument. Chapter 8 describes C* functions.
- Methods for communication among parallel variables. See Chapter 10.
- Library functions that also allow communication among parallel variables. Chapters 11–14 describe these functions.

1.3.1 Program Development Facilities

C* uses its own compiler, run-time libraries, and header files. The compiler translates a C* program into a serial C program made up of standard serial C code and calls to Paris. This code is then passed to the front end's C compiler, which handles it in the normal way to produce an executable load module. The serial C code is executed on the front end; the Paris instructions are executed on the CM.

C* can use standard UNIX programming tools such as **dbx**, **gprof**, and **make**. The C* compiler and related program development facilities are described more fully in the *C* User's Guide*.

1.4 C* and the CM

Although C* is designed for the CM system, it is not necessary to understand the details of the CM hardware in order to use the language. For example, when the size of the data set requires it, C* automatically takes advantage of the virtual processor mechanism described above; the programmer need not be aware of the details. If you do understand the CM hardware, however, the relationship between the language and the system may sometimes be clear: for example, "positions" in C* are implemented in the CM on physical processors or on virtual processors.

If you are familiar with Paris, the CM's parallel instruction set, you will probably find it helpful to consult Chapter 2 of the *C* User's Guide*. A section in this chapter describes the relationship between C* concepts such as shapes and parallel variables and Paris concepts such as VP sets and field IDs.

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 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*. (This is, by the way, a typical use of the CM, with parallel variables having tens of thousands of elements.) It then assigns integer constants to each element of these parallel variables and performs simple arithmetic on them.

Example 1. A simple C* program: **add.cs**

```
#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
 */
      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);
   }
}
```

Example 1. Output

```
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;
```

declares a shape called **ShapeA**. **ShapeA** consists of 65,536 *positions*, as shown in Figure 2.

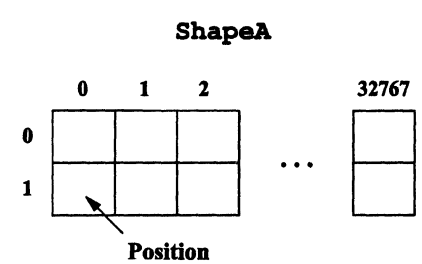

Figure 2. The shape **ShapeA**

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 3 shows the three parallel variables of shape **ShapeA**.

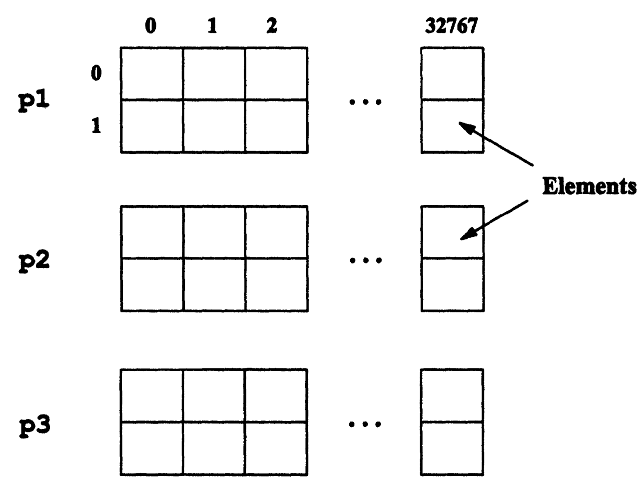

Figure 3. Three parallel variables of shape **ShapeA**

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 standard C variable. These C variables are called *scalar* in this guide to distinguish them from C* parallel variables. Memory for standard C variables is allocated on the front end rather than on the CM.

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 4.

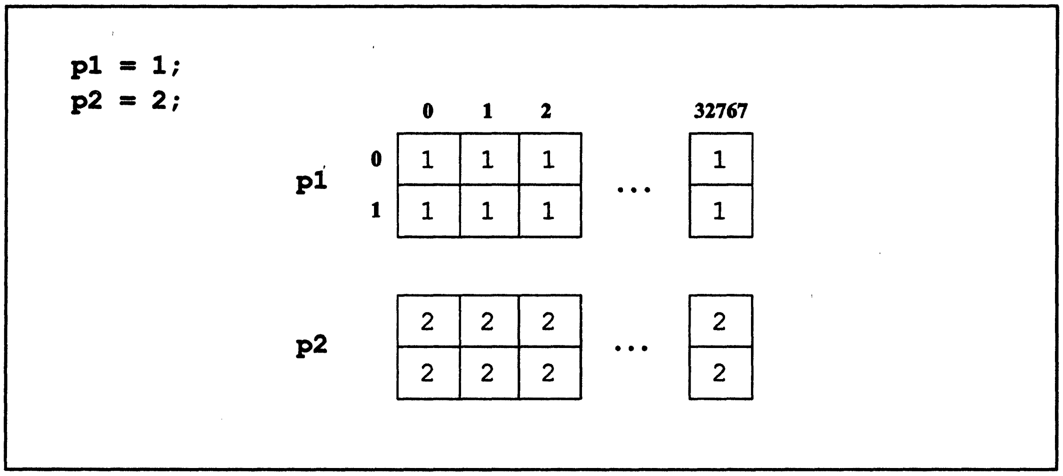

Figure 4. Initialized parallel variables

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 5 shows the result of this statement.

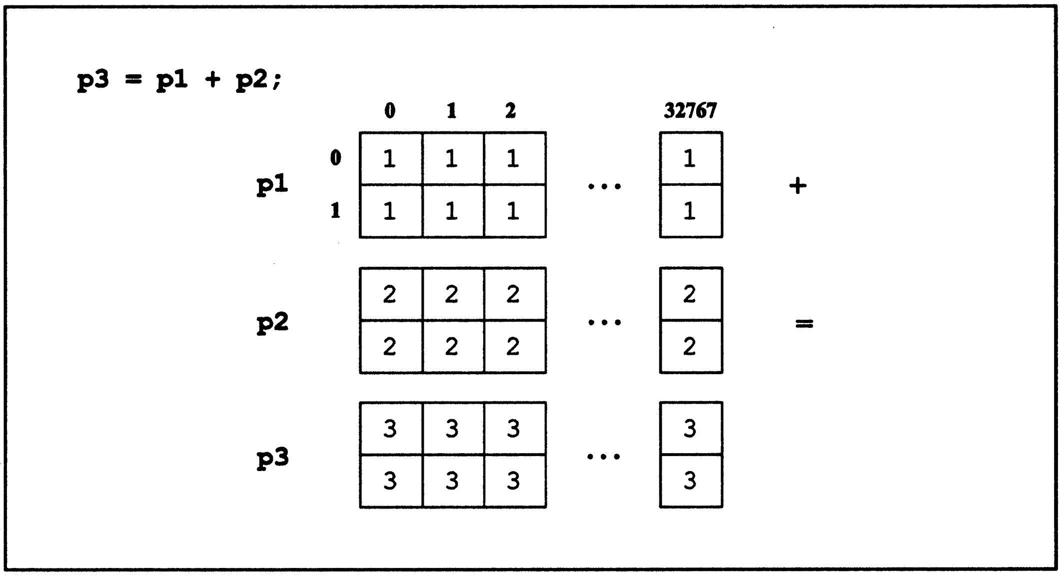

Figure 5. Addition of parallel variables

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.

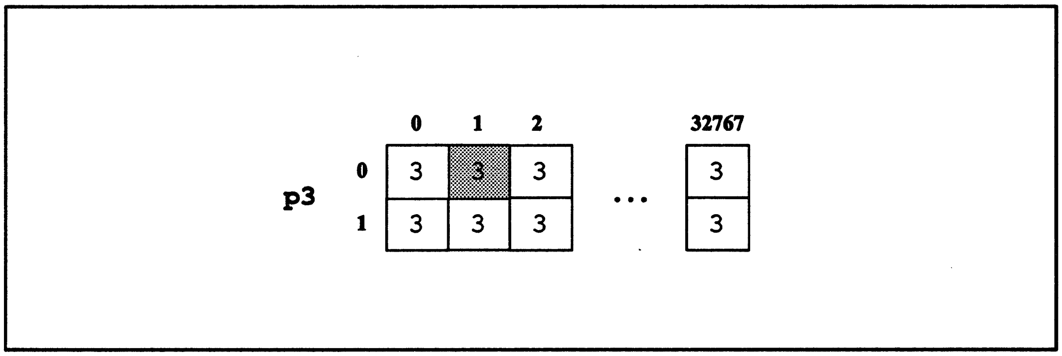

Figure 6. Element [0][1] of **p3**

Note that the following **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. The following 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 7. (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.)

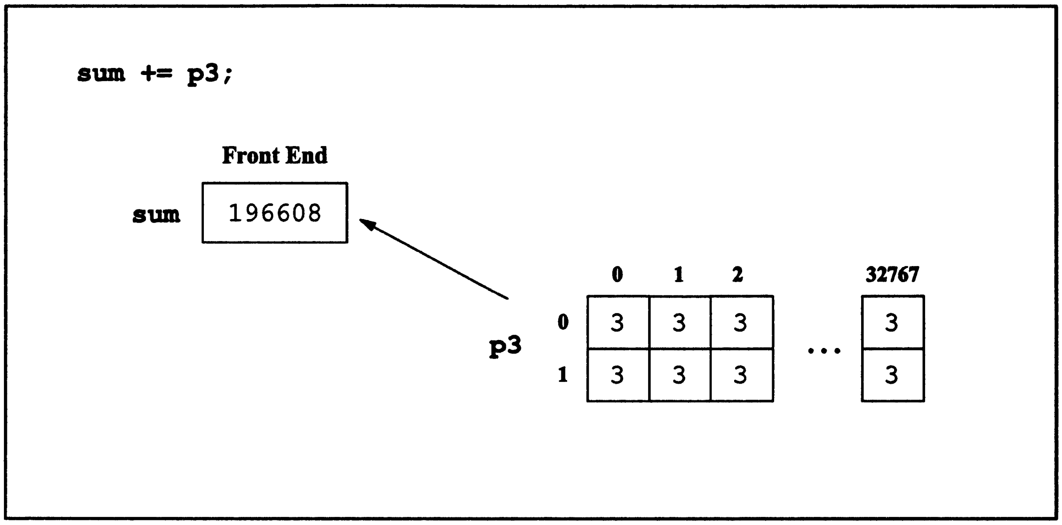

Figure 7. The reduction assignment of parallel variable **p3**

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 compiler command **cs** on the front end. To compile the program **add.cs**, type the following:

```
% 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**.

2.7.2 Executing

To execute the resulting load module, you can use the CM System Software command **cmattach**, as in the following example:

```
% cmattach a.out
```

Issuing this command for the executable version of **add.cs** produces a response from the system like the following (provided that CM resources are available):

```
Attaching the Connection Machine system [ name ]...
cold booting... done.
Attached to 8192 physical processors on sequencer 0, microcode
version 6002
Paris safety is off.


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


Detaching... done.
%
```

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

Part II
Programming in C*

Chapter 3

Using Shapes and Parallel Variables

The sample C* program in Chapter 2 began by declaring a shape and several parallel variables. Shapes and parallel variables are the two most important additions of C* to standard C. This chapter introduces these topics; Chapter 9 discusses them in more detail.

3.1 What Is a Shape?

A shape is a template for parallel data, a way of logically configuring data. In C*, you must define the shape of the data before you can operate on it. A shape is defined by the following:

- The number of its dimensions. This is referred to as the shape's *rank*. For example, a shape of rank 2 has two dimensions. A shape can have from 1 to 31 dimensions. A dimension is also referred to as an *axis*.
- The number of *positions* in each of its dimensions. A position is an area that can contain individual 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. Thus, a 2-dimensional shape with 4 positions in axis 0 (the first dimension) and 8 positions in axis 1 (the second dimension) has 32 total positions, organized as shown in Figure 8. (By convention in this guide, axis 0 denotes the row number, and axis 1 denotes the column number.)

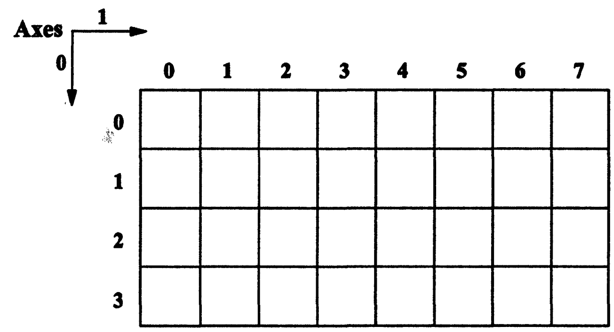

Figure 8. A 4-by-8 shape

The CM system currently imposes the following restrictions on shapes in C*:

- The number of positions in each dimension of a shape must be a power of two.
- 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.

For example, if the program is running in a CM section with 8192 physical processors, it can have shapes with 8192 positions, 16384 positions, and so on. You can arrange them 2 by 4096, 4 by 4 by 512, and so on.

3.2 Choosing a Shape

The choice of a shape depends on the data that the C* program is going to be using. The shape typically reflects the natural organization of the data. For example:

- A database program for the employee records of a large company might use a 1-dimensional shape, with the number of positions equaling the number of employees.
- A graphics program might use a shape representing the 2-dimensional images that the program is to process. If the images have 256 pixels in the vertical dimension and 256 pixels in the horizontal dimension, a shape of rank 2 with 256 positions

in each dimension would be appropriate. This would let each position represent a pixel in an image.

- A program to analyze stress in a solid object might use a 3-dimensional shape, with each axis representing a dimension of the object, and each position representing some portion of the volume of the object.

3.3 Declaring a Shape

Here is a declaration of a shape in C*:

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

This statement declares a shape called **employees**. It has one dimension (a rank of 1) and 16384 positions.

Let's take a closer look at the components of the statement:

- **shape** is a new keyword that C* adds to standard C.
- **[16384]** specifies the number of positions in the shape. If the shape is declared at file scope, or as an **extern** or **static** at block scope, the value in brackets must be a constant expression. Otherwise, it can be any expression that can be evaluated to an integer. This follows the ANSI C standard.
- **employees** is the name of the shape. Shape names follow standard C naming rules. They are in the same name space as variables, functions, **typedef** names, and enumeration constants.

Figure 9 shows this shape.

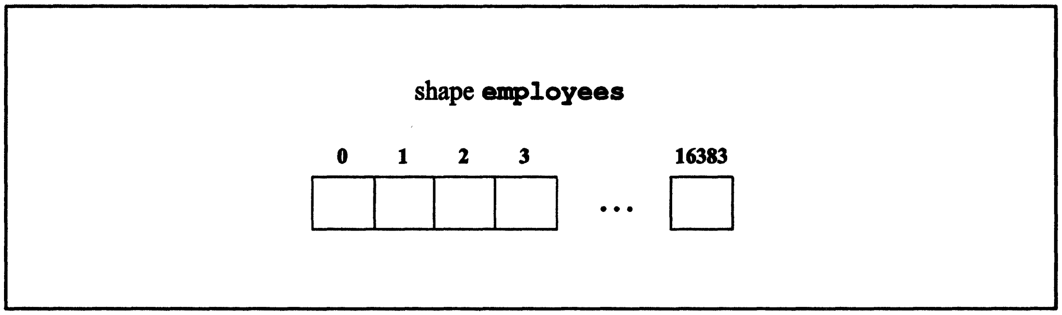

Figure 9. The shape **employees**

A 2-dimensional shape adds another number, in brackets, to the right of the first set of brackets. This number represents the number of positions in the second dimension. For example:

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

This shape has 256 positions along axis 0 and 512 positions along axis 1. Each additional dimension is represented by another number in brackets, to the right of the previous dimensions.

Individual positions within a shape can be identified using bracketed numbers as *coordinates*. For example, position [4] of shape **employees** is the fifth position in the shape (numbering starts with 0, as in C). Position [47][112] of shape **image** is the position at coordinate 47 along axis 0 and 112 along axis 1.

3.3.1 Declaring More Than One Shape

A program can include many shapes. You can use a single **shape** statement to declare more than one shape. For example:

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

3.3.2 The Scope of a Shape

A shape's scope is the same as that of any identifier in standard C. For example, a shape declared within a function or block is local to that function or block. A shape declared at global scope can be referenced anywhere in the source file after its declaration.

NOTE: If a block contains a shape declaration, you should not branch into it (for example, with a **switch** or **goto** statement); the behavior is undefined.

3.4 Obtaining Information about a Shape

You can obtain information about a shape by using the C* intrinsic functions **positionsof**, **rankof**, and **dimof**. (Intrinsic functions are new in C*; they have function-like syntax, but they must be known to the compiler—for example, because they don't follow all ANSI C rules for functions.)

- **positionsof** takes a shape as an argument and returns the total number of positions in the shape.
- **rankof** takes a shape as an argument and returns the shape's rank.
- **dimof** takes two arguments: a shape and an axis number. It returns the number of positions along that axis.

The simple C* program in Example 2 displays information about a shape.

Example 2. Obtaining information about a shape: **shape.cs**

```
#include <stdio.h>

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

main()
{
 printf ("Shape 'employees' has rank %d and %d positions.\n",
         rankof(employees), positionsof(employees));
 printf ("Shape 'image' has rank %d and %d positions.\n",
         rankof(image), positionsof(image));
 printf ("Axis 0 has %d positions; axis 1 has %d positions.\n",
         dimof(image,0), dimof(image,1));
}
```

Example 2. Output

```
Shape 'employees' has rank 1 and 16384 positions.
Shape 'image' has rank 2 and 131072 positions.
Axis 0 has 256 positions; axis 1 has 512 positions.
```

These intrinsic functions can be used in other, more interesting contexts, as we discuss later.

3.5 More about Shapes

So far, we have covered the basics about shapes in C*. Chapter 9 discusses more advanced aspects of shapes. For example:

- Partially specifying a shape
- Copying shapes
- Dynamically allocating a shape

3.6 What Is a Parallel Variable?

Once a program has declared a shape, it can declare variables of that shape. These variables are called *parallel variables*.

3.6.1 Parallel and Scalar Variables

A good way to understand parallel variables is to compare them with standard C variables. As we mentioned in Chapter 2, standard C variables are referred to in this guide as *scalar* to distinguish them from parallel variables. A scalar variable contains only one "item"—one number, one character, and so on. A parallel variable contains many items. (Note that ANSI C uses the term *scalar* in a slightly different way, to refer collectively to arithmetic and pointer types. We consider a standard C array or structure, for example, to be scalar because it contains only one array or structure.)

A scalar variable has the following associated with it:

- A type, along with its modifiers and qualifiers, (for example, **`char`**, **`unsigned int`**, **`long double`**) that defines how much memory is to be allocated for the variable and how operators deal with it
- A storage class (for example, **`auto`**, **`static`**) that defines the manner in which the memory is to be allocated

Like a scalar variable, a parallel variable has a type and a storage class, but in addition it has a *shape*. The shape defines how many *elements* of a parallel variable exist, and how they are organized. Each element exists in a position in the shape and contains a single value for the parallel variable. 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 the advantage of a parallel variable is that C* allows a program to carry out operations on all elements (or any subset of elements) of a parallel variable at the same time. As the sample program in Chapter 2 demonstrated, you can:

- Assign a constant to all elements of a parallel variable at the same time.
- Declare multiple parallel variables of the same shape.
- Perform an arithmetic operation on all elements of a parallel variable at the same time.
- Do reduction assignments of data in all elements of a parallel variable.

As we explain later in this manual, parallel variables that have different shapes can interact, but interactions between parallel variables are more efficient if the parallel variables are of the same shape.

3.7 Declaring a Parallel Variable

Before declaring a parallel variable, you must define the shape that the parallel variable is to take. For example, assume that the following shape has been defined:

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

You can then declare parallel variables of this shape. For example:

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

Interpret the colon in this syntax to mean "of shape *shapename*." Thus, this statement declares a parallel variable called **`employee_id`** that is of shape **`employees`**. **`unsigned int`** specifies the type of the parallel variable **`employee_id`**. Parallel variable names, like shape names, follow standard C naming rules.

Figure 10 shows this parallel variable.

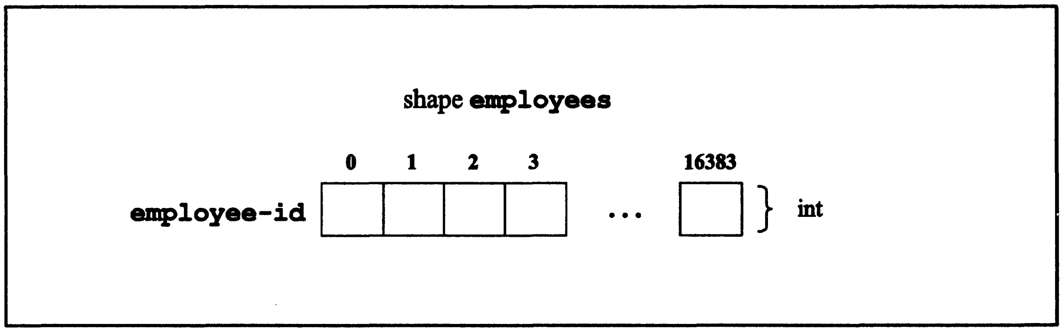

Figure 10. A parallel variable of shape **employees**

3.7.1 Declaring More Than One Parallel Variable

You can declare more than one parallel variable in the same statement, if they are of the same type. For example:

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

The parallel variables need not be of the same shape. For example:

```
unsigned int employee_id:employees, field1:image;
```

A Shortcut for Declaring More Than One Parallel Variable

If parallel variables have the same type and same shape, C* provides a more concise method for declaring them. Put the "`:`*shapename*" after the type rather than after each parallel variable. For example:

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

The parallel variables **employee_id**, **age**, and **salary** are all **unsigned ints** of shape **employees**. This syntax is generally used except when parallel variables of different shapes are being declared.

Figure 11 shows the three parallel variables that this statement creates.

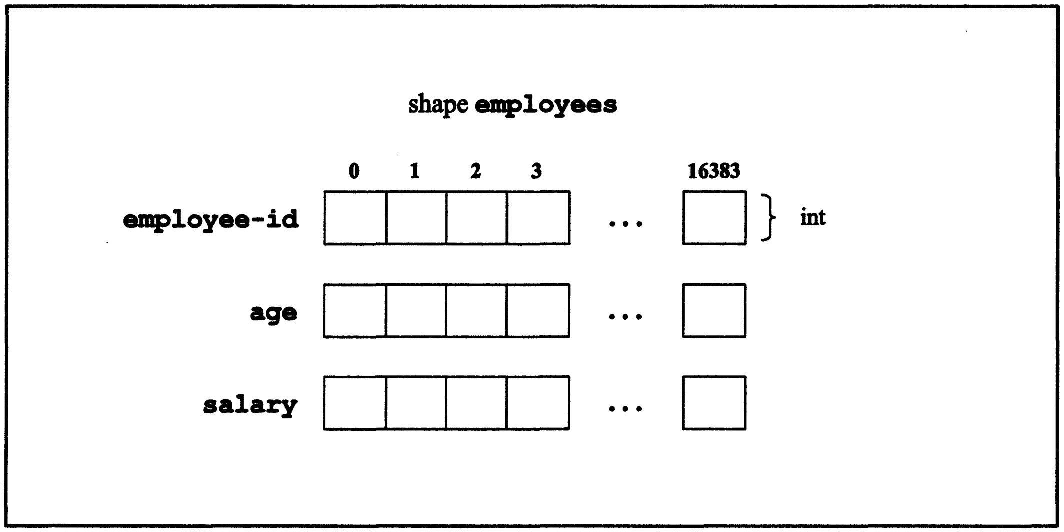

Figure 11. Three parallel variables of shape **employees**

3.7.2 Positions and Elements

As we have mentioned, a shape is a template for the creation of parallel variables. It is important to keep in mind the distinction between positions of a shape and elements of parallel variables that have been declared to be of that shape. As shown in Figure 12, elements with the same coordinates can be considered to occupy the same position in the shape. For example, the third elements of **employee-id**, **age**, and **salary** are all at position [2] of shape **employees**. These elements are referred to as *corresponding elements*. Corresponding elements are an important concept in C*.

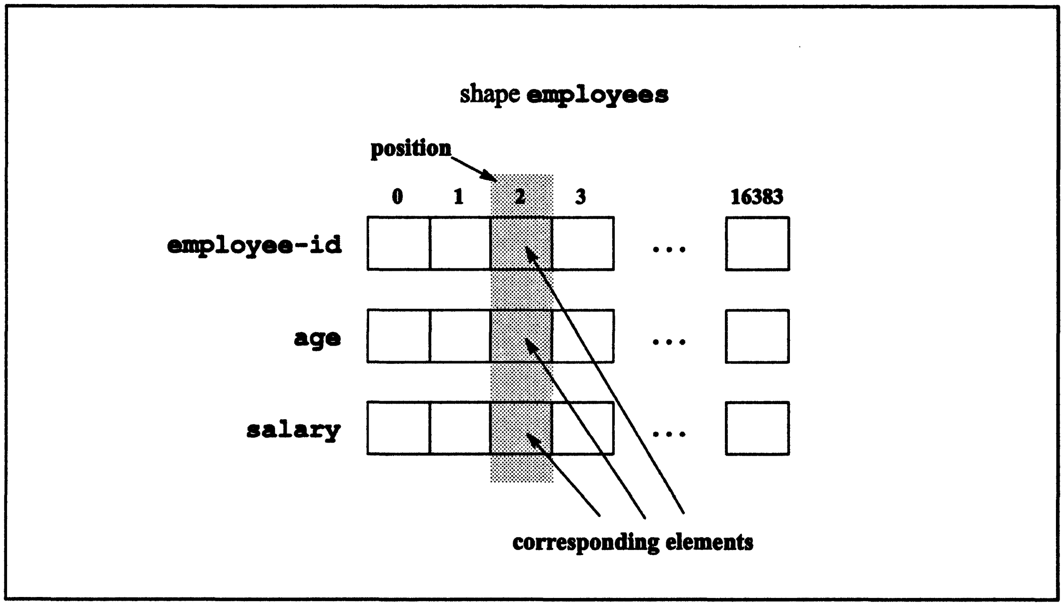

Figure 12. Corresponding elements

3.7.3 The Scope of Parallel Variables

Parallel variables follow the same scoping rules as standard scalar variables (and shapes). For example, a parallel variable declared within a function or block is local to that function or block.

NOTE: As with shape declarations, if a block contains a parallel variable declaration, you should not branch into it (for example, with a **switch** or **goto** statement); the behavior is undefined.

3.8 Declaring a Parallel Structure

You can declare an entire structure as a parallel variable. For example:

```
shape [16384]employees;
struct date {
```

```
    int month;
    int day;
    int year;
    };
struct date:employees birthday;
```

The final line of code defines a parallel variable called **birthday**. It is of shape **employees** and of type **struct date**. This parallel structure is shown in Figure 13.

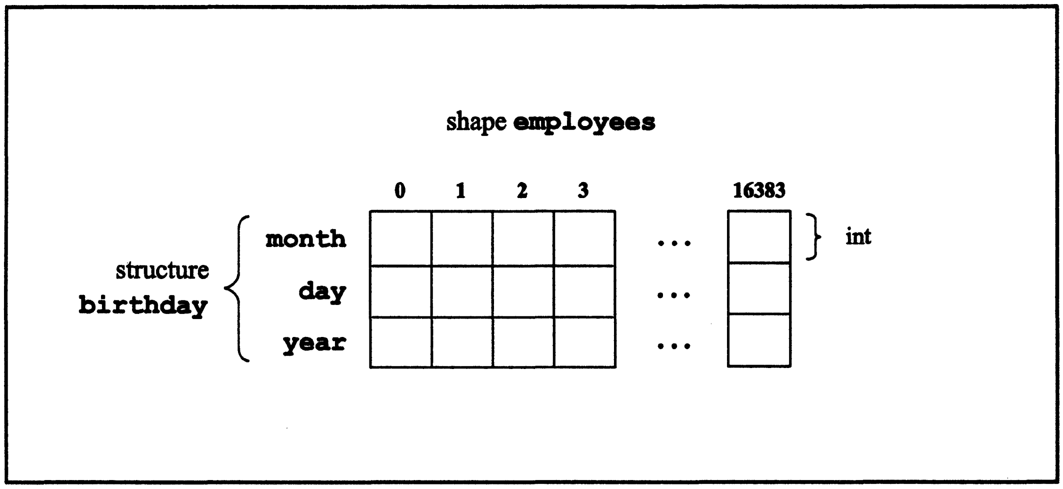

Figure 13. A parallel structure of shape **employees**

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

As with non-structured variables, you can declare more than one parallel structure in a single statement. For example:

```
struct date:employees birthday, date_of_hire;
```

You can declare parallel structures of different shapes. For example:

```
struct date birthday:employees, date_of_purchase:equipment;
```

Note the different syntax, with "*:shapename*" coming after each parallel variable.

You can also use the following syntax for declaring a parallel structure:

```
struct date {
   int month;
   int day;
   int year;
   }:employees birthday;
```

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

Some additional points about structures:

- Only scalar (that is, non-parallel) variables are allowed within parallel or scalar structures.
- Shapes are not allowed within parallel or scalar structures; a pointer to a shape is allowed within a scalar structure. (Pointers to shapes are discussed in Chapter 7.)
- You can include a scalar array within a parallel structure; you cannot include pointers of any kind.
- C*, like standard C, allows structures to be nested.

3.9 Declaring a Parallel Array

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

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

declares an array of three parallel **ints** of shape **employees**, as shown in Figure 14. **ratings[0]** specifies the first of these parallel variables, **ratings[1]** the second, and **ratings[2]** the third.

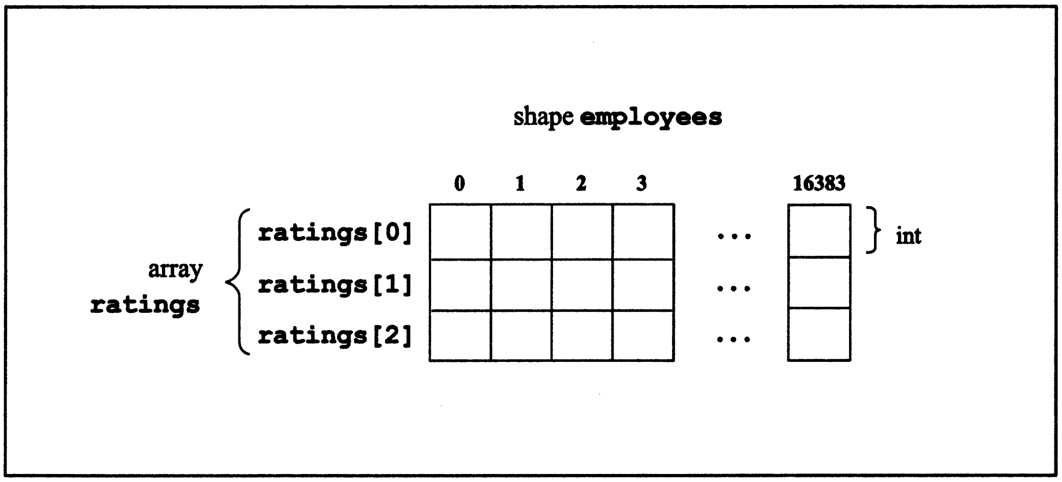

Figure 14. A parallel array of shape **employees**

Please note the difference between an *element of a parallel array* and an *element of a parallel variable*:

- An element of a parallel array, like **ratings[0]** in Figure 14, is a parallel variable. It has values for each position of its shape.
- An element of a parallel variable is scalar, and exists in only one position. **ratings[0]** consists of 16384 separate parallel variable elements.

You can also use the alternative syntax for declaring a parallel array. For example:

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

We discuss parallel arrays further in Chapter 7, where we explain their relationship to pointers.

3.10 Initializing Parallel Variables

You can initialize a parallel variable when you declare it. The initializer must be a single scalar value. Each element of the parallel variable is set to that value. For example,

```
shape [65536]ShapeA;
int:ShapeA p1 = 6;
```

sets each element of parallel variable **p1** to 6.

If the variable is an automatic, the initializer can be an expression that can be evaluated at the variable's scope. For example:

```
main()
{
   int i = 12;
   shape [65536]ShapeA;
   int:ShapeA p1 = (6+i);
}
```

sets each element of **p1** to 18.

If there is no initializer in a parallel variable declaration, and the variable has static storage duration, each element of the parallel variable is set to 0; this follows the ANSI C standard.

3.10.1 Initializing Parallel Structures and Parallel Arrays

Members of parallel structures and elements of parallel arrays can be initialized only to scalar constants; this too follows the ANSI standard.

3.11 Obtaining Information about Parallel Variables

Once you have declared a parallel variable in a program, you can obtain information about it, just as you can for a shape.

3.11.1 The positionsof, rankof, and dimof Intrinsic Functions

The **positionsof**, **rankof**, and **dimof** intrinsic functions described in Section 3.4 can be applied to parallel variables as well as to shapes. For example, if **age** is a parallel variable of shape **employees**:

- **rankof(age)** returns the rank of **employees**.
- **positionsof(age)** returns the total number of elements of **age** (and any other parallel variable of shape **employees**). Note that the number of elements of a parallel variable is the same as the number of positions in the parallel variable's shape.
- **dimof(age,0)** returns the number of instances in axis 0 of **age** (and any other parallel variable of shape **employees**).

3.11.2 The shapeof Intrinsic Function

C* includes another intrinsic function that applies only to a parallel variable. The **shapeof** intrinsic function takes a parallel variable as an argument and returns the shape of the parallel variable. For example, if a program contains the following declarations:

```
shape [16384]employees;
unsigned int:employees age;
```

shapeof(age) returns the shape **employees**.

shapeof(age) is a *shape-valued expression*; it can be used anywhere the shape name **employees** is used. For example, once **age** is declared, a subsequent declaration of a parallel variable:

```
unsigned int:employees salary;
```

could also be written:

```
unsigned int:shapeof(age) salary;
```

Similarly, a parallel structure like the one shown in Section 3.8 could be declared as follows:

```
struct date:shapeof(age) birthday;
```

3.12 Choosing an Individual Element of a Parallel Variable

As we described earlier, an individual position can be described by its coordinates along the axes of the shape. These coordinates are also used in specifying an individual element of a parallel variable. As with a shape declaration, the coordinates appear in brackets to the left of the variable name, starting with the coordinate for axis 0. These coordinates are also referred to as a *left index*.

Thus, if **age** is a parallel variable of a 1-dimensional shape named **employees**, **[0]age** specifies the first element of **age**, and **[4]age** specifies the fifth element of **age**.

For a 2-dimensional parallel variable called **pvar**,

- **[0][0]pvar** specifies the element in row 0, column 0.
- **[1][0]pvar** specifies the element in row 1, column 0.
- **[0][1]pvar** specifies the element in row 0, column 1.

and so on. Recall that axis 0 refers to the rows, and axis 1 refers to the columns.

A left index must be 0 or greater. The behavior of an operation that includes a negative left index is undefined.

You can use a left index with an element of a parallel array. For example:

```
[77]A1[4]
```

specifies the seventy-eighth parallel variable element of **A1[4]**, which is the fifth array element of the parallel array **A1**.

You can use scalar variables or expressions in place of numbers in the left index. For example, if a program contains the following declaration:

```
int j = 4;
```

the expression **[j]age** specifies the fifth instance of **age**.

It is also possible to use parallel variables or expressions in the left index. We leave that topic, however, for Chapter 10.

Chapter 4

Choosing a Shape

In Chapter 3 we described how to declare a shape, which is used as a way of organizing parallel data. You can declare more than one shape in a C* program. However, a program can in general operate on parallel data from only one shape at a time. That shape is known as the *current* shape. You designate a shape to be the current shape by using the **with** statement, which C* has added to standard C.

4.1 The with Statement

Assume a program contains the following declarations for a shape and three parallel variables of that shape:

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

Before operations can be performed on these parallel variables, **employees** must become the current shape.

To make **employees** the current shape, use the **with** statement as follows:

```
with (employees)
```

Any statement (or block of declarations and statements) following **with (employees)** can operate on parallel variables of shape **employees**. For example,

```
with (employees)
   age = 0;
```

initializes all elements of the parallel variable **age** to 0. (We discuss parallel assignment statements in the next chapter.) If each element of **salary** has been initialized to each employee's current salary, the following code:

```
unsigned int:employees new_salary;
with (employees)
   new_salary = salary*2;
```

stores twice each employee's salary in the elements of **new_salary**. (Once again, we cover arithmetic with parallel variables in the next chapter.)

You can also include operations on scalar variables inside a **with** statement. For example, you can declare a scalar variable called **sample_salary** and assign one of the values of **salary** to it:

```
with (employees) {
   unsigned int sample_salary;
   sample_salary = [0]salary;
}
```

Here is what you *can't* do inside a **with** statement:

```
shape [16384]employees, [8192]equipment;
unsigned int employee_id:employees, date_of_purchase:equipment;

main()
{
   with (employees)
      date_of_purchase = 0;    /* This is wrong */
}
```

The program cannot perform this operation on **date_of_purchase**, since this parallel variable is not of the current shape. However, the following is legal:

```
shape [16384]employees, [8192]equipment;
unsigned int employee_id:employees, date_of_purchase:equipment;

main()
{
   with (employees)
      [6]date_of_purchase = 0;    /* This is legal */
}
```

In this case, **`[6]date_of_purchase`** is scalar, since it refers to a single element. Scalar operations are allowed on parallel variables that are not of the current shape.

See Section 4.4 for a list of the situations in which a program can operate on parallel variables that are not of the current shape.

4.1.1 Default Shape

Note that the sample program in Chapter 2 included a **`with`** statement, even though only one shape was declared. You must include a **`with`** statement to perform parallel operations on parallel data, even if only one shape has been declared.

NOTE: There is no default shape in C*. However, an implementation can define a default shape. See the *C* User's Guide* for more information on default shapes.

4.1.2 Using a Shape-Valued Expression

You can use a shape-valued expression instead of a shape name to specify the current shape. For example:

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

main()
{
   with (shapeof(age))
      salary = 200;
}
```

The current shape is **`employees`**, because **`shapeof(age)`** returns the shape of the parallel variable **`age`**.

4.2 Nesting with Statements

Consider the following **with** statement:

```
with (employees)
   add_salaries();
```

where **add_salaries** is a function defined elsewhere in the program. Clearly, **employees** remains the current shape while executing the code within **add_salaries**. But what if **add_salaries** contains its own **with** statement? The new **with** statement then takes effect, and the shape it specifies becomes current. When the **with** statement's scope is completed, **employees** once again becomes the current shape.

You can therefore nest **with** statements. The current shape is determined by following the chain of function calls to the innermost **with** statement. Returning to an outer level resets the current shape to what it was at that outer level. For example:

```
shape [16384]ShapeA, [32768]ShapeB;
int:ShapeA p1, p2;
int:ShapeB q1;

main()
{
   with (ShapeA) {
      p1 = 6;
      with (ShapeB)
         q1 = 12;
      p2 = 18;
   }
}
```

Once the code in this example leaves the scope of the nested **with** statement, **ShapeA** once again becomes the current shape. The assignment to **p2** is therefore legal.

The **break**, **goto**, **continue**, and **return** statements also reset the current shape when they branch to an outer level. For example, the following code is legal:

```
with (ShapeA) {
   loop:
   /* C* code in ShapeA . . . */
   with (ShapeB) {
      /* C* code in ShapeB . . . */
      goto loop;
```

```
      }
   }
```

When the **goto** statement is executed and the program returns to **loop**, **ShapeA** once again becomes the current shape.

C* does not define the behavior when a program branches *into* the body of a nested **with** statement, however. For example, the following code results in undefined behavior:

```
goto loop;
   with (ShapeA) {
      loop:  /* This is wrong */
   }
```

4.3 Initializing a Variable at Block Scope

Section 3.10 described how to initialize parallel variables; it stated that you can initialize an automatic variable with an expression that can be evaluated at the variable's scope. Note that if the expression contains a parallel variable, the parallel variable must therefore be of the current shape. In the following code, **p2** is initialized to the values of **p1**; **p1** must therefore be of the current shape.

```
shape [16384]ShapeA;
int:ShapeA p1 = 6;

main()
{
   with (ShapeA) {
      int:ShapeA p2 = p1;
      /* ... */
   }
}
```

4.4 Parallel Variables Not of the Current Shape

As we mentioned above, there are certain situations in which a program can operate on a parallel variable that is not of the current shape. They are as follows:

- You can declare a parallel variable of a shape that is not the current shape. You cannot initialize the parallel variable using another parallel variable, however (because that involves performing an operation on the parallel variable being declared).
- As we discussed in Section 4.1, a parallel variable that is not of the current shape can be operated on if it is left-indexed by a scalar or scalars, because it is treated as a scalar variable.
- You can left-index any valid C* expression with a parallel variable of the current shape, in order to produce an lvalue or rvalue of the current shape. This topic is discussed in detail in Chapter 10.
- You can apply an intrinsic function like **`dimof`** and **`shapeof`** to a parallel variable that is not of the current shape.
- You can use the **`&`** operator to take the address of a parallel variable that is not of the current shape. See Chapter 7.
- You can right-index a parallel array that is not of the current shape with a scalar expression.
- You can use the "dot" operator to select a field of a parallel structure or union that is not of the current shape—provided that the field is not an aggregate type (for example, another structure or union).

You can also perform these operations (except for left-indexing by a parallel variable) even if there is *no* current shape—that is, outside the scope of any **`with`** statement.

Chapter 5

Using C* Operators and Data Types

C* uses all the standard C operators, plus a few new operators of its own. In addition, C* provides new meanings for the standard C operators when they are used with parallel variables. Sections 5.1–5.3 of this chapter describe C* operators and how to use them.

C* also provides a new data type, **bool**, which it adds to the standard C data types. Section 5.4 describes **bool**s.

Section 5.5 discusses parallel unions.

Throughout the chapter, variables beginning with *s* (for example, **s1**, **s2**) are scalar; variables beginning with *p* (**p1**, **p2**) are parallel.

5.1 Standard C Operators

5.1.1 With Scalar Operands

If all the operands in an operation are scalar, C* code performs exactly like standard C code. Recall that scalar variables are allocated on the front end, not on the CM. Therefore, code like this:

```
int s1=0, s2;
s2 = s1 << 2;
s1++;
s1 += s2;
```

allocates scalar variables on the front end and carries out the specified operations on them, just as in standard C.

The more interesting situations occur when a parallel operand is involved in an operation. The rest of this section considers these situations.

5.1.2 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.

Assignment with a Parallel LHS and a Scalar RHS

We have already shown examples of a parallel left-hand side (LHS) and a scalar right-hand side (RHS) with simple assignment statements, where a scalar constant is assigned to a parallel variable. For example:

```
p1 = 6;
```

In this statement, 6 is assigned to every element of the parallel variable **p1**. Technically, the scalar value is first *promoted* to a parallel value of the shape of the parallel operand, and this parallel value is what is assigned to the elements on the left-hand side.

Similarly,

```
p1 = s1;
```

causes the scalar variable **s1** to be promoted to a parallel variable, and its value is assigned to every element of parallel variable **p1**. Thus, a scalar-to-parallel assignment produces a parallel result; see Figure 15.

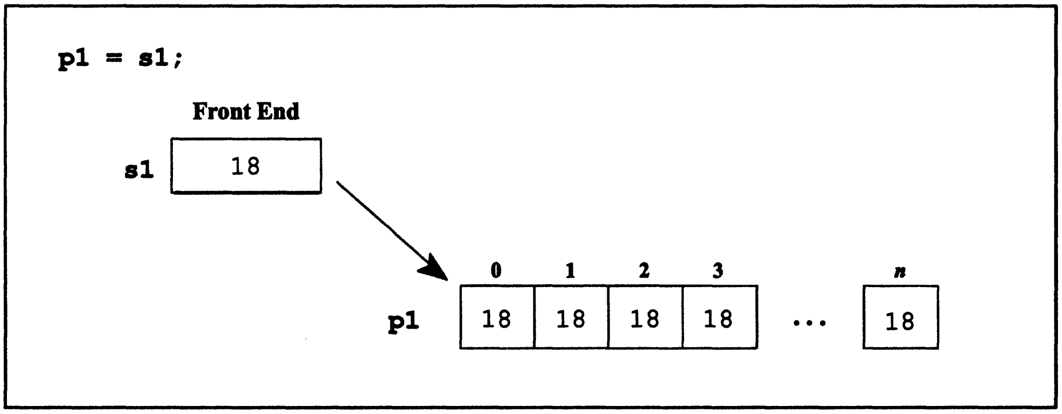

Figure 15. Promotion of a scalar variable to a parallel variable

Other binary operators work in the same way. For example,

```
p1 + s1
```

adds the value of **s1** to each element of **p1**.

```
p1 == s1
```

tests each element of **p1** for equality to the value of **s1**. For each element, it returns 1 if the values are equal, 0 if they are not equal.

```
p1 << s1
```

shifts the value of each element of **p1** to the left by the number of bits given by the value of **s1**.

```
(p1 > 2) && (s1 == 4)
```

for each element of **p1**, returns 1 if **p1** is greater than 2 and **s1** equals 4; otherwise the expression returns 0 for that element. See Chapter 6 for a further discussion of the **&&** operator when one or both of its operands is parallel.

Assignment with a Scalar LHS and a Parallel RHS

In an assignment statement, promotion occurs only when the scalar variable is on the right-hand side and the parallel variable is on the left-hand side. A scalar variable on the left-hand side is not promoted, and the following statement generates a compile-time error:

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

You can, however, explicitly *demote* the parallel variable to a scalar variable, by casting the parallel variable to the type of the scalar variable. For example:

```
int s1;
int:ShapeA p1;

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

(Parallel-to-scalar casts are discussed in more detail in Chapter 9.) But what value does C* assign, when the parallel variable could have thousands of different values?

In the case of a simple parallel-to-scalar assignment, with the parallel variable cast to the type of the scalar, C* simply chooses one value of the parallel variable and assigns that value to the scalar variable; see Figure 16. The value that is chosen is defined by the implementation.

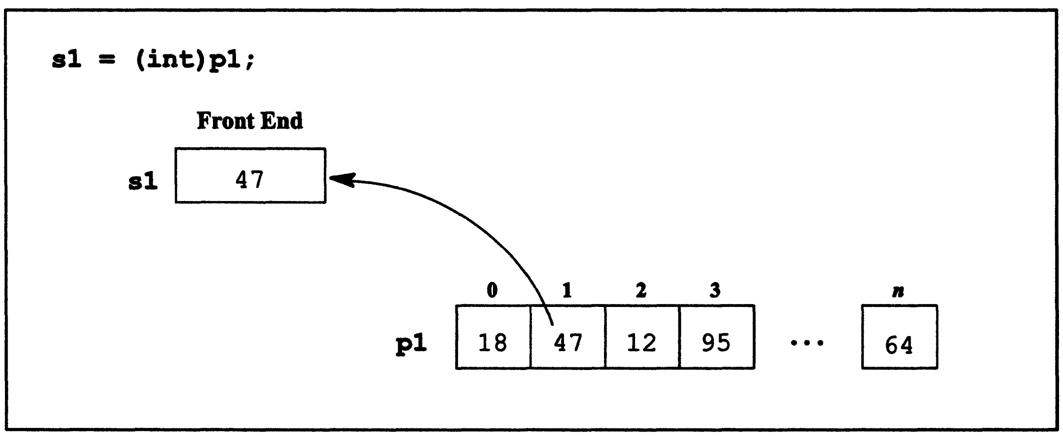

Figure 16. Selection of a value in a parallel-to-scalar assignment

What is the point of obtaining the value of an element of a parallel variable, if the language doesn't specify which value it will be? One use of demoting a parallel variable to a scalar is to cycle through all elements of a parallel variable and operate on each in turn individually; Chapter 6 has an example of this.

Note that the issues discussed here do not affect a statement like the following:

```
s1 = [2]p1;
```

This is a scalar operation. In it, an individual element of **p1** has been selected by using the left index **[2]**. Since only one element is selected, there is no possibility of a collision, and the value of the element can be assigned to **s1** without a problem.

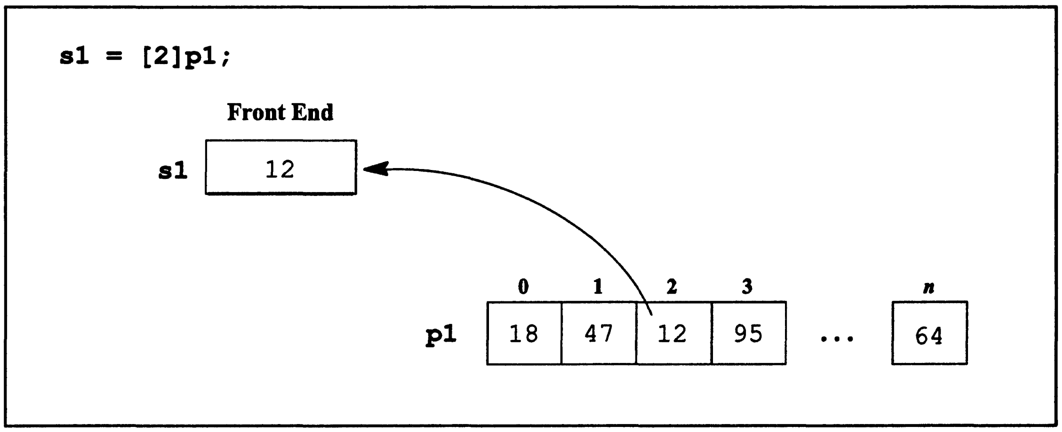

Figure 17. Assignment of a single element of a parallel variable to a scalar variable

The C compound assignment operators (for example, += and -=) have a special use with a parallel RHS and a scalar LHS; they are discussed in Section 5.3.

5.1.3 With Two Parallel Operands

Standard binary C operators can work with two parallel operands, if both are of the current shape. For example,

```
p2 = p1;
```

assigns the value in each element of **p1** to the element of **p2** that is at the same position—that is, to the *corresponding element* of **p2**; see Figure 18.

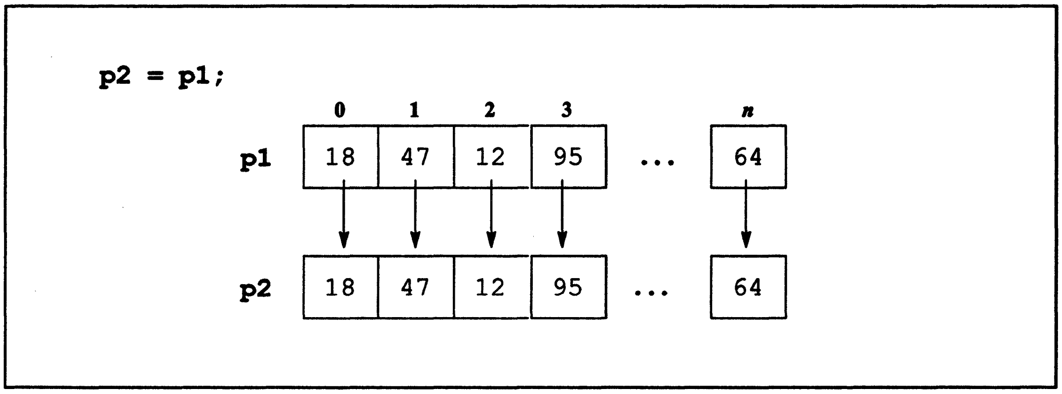

Figure 18. Assignment of a parallel variable to a parallel variable

```
p1 * p2
```

multiplies each element of **p1** by the corresponding element of **p2**.

```
p1 >= p2
```

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

```
(p1 > 2) || (p2 < 4)
```

returns, for each element, 1 if **p1** is greater than 2 or **p2** is less than 4, and 0 otherwise. Both operands are evaluated if either is parallel. See Chapter 6, however, for a further discussion of this operator and the **&&** operator.

5.1.4 Unary Operators for Parallel Variables

Standard C unary operators can be applied to parallel variables. For example:

```
p1++
```

increments the value in every element of the parallel variable **p1**.

```
!p1
```

provides the logical negation of each element of **p1**. If the value of the element is 0, the expression returns 1; if the value of the element is nonzero, the expression returns 0.

5.1.5 The Conditional Expression

The ternary conditional expression **?:** operates in slightly different ways depending on the mix of parallel and scalar variables in the expression.

For example, in the following statement:

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

the first operand is scalar, and the other two operands are parallel. The interpretation of this statement is relatively straightforward: if the scalar variable **s1** is less than 5, the value in each element of the parallel variable **p2** is assigned to the corresponding element of **p1**; if **s1** is 5 or greater, the value in each element of **p3** is assigned to **p1**. All the parallel variables must be of the current shape.

In the following statement:

```
p1 = (s1 < 5) ? p2 : s2;
```

the first operand and one of the other operands are scalar. In this case, **s2** is promoted to a parallel variable of the current shape, and the expression is evaluated in the same way as the previous example.

What happens if the first operand is parallel? For example:

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

In this case, each element of **p2** is evaluated separately. If the value in **p2** is less than 5 in a particular element, the value of **p3** is assigned to **p1** for the corresponding element. Otherwise, the value of **p4** is assigned to **p1**. Figure 19 gives an example of this; the arrows in the figure show examples of the data movement, based on the value of **p2**.

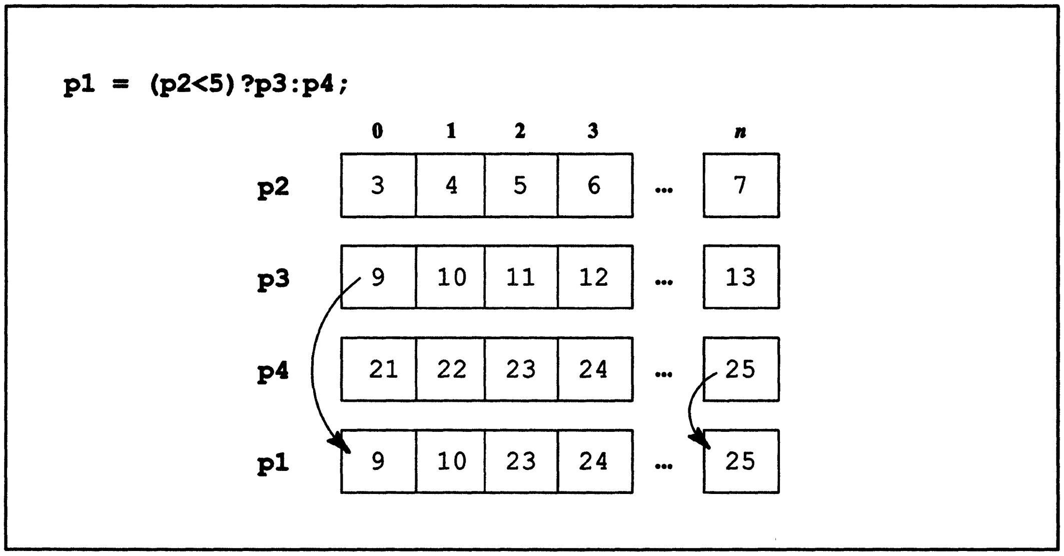

Figure 19. Use of the conditional operator with parallel variables

If either or both of the operands (other than the first) were scalar in this example, they would be promoted to parallel in the current shape, and the expression would be evaluated in the same way.

Both operands are evaluated if the condition is parallel.

See Chapter 6 for a further discussion of this operator.

5.2 New C* Operators

C* adds several new operators to standard C.

5.2.1 The <? and >? Operators

The **<?** and **>?** operators provide, respectively, the minimum and maximum of two variables. These operators are typically expressed as macros in standard C. For example, the C macro

```
( ((a) < (b)) ? (a) : (b) )
```

is similar to

```
a <? b;
```

in C*, except that C* evaluates the operands only once.

There are also assignment operator versions of **<?** and **>?**. For example,

```
s1 >?= s2;
```

assigns the value of **s2** to **s1** if the value is greater than the value of **s1**; otherwise **s1** is unchanged.

The minimum and maximum operators follow standard C rules for type conversions and compatibility. For example, if one operand is a **float** and the other is an **int**, the **int** is promoted to a **float**.

These operators can be used with parallel as well as scalar variables. For example,

```
p1 <?= p2;
```

assigns the lesser of **p1** and **p2** to **p1**, for every pair of corresponding elements of these parallel variables.

The minimum and maximum operators are discussed further in Section 5.3.

5.2.2 The %% Operator

The new **%%** operator provides the modulus of its operands. It is patterned after the standard C **%** operator; for example, it has the same precedence and associativity, accepts and returns the same types, and performs the same conversions. It also gives the same answer when both of its operands are positive—the answer is the remainder when the first operand (the numerator) is divided by the second operand (the denominator). For example, the following statements are both true:

```
(8 % 6) == 2
(8 %% 6) == 2
```

The difference between the two occurs when one or both of the operands is negative. In that case, different implementations of `%` can give different answers. For example, the sign of the answer can be either positive or negative.

`%%` does the following when one or both of the operands is negative:

- It divides the first operand by the second operand. If the result is not an integer, it converts this result to the next lower integer. For example, the result of dividing 17 by –4 is –4.25, so `%%` converts this to –5, because –5 is smaller than –4.
- It multiplies the second operand by this result. In the above example, –5 * –4 is 20.
- It subtracts *that* result from the first operand. The answer is the result of the operation. In our example, 17 minus 20 is –3. Therefore:

```
(17 %% -4) == -3
```

A consequence of this procedure is that the result always has the same sign as that of the second operand. For example:

```
(-17 %% 4) == 3
(17 %% 4) == 1
(-17 %% -4) == -1
```

The `%%` operator is discussed further in Section 10.3.2.

5.3 Reduction Operators

Standard C has several compound assignment operators, such as +=, that perform a binary operation and assign the result to the LHS. 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.

The following code presents a parallel-to-scalar reduction assignment.

```
#include <stdio.h>

shape [16384]employees;
unsigned int:employees salary;

main()
{
  unsigned int payroll=0;

/* Initialization of salary omitted */

   with (employees)
      payroll += salary;

   printf ("Total payroll is $%d.\n", payroll);
}
```

In this code, the **+=** operator sums the value in each element of **salary** and adds this sum to the scalar variable **payroll**, as shown in Figure 20. Note that the scalar variable on the left-hand side is included in the operation; that is why **payroll** must be initialized to 0.

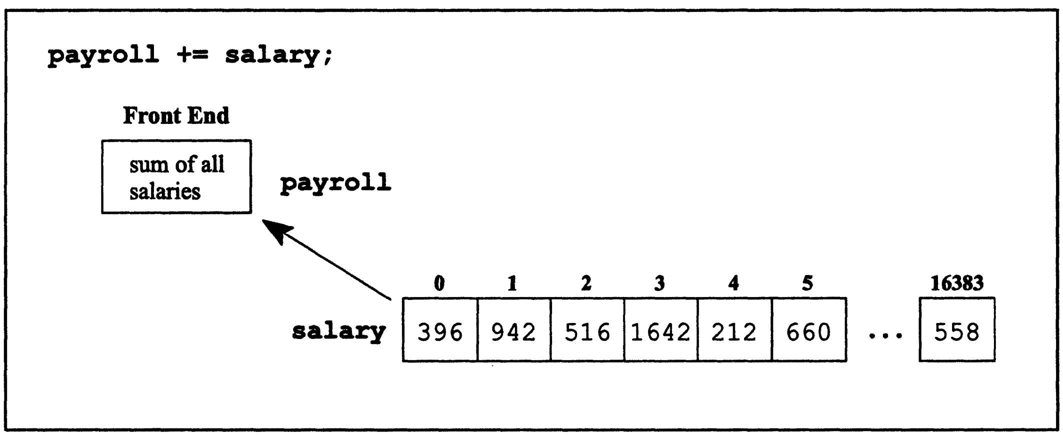

Figure 20. A reduction assignment

5.3.1 Unary Reduction

As the sample code shows, binary reduction assignment operators include the left-hand side as one of their operands, so you must initialize the variable on the left-hand side appropriately. You can also use any of these operators as a unary operator with a parallel operand. We can therefore simplify the sample code by eliminating the scalar variable and revising the **printf** statement as follows:

```
printf("Total weekly payroll is $%d.\n", +=salary);
```

5.3.2 Parallel-to-Parallel Reduction Assignment

The left-hand side of a reduction assignment can be an individual element of a parallel variable, instead of a front-end scalar variable. For example,

```
shape [16384]employees;
unsigned int:employees salary, payroll=0;

main()
{
   /* Initialization of salary omitted */

   with (employees)
      [0]payroll += salary;
}
```

declares **payroll** to be a parallel variable, and puts the total of the **salary** values into element [0] of **payroll**.

5.3.3 List of Reduction Operators

Table 1 lists the C* reduction operators. All can be used for parallel-to-scalar reduction assignment, parallel-to-parallel reduction assignment, and unary reduction.

Table 1. Reduction assignment operators

Operator	Meaning
+=	Sum of values of parallel variable elements
−=	Negative of the sum of values
&=	Bitwise AND of values
^=	Bitwise XOR of values
|=	Bitwise OR of values
<?=	Minimum of values
>?=	Maximum of values

Note that simple parallel-to-scalar assignment using a cast is also a form of reduction assignment; see page 46.

Note also that the C compound operators ***=**, **/=**, **%=**, **<<=**, and **>>=** cannot be used as C* reduction assignment operators.

We have already discussed the **+=** operator; now let's look at the other reduction operators.

5.3.4 The −= Reduction Operator

When used as a binary operator, **−=** subtracts the sum of the parallel RHS's values from the scalar LHS, and assigns the result to the LHS. Therefore,

```
s1 -= p1;
```

is equivalent to the following:

```
s1 = (s1 - (+=p1));
```

Initialize the scalar LHS to 0 to obtain the negative of the sum of the parallel variable's values.

5.3.5 Minimum and Maximum Reduction Operators

The **<?=** and **>?=** operators can be used as unary operators to obtain the minimum and maximum values in all elements of a parallel variable. To find out the lowest and highest salaries in the parallel variable **salary**, for example, add the following **printf** statements to the code example shown on page 53:

```
printf ("The lowest salary is $%d.\n", <?=salary);
printf ("The highest salary is $%d.\n", >?=salary);
```

Note once again that, when used as binary operators, **<?=** and **>?=** include the left-hand side as an operator. To assign the lowest value of a parallel variable to a scalar variable, therefore,

```
s1 <?= p1;
```

might not work, since **s1** might be the lowest value. Instead, use **<?=** as a unary operator, and use = to assign the result to the scalar variable. For example:

```
s1 = <?=p1;
```

5.3.6 Bitwise Reduction Operators

The bitwise reduction assignment operators mask all elements of a parallel variable, as described in the subsections below.

Bitwise OR

The |= operator performs a bitwise OR of all elements of a parallel variable. For example, in this statement:

```
s1 |= p1;
```

all elements of **p1** are first bitwise OR'd; if a particular bit is a 1 in any element, that bit is a 1 in the result. This result is then bitwise OR'd with **s1**, and the result is assigned to **s1**.

Bitwise OR is particularly useful in testing if any elements of a parallel variable meet a condition. The **if** statement in C* works in the same way as the **if** statement in standard

C: if the condition expression evaluates to 0, then the statement following is not executed; if the condition expression is non-zero, the statement is executed. In the following code,

```
if (|=(p1 > 5))
   p2 = 10;
```

if there are any elements of **p1** greater than 5, the condition expression is non-zero, and 10 is assigned to each element of **p2**. If there are no elements of **p1** greater than 5, the bitwise OR evaluates to 0, and the following statement is not executed.

Bitwise AND

In a bitwise AND, if a particular bit is a 0 in any element of the specified parallel variable, that bit is a 0 in the result. Bitwise AND provides a way to test whether all elements of a parallel variable meet a condition. In the following code:

```
if (&=(p1 > 5))
   p2 = 10;
```

each element of **p2** is set to 10 only if all elements of **p1** have values greater than 5.

Bitwise Exclusive OR

You can view the bitwise exclusive OR operator as operating pair-wise on elements of a parallel variable. For example, if three parallel bit-fields each contain a 1, bitwise exclusive OR first operates on two of them: the two 1 bits yield a 0 bit. This 0 bit is then exclusive OR'd with the remaining 1 bit, and the result is a 1 bit. In general, the result of a bitwise exclusive OR operation is 1 if the corresponding bit is 1 in an *odd* number of elements; it is 0 if the corresponding bit is 1 in an *even* number of elements. Note that in a reduction assignment the scalar LHS is included in this calculation.

5.3.7 Reduction Assignment Operators with a Parallel LHS

Reduction assignment operators can be used with a parallel LHS when the parallel variable is left-indexed with a parallel subscript. This topic is discussed in Section 10.1.5.

5.4 The bool Data Type

In addition to parallelism, the CM has one other major difference from other computers: it aligns data on bit, rather than byte, boundaries. C* introduces a new data type, **bool**, that allows you to take advantage of this in allocating CM memory. Typically, **bools** are used as parallel variables to store flags.

The **bool** is an unsigned single-bit integral data type. The actual size and alignment of a **bool** are implementation-dependent: on the CM-2 it occupies one bit of memory and is aligned on a bit boundary; on the front end it is stored as a **char**. It behaves as a single-bit quantity, however, no matter what its actual size is.

When you cast a variable of a larger data type to a **bool**, the expression has logical (rather than arithmetic) behavior. That is, if the value of the larger data type is 0, 0 is the result; if the value is non-zero, 1 is the result. Thus:

```
int i=0, j=4;
printf("%d\n", (bool)i);   /* prints "0" */
printf("%d\n", (bool)j);   /* prints "1" */
```

Also note the following behavior:

```
int i, j=1, k=1;
bool:current b;
i = j + k;   /* i=2 */
b = j + k;   /* b=1 */
```

All elements of **b** are assigned the value 1 because the value of the expression **(j + k)** is non-zero.

A **bool**, like a **char**, is promoted to an **int** when used as an operand of most operators. Thus, performing operations on **bools** could be slower than performing the same operations on larger data types. The compiler in some cases avoids this promotion, however, by following this rule: An expression is evaulated at the precision of the variable to which it is assigned, as long as the results are the same as if standard ANSI promotion rules had been followed. For example, if **a**, **b**, and **c** are all **bools**, this statement:

```
a = b | c;
```

is evaluated at **bool** precision, because the expression is assigned to a **bool**. However, in the following code:

```
where (b | c == 0) {
   /* ... */
}
```

the expression is evaluated at **int** precision, because it is not explicitly stored anywhere.

5.4.1 The boolsizeof Operator

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

With a Parallel Variable or Data Type

When a parallel variable is used as the operand, **boolsizeof** returns the number of bits a single element of the variable occupies in CM memory. For a parallel data type, **boolsizeof** returns the number of bits that must be allocated for a single instance of the data type. For example,

```
boolsizeof(int:ShapeA); /* Size in bools of a parallel int */
```

returns 32 in the current implementation.

With a Scalar Variable or Data Type

When a scalar variable is used as the operand, **boolsizeof** returns the number of *bytes* that the variable occupies in front-end memory (because a **bool** is stored as a **char** on the front end). It therefore gives the same result as the **sizeof** operator when applied to a scalar operand. For example,

```
boolsizeof(int);  /* Size in bytes of a front-end int */
```

returns 4 in the current implementation.

Note the difference in result between **boolsizeof** when applied to a parallel operand and **boolsizeof** when applied to a scalar operand.

5.5 Parallel Unions

You can create parallel unions. Like parallel structures, they can only contain scalar variables. For example, the following code:

```
union ptype {
      int i;
      float f;
};

union ptype:ShapeA p1;
```

defines a parallel variable **p1** of shape **ShapeA** and of the union type **ptype**. The following initializes **p1** as an integer:

```
p1.i = 50;
```

Each element of **p1** is an **int** containing the value 50.

The following initializes **p1** as a **float** containing the value 89.7:

```
p1.f = 89.7;
```

Unions can also appear within structures, as in standard C.

5.5.1 Limitations

The current implementation of parallel unions has the following limitations:

- You cannot use an initializer to initialize a parallel union or any object containing a parallel union.
- You cannot assign a scalar union to a parallel union.
- You cannot promote a scalar union to be parallel (for example, by a scalar-to-parallel cast; see Chapter 9).
- You cannot demote a parallel union to be scalar.

Chapter 6

Setting the Context

In Chapter 4, we discussed how to use the **with** statement to select a current shape. Once there is a current shape, a program can perform operations on parallel variables that have been declared to be of that shape.

But what if you want an operation to be performed only on certain elements of a parallel variable? For example, you have a database containing the physical characteristics of a population, and you want to know the average height of people who weigh over 150 pounds. To do this, specify which positions are *active* by using a **where** statement, which C* has added to standard C. Code in the body of a **where** statement operates only on elements in active positions. Using **where** to specify active positions is known as *setting the context.*

6.1 The where Statement

When a **with** statement first selects a shape, all positions of that shape are active; code in the body of the **with** statement operates on every element of a parallel variable. A **where** statement selects a subset of these positions to remain active. For example, the following code:

```
with (population)
   where (weight > 150.0) {
      /* ... */
   }
```

selects only those positions of shape **population** in which the value of parallel variable **weight** is greater than 150. (This assumes that the elements of **weight** have previously been initialized to some values.) Parallel code in the body of the **where** statement applies only to those positions. Figure 21 shows the effect of the **where** statement.

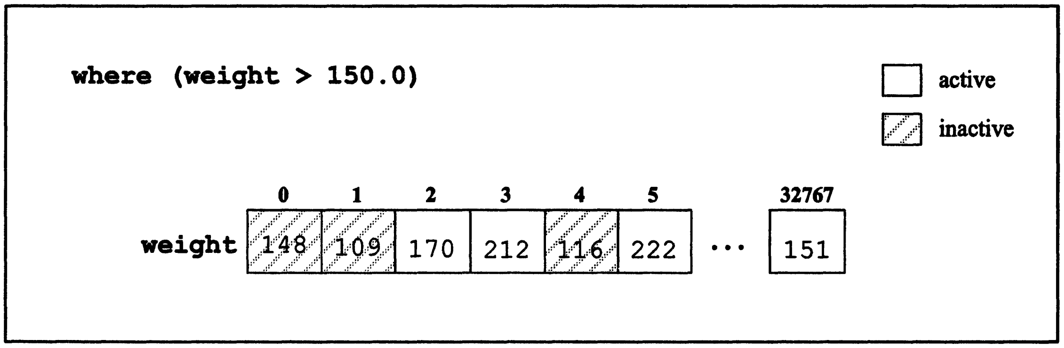

Figure 21. Using **where** to restrict the context

In the figure, positions 0, 1, and 4 become inactive in the body of the **where** statement; positions 2, 3, 5, and 32767, all of which have weights over 150, remain active.

The controlling expression that **where** evaluates to set the context must operate on a parallel operand of the current shape. (Other controlling expressions—for example, for the **while** and **if** statements—operate only on scalar variables.) Like other controlling expressions, it evaluates to 0 or non-zero, but it does so separately for each parallel variable element that is currently active.

The following code calculates the average height of people weighing over 150 pounds (assuming that the values of **height** and **weight** have been initialized):

```
shape [32768]population;
float:population weight, height;
unsigned int:population count;
float avg_height;

main()
{
   /* Code to initialize height and weight omitted. */

   with (population) {
      count = 1;
      where (weight > 150.0)
         avg_height = (+=height / +=count);
   }
}
```

NOTE: There is a slightly easier way of obtaining the number of active positions than the one shown in in this code fragment; it involves a *scalar-to-parallel cast.* For example:

```
(int:population)1
```

promotes 1 to a parallel variable of shape **population**. Using the **+=** operator on this variable produces the number of active positions. Scalar-to-parallel casts are discussed in Section 9.6.1.

Like the **with** statement, a **where** statement can include scalar as well as parallel code within its body, and there are the same restrictions on operating on parallel variables that are not of the current shape; see Section 6.5 for a discussion of what happens to scalar and parallel code when a **where** statement causes no positions to remain active.

The context set by the **where** statement remains in effect for any procedures called within its body. Once the body of the **where** statement has been exited, however, the context is reset to what it was before the **where** statement. For example, if we add two statements to the code fragment above:

```
with (population) {
   float avg_weight;
   count = 1;
   where (weight > 150.0)
      avg_height = (+=height / +=count);
   avg_weight = (+=weight / +=count);
}
```

avg_weight is assigned the average weight for all positions of shape **population**, not just for the positions where **weight** is greater than 150.

6.1.1 The else Clause

Like **if** statements in standard C, **where** statements can include an **else** clause. The **else** following an **if** says: *Perform the following operations if the* **if** *condition is not met.* The **else** following a **where** says: *Perform the following operations on positions that were made inactive by the* **where** *condition.* It "turns on" all of the positions that were "turned off" by the **where** condition, and turns off all the positions that the **where** condition left on. Figure 22 shows the effect of an **else** clause on the set of active positions in Figure 21.

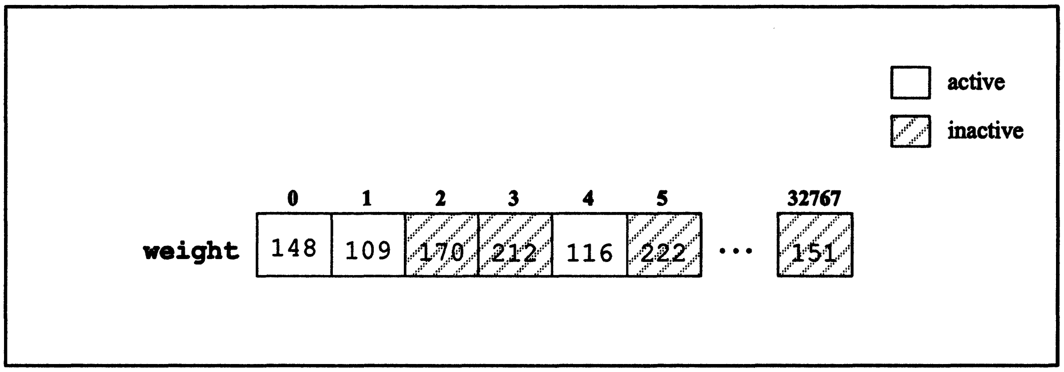

Figure 22. The effect of **else** on the context shown in Figure 21

The following code calculates separate average heights for those weighing more than 150 pounds, and for those weighing 150 pounds or less:

```
shape [32768]population;
float:population weight, height;
unsigned int:population count;
float avg_height_heavy, avg_height_light;

main()
{
   with (population) {
      count = 1;
      where (weight > 150.0)
         avg_height_heavy = (+=height / +=count);
      else
         avg_height_light = (+=height / +=count);
   }
}
```

6.1.2 The where Statement and positionsof

Using **where** to restrict the context does not affect the value returned by the **positionsof** intrinsic function. **positionsof** returns the total number of positions in a shape, not the number of active positions.

6.1.3 The where Statement and Parallel-to-Scalar Assignment

In Chapter 5 we discussed assigning a parallel variable to a scalar variable: you must cast the parallel variable to the type of the scalar variable. The operation then chooses (in an implementation-defined way) one value of the parallel variable and assigns it to the scalar variable. If a **where** statement restricts the context, however, the value chosen is from one of the active positions.

6.2 The where Statement and Scalar Code

As we noted above, you can include scalar code within the scope of a **where** statement. So, for example, the following is legal:

```
shape [32768]population;
float:population weight;
float avg_height;

main()
{
   with (population) {
      where (weight > 150.0)
         avg_height = 0;
   }
}
```

Recall that an element of a parallel variable is considered to be scalar. That means you can perform operations on an element *even if its position is inactive*. For example, if position 0 becomes inactive when we choose positions where **weight** is over 150, we can still do the following:

```
shape [32768]population;
float:population weight;
unsigned int:population count;

main()
{
   with (population) {
      count = 1;
      where (weight > 150.0) {
```

```
            [0]weight = 225;     /* These are all legal. */
            [0]weight = [1]weight;
            [0]count += count;
        }
    }
}
```

Note the final statement in this code fragment. In it, the values of the active elements of **count** are summed; this sum does not include the value of **[0]count**, because position [0] became inactive as a result of the **where** statement. However, the result of the sum *can* be placed in **[0]count**, because **[0]count** is scalar. Thus:

- You can read from or write to an individual parallel variable element in an inactive position.
- An element in an inactive position is not included in operations on the parallel variable as a whole.

6.3 Nesting where and with Statements

6.3.1 Nesting where Statements

You can nest **where** statements. The effect is to continually shrink the set of active positions. For example, we might want to calculate average heights separately for males and females weighing over 150 pounds in the **population** data base. Let's add a parallel variable called **sex**, therefore, and assume that it has been initialized: 0 for females and 1 for males. The following code would then produce the desired results.

```
shape [32768]population;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height;

main()
{
    with (population) {
        count = 1;
        where (weight > 150.0) {
            where (sex)
```

```
            avg_male_height = (+=height / +=count);
         else
            avg_female_height = (+=height / +=count);
      }
   }
}
```

6.3.2 Nesting with Statements

It is also possible to choose another shape within the body of a **where** statement. For example:

```
shape [32768]population, [16384]employees;
int:employees salary;
int payroll;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height;

main()
{
   with (population) {
      count = 1;
      where (weight > 150.0) {
         where (sex)
            avg_male_height = (+=height / +=count);
         with (employees)
            payroll += salary;
      }
   }
}
```

Since each shape has a different set of positions, the context established by a **where** statement for one shape has no effect on the context of expressions in another shape. Therefore, the statement

```
payroll += salary;
```

in the code example above uses the entire set of positions of shape **employees**. Of course, we could add another **where** statement to set the context for the nested **with** statement.

Once control leaves the body of the nested **with** statement, the context returns to whatever it was before the **with** statement was executed. For example:

```
with (population) {
    count = 1;
    where (weight > 150.0)
        where (sex) {
            avg_male_height = (+=height / +=count);
            with (employees)
                payroll += salary;
        }
        else
            avg_female_height = (+=height / +=count);
}
```

When **population** becomes the current shape for the second time, the context is once again the positions where **weight** is greater than 150 and **sex** is 0.

With nesting, it is therefore possible to switch back and forth between shapes and maintain separate contexts for each.

6.3.3 The break, goto, continue, and return Statements

Section 4.2 described the behavior of **break**, **goto**, **continue**, and **return** statements in nested **with** statements. They behave similarly for nested **where** statements. Specifically:

- Branching to an outer-level **where** statement resets the context to what it was at that level.
- The behavior of branching into a nested **where** statement is not defined. Don't do it.

The behavior of functions that contain nested **where** statements is discussed in Section 8.1.2.

6.4 The everywhere Statement

A **where** statement can never increase the number of active positions for a given shape; nesting **where** statements has the effect of creating smaller and smaller subsets of the original set of active positions. C* does, however, provide an **everywhere** statement that allows operations on all positions of the current shape, no matter what context has been set by previous **where** statements.

For example, in the following code:

```
shape [32768]population;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height, avg_height;

main()
{
   with (population) {
      count = 1;
      where (weight > 150.0) {
         where (sex)
            avg_male_height = (+=height / +=count);
         else
            avg_female_height = (+=height / +=count);

         everywhere
            avg_height = (+=height / +=count);
      }
   }
}
```

the scalar variable **avg_height** is assigned the average height for all positions of shape **population**, even though this average is calculated within the body of a **where** statement that deactivates some positions of **population**.

After the **everywhere** statement, the context returns to what it was before **everywhere** was called. In this case, once again only positions where **weight** is greater than 150 are active.

Note that if **avg_height** had been calculated after the body of the **where** statement, the **everywhere** statement would not have been needed, since the context reverts to what it was before the **where** statement. In this case, all positions of shape **population** become active once again.

As with the **where** statement, branching from an **everywhere** statement to an outer level via a **break, goto**, **continue**, or **return** statement resets the context to what it was at the outer level. The behavior of branching into an **everywhere** statement is not defined.

6.5 When There Are No Active Positions

What happens when the controlling expression of the **where** statement leaves no positions active? Consider the situation shown in Figure 23.

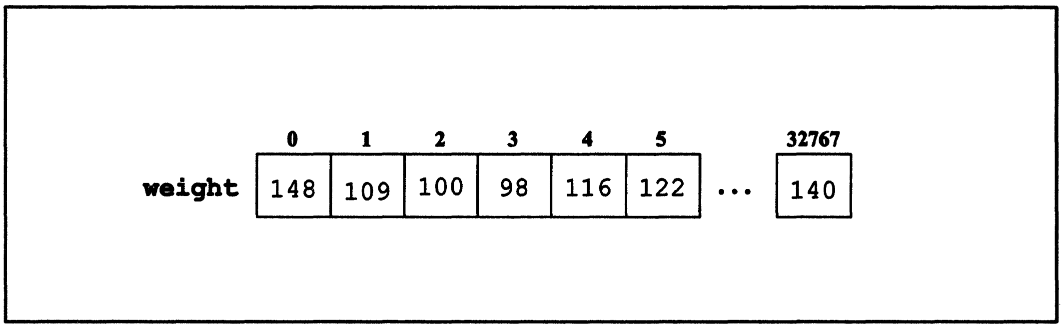

Figure 23. A shape where all weights are less than 150

If **population** is initialized entirely with values of 150 and below, the following code makes all positions inactive, since no position has **weight** greater than 150:

```
with (population)
   where (weight > 150.0) {
      /* ... */
   }
```

Code is still executed in this situation, but an operation on a parallel variable of the current shape has no result. For example,

```
weight++;
```

does not increment any of the values of **weight**, because no elements of **weight** are active.

But note that operations on individual elements do have results, since they are scalar. For example,

```
[0]weight = 225;
```

assigns 225 to element [0] of **`weight`**, even though no positions are active.

The result of a parallel-to-scalar assignment using = is undefined when no positions are active.

The results of reduction assignment operations are discussed below.

6.5.1 When There Is a Reduction Assignment Operator

Unary Reduction Operators

Consider the following code fragment, where **`maximum`** is a scalar variable, and **`weight`** is a parallel variable:

```
where (weight > 150.0)
   maximum = (>?=weight);
```

If there are no active positions, what gets assigned to **`maximum`**?

C* provides default values for unary reduction operators when there are no active positions. These values are listed in Table 2.

The values in Table 2 are basically identities for the operations. For example, the result of a += operation (when no positions are active) added to the result of another += operation gives the result of the other operation.

Table 2. Values of unary reduction operators when there are no active positions

Unary Reduction Operator	Value
+=	0
-=	0
&=	~0 (all one bits)
^=	0
|=	0
<?=	maximum value representable
>?=	minimum value representable

Binary Reduction Assignment Operators

Recall that the left-hand side is included in binary reduction assignments. When there are no active positions, and a binary reduction assignment operator is used, the LHS remains unchanged.

6.5.2 Preventing Code from Executing

Of course, you might not want scalar code, or code in another shape, to execute if there are no positions active. To keep the code from executing, use an **if** statement with a bitwise OR reduction operator to conditionalize the entire **where** statement. For example:

```
if (|=(weight > 150.0))
   where (weight > 150.0) {
      float avg_height = 0;
      /* ... */
}
```

In this code fragment, the scalar variable **avg_height** is declared and initialized only if there are any positions with **weight** greater than 150. See Section 5.3.6 for a discussion of using the bitwise OR reduction operator in an **if** condition.

If the condition in the **if** statement has side effects, more code is required to ensure that the condition is evaluated only once. Do the following:

1. Create a temporary parallel variable of the current shape.
2. In the **if** condition, assign to this temporary variable the results of the parallel expression you would otherwise have evaluated in the **where** statement, and perform a bitwise OR reduction of the temporary variable.
3. Have **where** evaluate the temporary variable.

For example:

```
with (population) {
   unsigned int:population temporary = 0;
   if (|=(temporary = (++weight > 150.0)))
      where (temporary) {
         float avg_height = 0;
         /* ... */
      }
}
```

6.6 Looping through All Positions

Some of the C* features we have discussed so far can be used to loop through all positions of a shape, allowing operations to be performed on each position separately.

For example, consider a database initialized as shown in Figure 24. Note that each position has a unique identifier, **case_no**.

shape population

	0	1	2	3	4	5		32767
case_no	0	1	2	3	4	5	...	32767
weight	148	109	100	212	200	122	...	140
height	62	58	60	72	75	68	...	66

Figure 24. A database

The following code picks a case of shape **population**, prints the weight and height of its corresponding elements, then picks another case, until all cases have been chosen.

```
#include <stdio.h>

shape [32768]population;
unsigned int:population case_no, weight, height;
unsigned int index;

/* Code to initialize parallel variables omitted. */

main()
{
   with (population) {
      bool:population active;
      active = 1;
      while (|= active) {
         where (active) {
            index = (unsigned int)case_no;
            where (index == case_no) {
               printf ("Height is %d; weight is %d.\n",
                       [index]height, [index]weight);
               active = 0;
            }
         }
      }
```

```
        }
    }
```

In this program, a **while** loop with a bitwise OR reduction controls the selection of positions. The = operator chooses a value of **case_no** and stores it in **index** (note the use of the cast to explicitly demote the parallel variable to a scalar variable). The inner **where** expression then selects the position that contains this value for **case_no**. (There will only be one, because each value of **case_no** is unique.) Since each value of **case_no** corresponds to the coordinate of its position, we can use that value (now assigned to **index**) as a left index for the other parallel variables in order to choose an element of them for printing.

At the end of the **where** statement, **active** is set to 0 for the active position, turning it off for the next iteration of the loop. When all the positions have been selected, all the positions will have been turned off. At this point the controlling expression of the **while** loop evaluates to false, and the program completes.

NOTE: A more efficient way of doing this is to use the **pcoord** function, which is described in Section 10.2.

6.7 Context and the ||, &&, and ?: Operators

6.7.1 || and &&

The **||** and **&&** operators perform implicit contextualization when one or both of their operands are parallel. (Recall that if one operand is parallel and the other is scalar, the scalar operand is promoted to parallel.)

Consider the following statement, in which all variables are parallel:

```
p3 = (p1 > 5) && (p2++);
```

Since at least one of the **&&** operands is parallel, we get the parallel version of the operator. This statement does two things:

- First, in each position, it assigns a 1 to the corresponding element of **p3** if both operands evaluate to non-zero ("TRUE"), and assigns a 0 otherwise.

- Second, it increments **p2** in each position where **p1** is greater than 5—that is, where the left operand evaluates to TRUE. In positions where the left operand evaluates to 0, **p2** is unchanged.

Figure 25 shows how the statement works with some sample values.

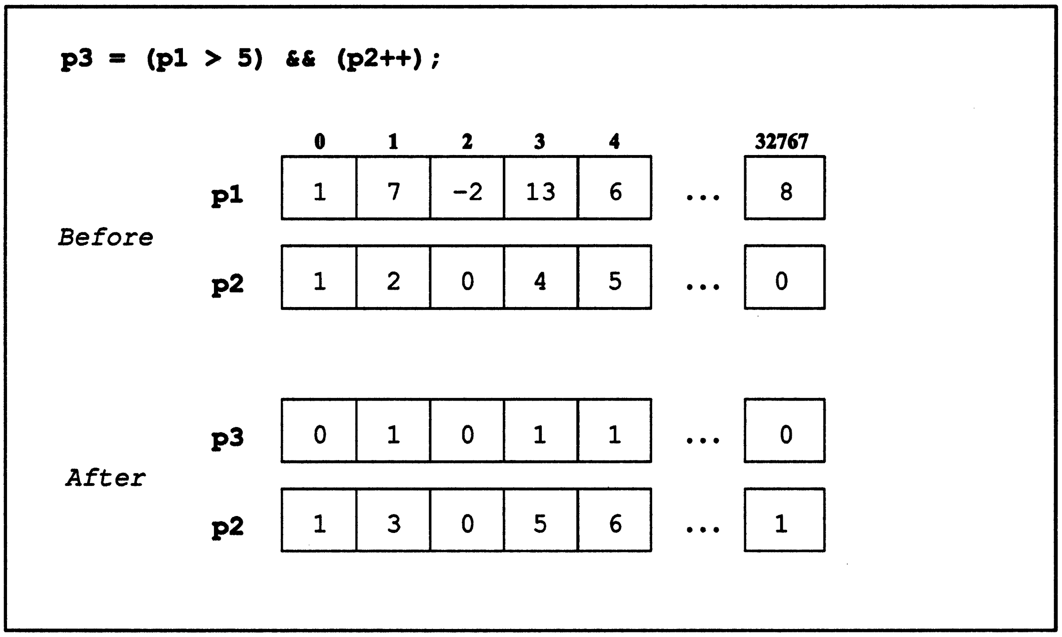

Figure 25. An example of the **&&** operator with parallel operands

Note that the left operand of the **&&** operator in this example effectively sets the context for the right operand. This is the "implicit contextualization" mentioned at the beginning of the section. That is, the operation above is equivalent to

```
where (p1 > 5)
   p2++;
```

except that the operation additionally returns the result (0 or 1) of the logical AND in each position.

After the operation, the context returns to what it was before the operator was called.

The **||** operator works similarly when one or both of its operands are parallel—except that the context for the right operand consists of those positions that evaluate to 0 for the left operand. In addition, the operator returns a 1 if either operand evaluates to TRUE, and 0 otherwise. For example,

```
p3 = (p1 > 5) || (p2++);
```

gives the results shown in Figure 26.

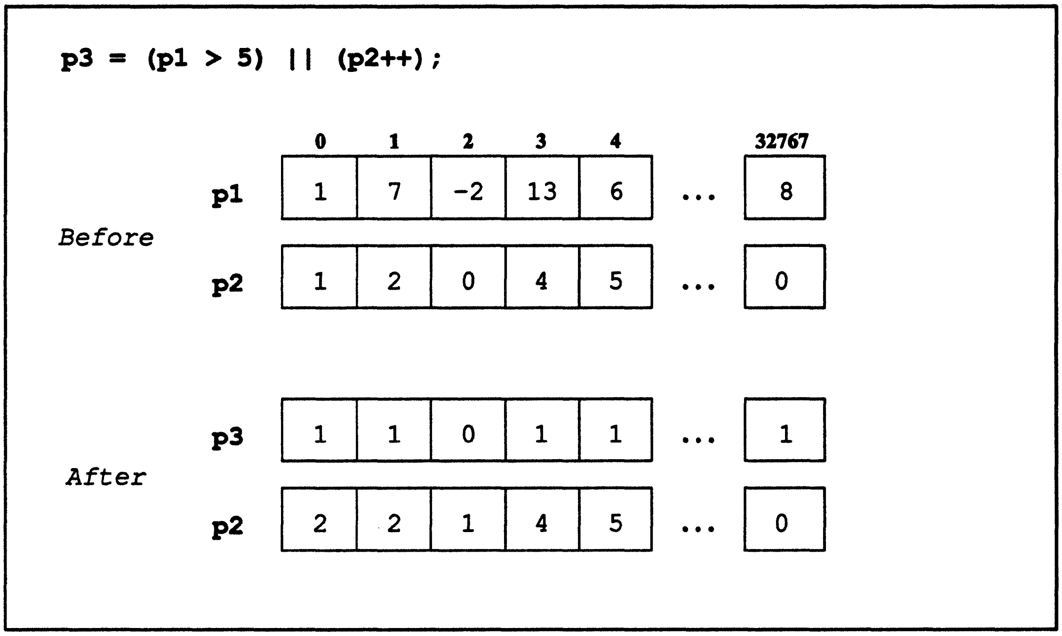

Figure 26. An example of the || operator with parallel operands

Notice the difference in the results between Figure 25 and Figure 26:

- With the || operator, **p2** is incremented only in the positions where **p1** is *not* greater than 5.
- With ||, the corresponding element of **p3** receives the logical OR of the operands for each position.

6.7.2 The ?: Operator

The **?:** operator provides implicit contextualization of its second and third operands when its first operand is parallel. For example, when **p1** is parallel,

```
(p1 > 5) ? p2++ : p3++;
```

is equivalent to:

```
where (p1 > 5)
   p2++;
else
   p3++;
```

See Section 5.1.5 for an example and for further discussion of this operator.

Appendix A discusses some efficiency considerations regarding C* operators that perform implicit contextualization.

Chapter 7

Pointers

C* has three kinds of pointers:

- The standard C pointer
- A scalar pointer to a shape
- A scalar pointer to a parallel variable

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

7.1 Scalar-to-Scalar Pointers

C* supports the standard C pointer. For example,

```
int *ptr;
```

declares **ptr** to be a pointer to an **int**; **ptr** is allocated on the front end. If **s1** is a scalar variable,

```
ptr = &s1;
```

puts the address of **s1** (on the front end) in **ptr**, and

```
s2 = *ptr;
```

puts the value of **s1** into **s2**. The CM is not involved in any of these operations.

7.2 Scalar Pointers to Shapes

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

```
shape *ptr;
```

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

```
ptr = &ShapeA;
```

makes **ptr** point to **ShapeA**. **ptr** is allocated on the front end.

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

```
with (*ptr)
```

makes **ShapeA** the current shape.

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

7.3 Scalar Pointers to Parallel Variables

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

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

declares a scalar pointer **ptr** that points to a parallel **int** of shape **ShapeA**. **ptr** is allocated on the front end.

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

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

```
ptr = &p1;
```

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

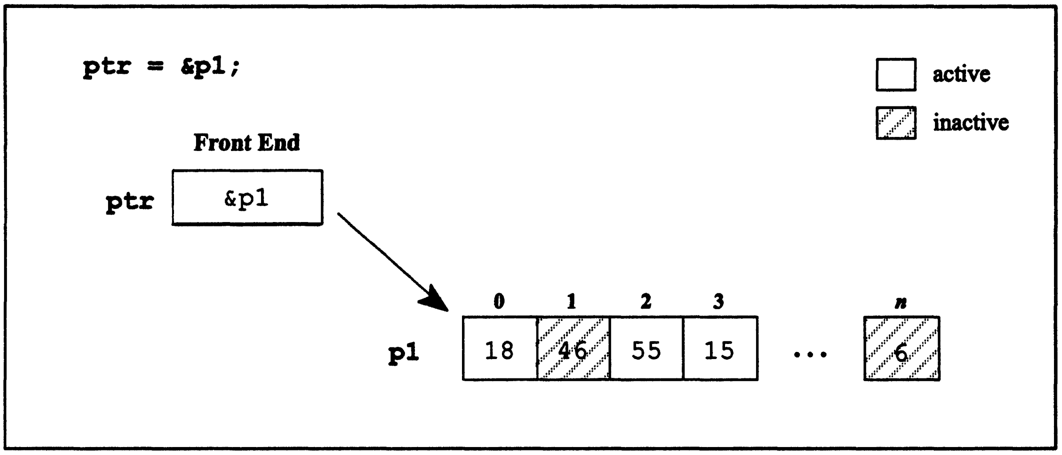

Figure 27. A scalar-to-parallel pointer

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

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

increments the value in each active element of **p1**, as shown in Figure 28.

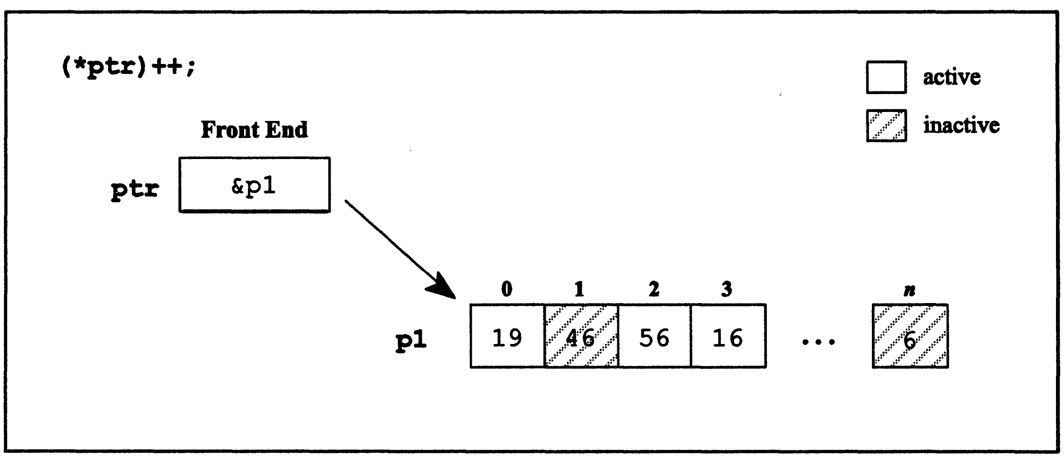

Figure 28. Dereferencing the scalar-to-parallel pointer shown in Figure 27

If **s1** is a scalar variable,

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

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

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

7.3.1 Alternative Declaration Syntax Not Allowed

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

```
int:ShapeA p1;
```

and

```
int p1:ShapeA;
```

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

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

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

7.3.2 Arrays

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

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

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

7.3.3 Pointer Arithmetic

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

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

we can do the following:

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

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

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

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

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

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

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

Here is something we *can't* do:

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

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

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

7.3.4 Parallel Indexes into Parallel Arrays

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

Consider the data shown in Figure 29. A parallel array, **A**, and a parallel variable, **i**, have been allocated in a 1-dimensional shape, **S**.

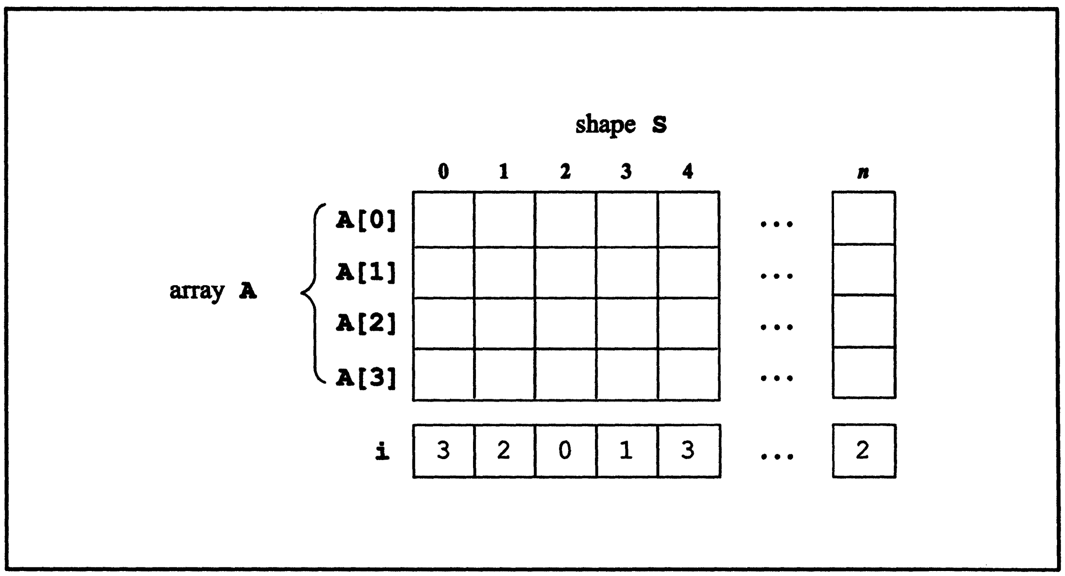

Figure 29. A parallel array and an index parallel variable

C* allows the expression **`A[i]`**. The expression says: *In each position, use the value of* **`i`** *as an index for choosing a parallel variable element.* For example, in position [0] the value of **`i`** is 3; therefore, the element of parallel variable **`A[3]`** in that position is chosen. In position [1], the value of **`i`** is 2; therefore, the element of **`A[2]`** in that position is chosen. The result is a "jagged" parallel variable consisting of parallel variable elements taken from the different parallel variables that make up the parallel array. Figure 30 shows the results.

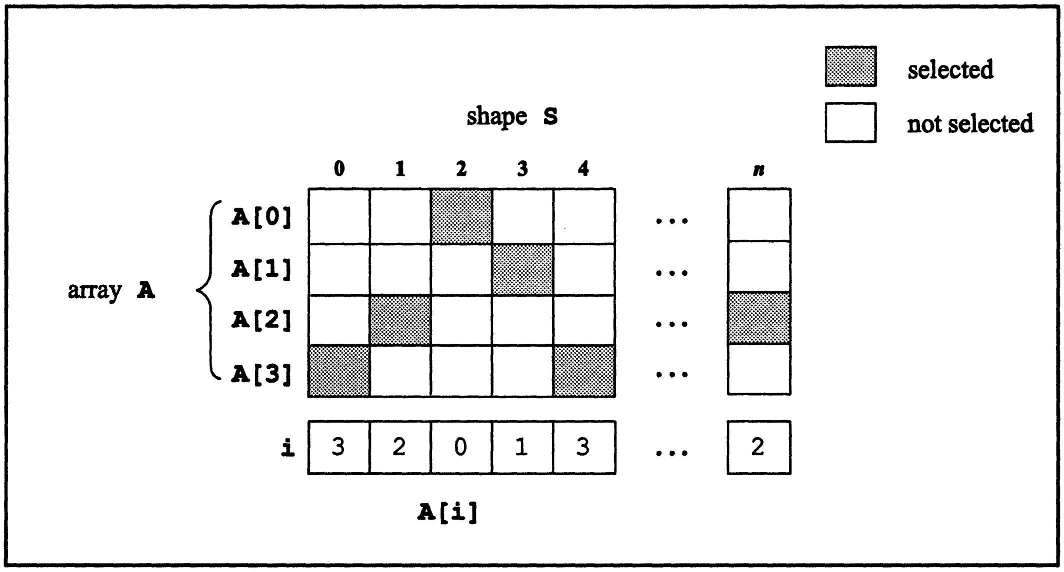

Figure 30. Indexing a parallel array by a parallel variable

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

Adding a Parallel Variable to a Pointer to a Parallel Variable

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

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

Limitations

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

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

the following statements are legal:

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

and the following statements are illegal:

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

Chapter 8

Functions

C* adds support for parallel variables and shapes to standard C functions. Specifically:

- C* functions can take parallel variables and shapes as arguments.
- C* functions can return parallel variables and shapes.
- C* adds a new keyword **`current`**, which you can use to specify that a variable is of the current shape.
- C* includes a **`void`** predeclared shape name so that you can declare an argument to be a pointer to a parallel variable of any shape.
- C* supports overloading of functions, so that (for example) functions operating on scalar and on parallel data can have the same name.

8.1 Using Parallel Variables with Functions

8.1.1 Passing a Parallel Variable as an Argument

C* functions accept parallel variables as arguments only if they are of the current shape. As in standard C, variables are passed by value; but see Section 8.2 for a discussion of passing by value versus passing by reference.

The following simple 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 parallel variable is %d.\n", +=x);
}
```

(Note that C* supports the new ANSI C function prototyping, in addition to the older method. The ANSI method is preferred.) There is actually a better way of writing this function; we describe it in Section 8.4.1.

If **p1** is a parallel variable of type **int** and shape **ShapeA**, you could call **print_sum** as follows:

```
print_sum(p1);
```

provided that **ShapeA** is the current shape. If **ShapeA** were not the current shape, passing **p1** to the function would violate the rule that a program can operate only on parallel variables of the current shape.

NOTE: If a function expects a scalar variable and you pass it a parallel variable instead, you receive a compile-time error.

If the Parallel Variable Is Not of the Current Shape

If you want to pass a parallel variable that is not of the current shape to a function, use a pointer to the parallel variable. Note, though, that if the function is to operate on the parallel variable, the function must include its own nested **with** statement, and the parallel variable that is passed must be of that shape. For example:

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

If **p1** is a parallel variable of type **int** and shape **ShapeA**, you could call **print_sum** as follows, no matter what the current shape is:

```
print_sum(&p1);
```

Section 8.4.2 discusses a more general way of passing parallel variables that are not of the current shape.

8.1.2 Returning a Parallel Variable

C* functions can return parallel values. For example, the following function:

```
float:ShapeA increment(float:ShapeA x)
{
   return (x + 1.);
}
```

takes as an argument a parallel variable of type **float** and shape **ShapeA**, and returns, for each active element of the variable, the value of the element plus 1. Assuming that **p1** and **p2** are parallel floats of shape **ShapeA**, and **ShapeA** is the current shape, you could call **increment** as follows:

```
p2 = increment(p1);
```

Note that when a function is to return a parallel variable, you must specify both the type and the shape of the variable. The header of the function **increment** could also have been written with the shape after the parameter list:

```
float increment(float:ShapeA x):ShapeA
```

You could also use a shape-valued expression. For example:

```
float increment(float:ShapeA x):shapeof(x)
```

See Chapter 3 for a discussion of the intrinsic function **shapeof**.

In a Nested Context

Consider a slightly different version of **increment**:

```
float:ShapeA  increment_if_over_5(float:ShapeA x,
      float:ShapeA y)
{
   where (y > 5.)
      return (x + 1.);
}
```

Figure 31 shows some sample results of a call to this new function.

```
with (ShapeA)
   p3 = increment_if_over_5(p1, p2);
```

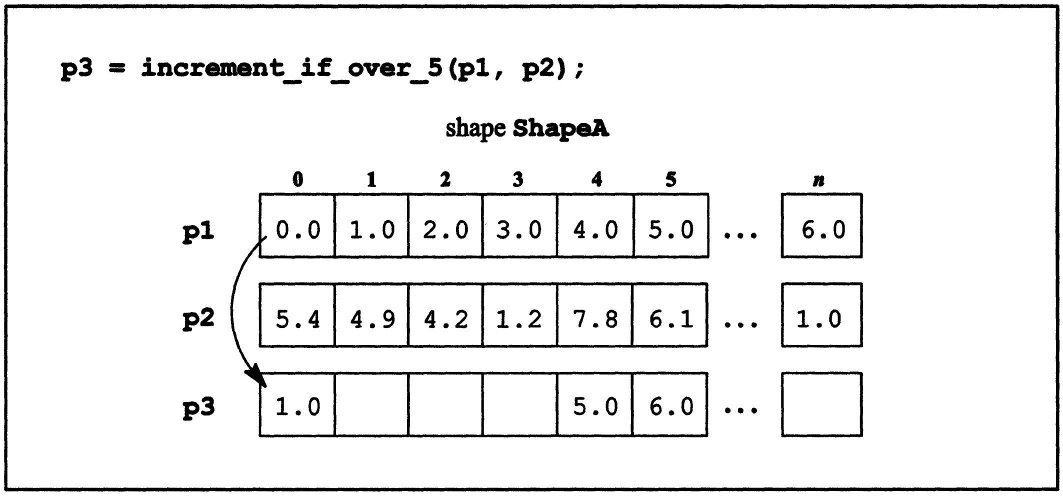

Figure 31. Three parallel variables after a function call

Here is the way things are upon return from **increment_if_over_5**:

- All positions have once again become active, as we discussed in Chapter 6.
- In every position where **p2** is greater than 5, the corresponding element of **p3** has been assigned the value of the corresponding element of **p1** plus 1.
- The values of all other elements of **p3** are undefined.

8.2 Passing by Value and Passing by Reference

You can pass parallel variables by value or by reference, just as you can scalar variables. However, in deciding whether to pass by value or pass by reference, you must take into account the effect of inactive positions.

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 in the following situations:

- If the function contains an **everywhere** statement to widen the context, and then operates on the parallel variable you pass.
- If it operates on an individual element of a parallel variable; see Section 6.2.
- If it performs send or get operations involving the parallel variable you pass; send and get operations are described in Chapter 10.

As an example of the first situation, consider the following function:

```
float:ShapeA f(float:ShapeA x)
{
   everywhere
      return (8. / x);
}
```

What happens if we pass in a parallel variable with an inactive element? Figure 32 gives an example.

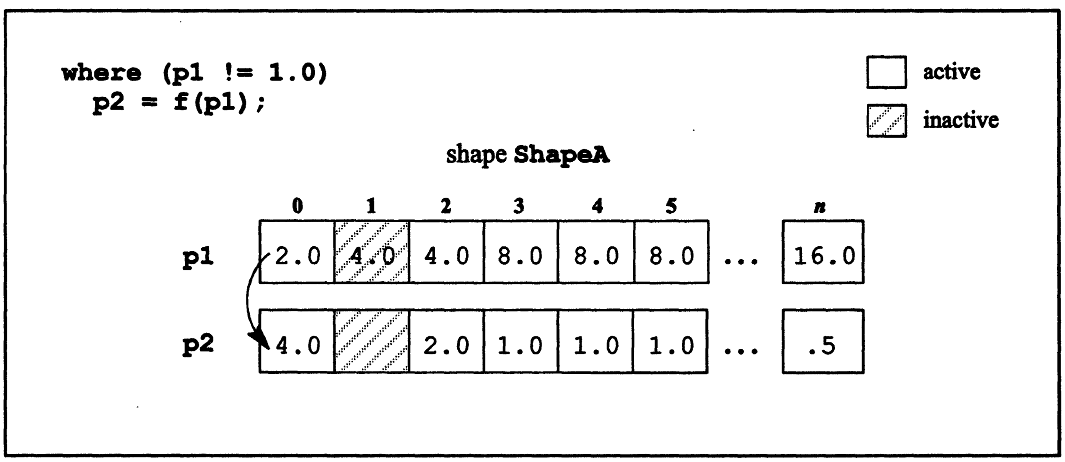

Figure 32. Passing by value when the function contains an **everywhere** statement

The copy made of **p1** contains an undefined value, rather than 4.0, in the inactive position; therefore, the value in **[1]p2** is also undefined. Note also that you would want to avoid dividing by an undefined value.

To avoid this situation, define the function so that it passes by reference rather than by value.

8.3 Using Shapes with Functions

8.3.1 Passing a Shape as an Argument

C* functions accept shapes as arguments. The following function takes a shape as an argument and allocates a local variable of that shape.

```
int number_of_active_positions(shape x)
{
   with (x) {
      int:x local = 1;
      return (+= local);
   }
}
```

The shape that you pass need not be the current shape.

If the function also returns a parallel variable that is of the shape specified in the parameter list, its shape must be declared *after* the parameter list, to avoid a forward reference. For example:

```
float raise(shape employees, float:employees salary):employees
{
   return (1.1 * salary);
}
```

This format is not especially useful in this case, since **employees** must be the current shape. The format becomes more useful when you pass more than one shape, and data is passing between the shapes. For information on communicating between shapes, see the discussion of parallel left indexing in Chapter 10 and the discussion of general communication in Chapter 14.

8.3.2 Returning a Shape

C* functions can also return a shape. For example:

```
shape choose_shape(shape ShapeA, shape ShapeB, int n)
{
   if (n)
      return ShapeA;
   else
      return ShapeB;
}
```

This function returns **ShapeA** or **ShapeB**, depending on the value of **n**.

A function that returns a shape can be used as a shape-valued expression—that is, you can use it in place of a shape name. For example:

```
with (choose_shape(shape1, shape2, s1))
   /* ... */
```

See Section 9.7, however, for limitations on the use of a function as a shape-valued expression when you are declaring a parallel variable.

8.4 When You Don't Know What the Shape Will Be

Some functions you write may be general enough that they can accept a parallel variable of *any* shape as an argument. For example, the **print_sum** function used as an example in Section 8.1 could work with any parallel variable. To allow this, C* introduces two new "predeclared" shape names: **current** and **void**. A predeclared shape name is provided as part of the language; you do not declare it in your program.

8.4.1 The current Predeclared Shape Name

The predeclared shape name **current** always equates to the current shape; **current** is a new keyword that C* adds to standard C. You can use **current** to declare a parallel variable as follows:

```
int:current variable1;
```

If **employees** is the current shape when this statement is executed, **variable1** is of shape **employees**; if **image** is the current shape, **variable1** is of shape **image**.

NOTE: Since **current** is dynamic, you cannot use it with a parallel variable of static storage duration.

Thus, we can generalize **print_sum** as follows to let it take any parallel **int** of whatever shape is current when the function is called:

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

In fact, this version of the function is more efficient than the version that specifies a particular shape name in the parameter list. If the function specifies a shape name (and you have turned safety on), the compiler has to first make sure that the shape is current, and that the parallel variable is of the current shape. If the function uses **current**, the compiler has to make sure only that the parallel variable is in fact of the current shape.

8.4.2 The void Predeclared Shape Name

C* extends the use of the ANSI C keyword **void**. In addition to the standard use, it can be used as the shape modifier for a scalar-to-parallel pointer; it specifies a shape without indicating what the shape's name is. C* does no type checking of a **void** shape.

Use **void** instead of a shape name in a function's parameter list to specify that *any* shape is acceptable as an argument to the function. If you are specifying a parallel variable that can be of any shape, a type specifier (for example, **int**, **float**) is still required. Since you cannot pass a parallel variable that is not of the current shape, **void** must be the shape modifier of a scalar-to-parallel pointer. For example, the following function sums the values of the active elements of a parallel **int** of any shape:

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

You can also use **void** outside a parameter list to declare a scalar pointer to a parallel variable. For example:

```
int:void *ptr;
```

This declares **ptr** to be a pointer to a parallel **int** of an undetermined shape. The shape is determined by the parallel variable whose address is ultimately assigned to the pointer. For example, if **ptr** points to **p1**:

```
ptr = &p1;
```

then **ptr** is a pointer to an **int** of shape **shapeof(p1)**. But note that a parallel variable of another shape could subsequently be assigned to **ptr**, and the C* compiler would not complain; **ptr** would then simply point to the new parallel variable.

Using shapeof with the void Shape

While convenient, using the **void** shape slows down a program if run-time safety is enabled. It is therefore preferable to use **void** only for the first parameter of a function. For subsequent parameters of the same shape, use the **shapeof** intrinsic function; **shapeof** provides more information to the compiler, thereby allowing the compiler to generate better code. Also use **shapeof** in the controlling expression of the **with** statement to choose the current shape.

For example:

```
int sum_of_two_vars(int:void *x, int:shapeof(*x) *y)
{
   with (shapeof(*x))
      return (+= (*x + *y));
}
```

For parameters declared locally within the function, use **current**:

```
float average(int:void *x)
{
   with (shapeof(*x)) {
      int:current y = 1;
      return (+=*x / +=y);
   }
}
```

Using void when Returning a Pointer

Consider the following function, which is passed a shape and returns a pointer to a parallel variable of that shape:

```
int *f(shape ShapeA):ShapeA        /* This is wrong */
{
   /* ... */
}
```

The shape of the return value must come after the parameter list, to avoid a forward reference. However, C* doesn't allow this alternative syntax for a function returning a pointer. The problem is the same as that discussed in Section 7.3.1; the compiler interprets the return value incorrectly as "a parallel pointer of shape **ShapeA** to a scalar **int**," and parallel-to-scalar pointers do not exist in C*.

Use **void** instead of the shape name for the return value in this situation. For example:

```
int:void *f(shape ShapeA)
{
   /* ... */
}
```

Note that this causes an unavoidable loss of some type-checking, since the compiler cannot check for the correct use of the shape of the variable pointed to.

8.5 Overloading Functions

It may be convenient for you 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, the following 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, the following statement specifies that there may be more than one version of the **increment** function:

```
overload increment;
```

Put the **overload** statement at the beginning of the file that contains the declarations of the functions. The statement *must* appear before the declaration of the second version of the function, and it must appear in the same relative order with respect to the function declarations in all compilation units. Thus, if it appears first in one compilation unit, it must appear first in all compilation units. If you use a header file for your function declarations, this happens by default.

If you have different versions of more than one function, separate the function names by commas in the **overload** statement. For example:

```
overload increment, average;
```

NOTE: The current implementation of C* restricts the shapes you can specify in parameters to an overloaded function. Only **current** and **void** can be used in overloaded functions.

Chapter 9

More on Shapes and Parallel Variables

Chapter 3 introduced C* shapes and parallel variables. This chapter discusses more aspects of these important topics. Specifically:

- Partially specifying a shape; see Section 9.1.
- Creating copies of shapes; see Section 9.2.
- Dynamically allocating and deallocating a shape; see Sections 9.3 and 9.4.
- Using the C* library function **palloc** to explicitly allocate storage for a parallel variable; see Section 9.5.
- Casting to a shape, and casting to or from a parallel data type; see Section 9.6.

9.1 Partially Specifying a Shape

It is possible to declare a shape without fully specifying its rank and dimensions. You might do this, for example, if the number of positions in the shape is to be determined from user input. For example,

```
shape ShapeA;
```

declares a shape **ShapeA** but does not specify its rank or dimensions. Such a shape is *fully unspecified*.

```
shape []ShapeB;
```

specifies that **ShapeB** has a rank of 1, but does not specify the number of positions. Such a shape is *partially specified.*

You must fully specify a shape before using it (for example, before allocating parallel variables of that shape). Sections 9.2 and 9.3 describe ways of fully specifying a partially specified or fully unspecified shape.

The **rankof** intrinsic function returns 0 for a fully unspecified shape. For a partially specified shape, it returns the negative of the rank. For example, given the following shapes:

```
shape s, [][]t, [8092]u;
```

The following statements are true:

```
rankof(s) == 0;
rankof(t) == -2;
rankof(u) == 1;
```

This information can be used if you don't know whether or not a shape is fully specified—for example, in a function, where the function can fully specify a shape only if necessary.

9.1.1 Partially Specifying an Array of Shapes

You can also create an array of shapes that is partially specified. For example,

```
shape ShapeC[10];
```

declares that **ShapeC** is an array of 10 shapes, but does not specify the rank or dimensions of any of them.

```
shape [][]ShapeD[10];
```

declares that **ShapeD** is an array of 10 shapes, each of rank 2, but does not specify the number of positions in any of them.

A shape within such an array is specified with a right index in the standard manner. For example,

```
with (ShapeD[0])
```

makes the first shape in the array the current shape. Note that the shape must become fully specified before you can use it in this way.

You cannot use a parallel variable as an index into an array of shapes.

Arrays and Pointers

The standard C equivalence of arrays and pointers is maintained in C* with arrays of shapes and pointers to shapes. For example, if we declare a scalar pointer to **`Sarray`**:

```
shape *ptr;
ptr = Sarray;
```

then **`*ptr`** is equivalent to **`Sarray[0]`** and to **`*Sarray`**. Similarly,

```
Sarray[3]
```

is equivalent to

```
*(ptr + 3)
```

and to

```
*(Sarray + 3)
```

9.1.2 Limitations

You cannot partially specify the dimensions of a shape. The following is incorrect:

```
shape [][4]ShapeE;      /* This is wrong */
```

Also, you cannot partially specify the rank of a shape. The following is incorrect, if you later want to specify the shape as having a rank of 2:

```
shape []ShapeF;
```

A program cannot call the **positionsof** or **dimof** intrinsic functions if the information they require has not yet been specified. If it is known when the program is being compiled that an error will result from such a call, the compiler reports an error. Otherwise, a run-time error is reported.

A shape must be fully specified before you can declare a parallel variable to be of that shape. You generally receive a compiler error if you try to declare a parallel variable to be of a shape that is not fully specified. A couple of exceptions:

- If the parallel variable is declared as an automatic in a nested scope. For example:

  ```
  shape ShapeA;

  main()
  {
     int:ShapeA p1;
  }
  ```

  In this case, the compiler assumes that **ShapeA** is fully specified elsewhere in the program. If it is not, a run-time error may be generated, depending on the safety level you choose.

- If the shape has a storage class of **extern**. For example:

  ```
  extern shape ShapeB;
  int:ShapeB p2;
  ```

  In this case, the compiler assumes that **ShapeB** is fully specified in some other compilation unit, and a run-time error may be generated if it is not.

The next section describes how to, in effect, create copies of shapes. The section after that describes how to fully specify a partially specified or fully unspecified shape using the C* intrinsic function **allocate_shape**.

9.2 Creating Copies of Shapes

One way to fully specify a shape is by using the assignment operator to copy a fully specified shape to a partially specified one. For example:

```
shape ShapeA;
shape [256][256]ShapeB;
ShapeA = ShapeB;
```

In this case, both **ShapeA** and **ShapeB** refer to the same shape. You can use either one in a **with** statement to make this shape the current shape. This is different from what would happen if both were declared separately, but with the same dimensions. For example:

```
shape [256][256]ShapeA;
shape [256][256]ShapeB;
```

In this case, **ShapeA** and **ShapeB** refer to two separate physical shapes that happen to have the same rank and dimensions.

You can also fully specify a shape by using a shape-valued expression as the RHS of the assignment. For example:

```
ShapeA = shapeof(p1);        /* p1 is a parallel variable of some
                                other shape  */
ShapeB = (new_shape());      /* new_shape returns a shape */
ShapeC = *ptr;               /* ptr is a pointer to a shape */
```

9.2.1 Assigning a Local Shape to a Global Shape

Be careful when assigning a fully specified shape in local scope to a partially specified shape in file scope. The following code illustrates the problem:

```
shape ShapeA;                   /* Unspecified shape ShapeA */

void f(void)
{
   shape [1024][512]ShapeB;     /* Fully specified shape ShapeB
                                   in local scope */
   ShapeA = ShapeB;             /* ShapeB assigned to ShapeA */
}

main()
{
   f();
   {
```

```
        int:ShapeA p1;           /* This allocation fails because
                                    ShapeA's shape was deallocated
                                    when function f exited. */
    }
}
```

In this case, the actual physical shape that **ShapeA** refers to is allocated in local scope. When function **f** exits in the sample code, this shape is deallocated. When the code subsequently tries to declare a parallel variable of shape **ShapeA**, it gets an error, because the shape no longer exists.

The situation is analogous to what happens when a local pointer is assigned to a global pointer in standard C.

9.3 Dynamically Allocating a Shape

Another way to fully specify a partially specified or fully unspecified shape is to use the C* intrinsic function **allocate_shape**. **allocate_shape**'s first argument is a pointer to a shape; its second argument is the rank of this shape; subsequent arguments are the number of positions in each rank. The function returns the shape it points to. For example,

```
shape []ShapeB;
ShapeB = allocate_shape(&ShapeB, 1, 65536);
```

complete the specification of the partially specified one-dimensional shape **ShapeB**.

You needn't partially specify a shape before calling **allocate_shape**. For example,

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

returns a three-dimensional shape called **new_shape**.

allocate_shape can also fully specify elements of an array of shapes. For example:

```
ShapeD[0] = allocate_shape(&ShapeD[0], 2, 4, 16384);
```

Alternatively, you can use an array to specify the number of positions in each rank. 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. The following example reads the rank and dimensions in from a file named **shape_info** and uses these values as arguments to **allocate_shape**.

```
#define  MAX_AXES  31
#include <stdio.h>

main()
{
   FILE *f;
   int axes[MAX_AXES], i, rank;
   shape ShapeA;

   f = fopen("shape_info", "r");

   fscanf(f, "%d", &rank);
   if (rank > MAX_AXES) {
      fprintf (stderr, "Rank bigger than maximum allowed.\n");
      exit(1);
   }
   for (i = 0; i < rank; i++)
      fscanf(f, "%d", &axes[i]);

   ShapeA = allocate_shape(&ShapeA, rank, axes);
}
```

Note that **axes** is initialized as an array of 31 elements, since the CM restricts shapes to a maximum of 31 dimensions. Of course, the file **shape_info** could contain fewer than the maximum number of dimensions.

NOTE: For certain programs you may be able to improve performance by using the intrinsic function **allocate_detailed_shape** instead of **allocate_shape**; see Appendix A.

9.4 Deallocating a Shape

Use the C* library function **deallocate_shape** to deallocate a shape that was allocated using the **allocate_shape** function. Its argument is a pointer to a shape. Include the header file **<stdlib.h>** if you call **deallocate_shape**. Note that this is not required for **allocate_shape**, which is an intrinsic function.

There are two reasons you might deallocate a shape:

- If you have reached the limit on the number of shapes imposed by your CM system
- If you want to reuse a partially specified shape

As an example of the latter, consider the following code:

```
#include <stdlib.h>

shape []S;
int positions = 4096;

main()
{
   while (positions<=65536) {
      S = allocate_shape(&S, 1, positions);
         {
            int:S p1, p2, p3;
            /* Parallel code omitted ... */
         }
      deallocate_shape(&S);
      positions *= 2;
   }
}
```

In this code, shape **S** is allocated every time it goes through the **while** loop, and deallocated at the end of the loop. This lets it have a different number of positions each time through the loop.

The results of deallocating a shape that was fully specified at compile time are undefined; the compiler generates an error when it notices a program doing this, but it doesn't guarantee that it will catch all cases.

You should not deallocate a shape that contains parallel variables; if you do, the behavior of these parallel variables is undefined. Note that in the code fragment above, the parallel variables declared to be of shape **S** go away when you leave the block.

As discussed in Section 9.2, you can create copies of shapes by assigning one shape to another. If you have created copies of shapes in this way and you deallocate one, the effect on the others is undefined.

9.5 Dynamically Allocating a Parallel Variable

The C* library routine **palloc** is the parallel equivalent of C library routines like **malloc** and **calloc**. Use it to explicitly allocate storage for a parallel variable. It can be called whether or not the parallel variable's shape is dynamically allocated. Include the file **<stdlib.h>** if you call **palloc** or its companion function **pfree**.

palloc takes two arguments: a shape, and a size (in **bools**). It allocates space of that size and shape, and returns a scalar pointer to the beginning of the allocated space. The shape passed as an argument must be fully specified before **palloc** is called.

palloc returns 0 if it cannot allocate the memory.

To allocate space for a parallel variable of shape **ShapeA**, for example, you could do the following:

```
#include <stdlib.h>

shape [16384]ShapeA;
int:ShapeA *ptr;

main()
{
   ptr = palloc(ShapeA, boolsizeof(int:ShapeA));
}
```

The scalar variable **ptr** now contains a pointer to an **int**-sized parallel variable of shape **ShapeA**. You can reference this parallel variable by using ***ptr**. The contents of the parallel variable are undefined.

Use **pfree** to deallocate storage you allocated with **palloc**. **pfree** takes as its argument the pointer returned by **palloc**. For example, to deallocate the storage allocated by the call to **palloc** above, call **pfree** as follows:

```
pfree(ptr);
```

The **palloc** and **pfree** calls can also be used with a dynamically allocated shape, as in the following example:

```
#include <stdlib.h>

shape S;
double:S *p;
```

```
main()
{
   S = allocate_shape(&S, 2, 4, 8192);
   p = palloc(S, boolsizeof(double:S));
   /* ... */
   pfree(p);
   deallocate_shape(&S);
}
```

Note that you can declare a scalar pointer to a shape that is not fully specified, even though you cannot declare a parallel variable of that shape.

9.6 Casting with Shapes and Parallel Variables

Use the C* cast operator to cast an expression to a particular shape and type. For example,

```
(char:employees)
```

specifies that the expression following it is to be formed into a **char** of shape **employees**. A data type is required as well as a shape in a parallel cast.

9.6.1 Scalar-to-Parallel Casts

Using a parallel cast is a quick way to promote a scalar value. The following code stores in scalar variable **s1** the number of active positions of the current shape:

```
s1 = +=(int:current)1;
```

In the statement, 1 is cast to a parallel **int** of the current shape. The **+=** reduction operator sums the resulting parallel variable for all active positions, and the result is assigned to the scalar variable **s1**.

9.6.2 Parallel-to-Parallel Casts

Parallel-to-parallel casts are also permitted.

Casts to a Different Type

You can cast a parallel variable so that it has a different type. For example:

```
int:ShapeA p1;
sqrt((double:ShapeA)p1);
```

The parallel version of **sqrt** requires a **float** or a **double**; therefore, we must cast the parallel **int p1** before we can pass it to this function.

Casts to a Different Shape

Casting of a parallel variable to a different shape is limited to the situation in which the same shape can be referenced by more than one name. In this case, a cast may sometimes be necessary to ensure that the compiler recognizes that two parallel variables are supposed to be of the same shape. For example:

```
shape [256][256]ShapeB, ShapeA;

main()
{
  ShapeA = ShapeB;
  {
   int a:ShapeA, b:ShapeB;
   with(ShapeB) {
      b = a;               /* This gets a compile-time error */
      b = (int:ShapeB)a;   /* This works */
      }
   }
}
```

The cast is required so that the compiler is made aware that **ShapeA** and **ShapeB** refer to the same shape.

No movement of data is implied in a parallel-to-parallel cast.

The effects of casting an expression between two shapes that are different (for example, with a different rank or number of positions) are undefined.

9.6.3 With a Shape-Valued Expression

You can use a shape-valued expression with a scalar-to-parallel or parallel-to-parallel cast. The expression must be enclosed in parentheses unless it is an intrinsic function. For example,

```
s1 = +=(int:(shape_array[3]))1;
```

casts 1 to be an **int** of the fourth shape in the array **shape_array**.

9.6.4 Parallel-to-Scalar Casts

You can cast a parallel variable to a scalar type. The result is similar to a demotion of a parallel variable when assigning it to a scalar (see Chapter 5); the operation picks one of the active values of the parallel variable and returns that as the result. If no positions are active, the result of the cast is undefined. If you choose the appropriate safety level, you receive a run-time error if no positions are active.

9.7 Declaring a Parallel Variable with a Shape-Valued Expression

A shape-valued expression, as we have described earlier, is an expression that can be used in place of a shape name. You can therefore use a shape-valued expression in declaring a parallel variable. The expression must be enclosed in parentheses unless it is the **shapeof** intrinsic function. For example:

```
shape [256][256]matrix;
int:matrix p1;
int:shapeof(p1) p2;     /* p2 is of shape matrix */
int:(get_a_shape()) p3; /* get_a_shape returns a shape */
```

However, if the declaration appears at file scope, or is **static** or **extern**, the shape-valued expression must be a constant. This means that the expression must be one of the following:

- A simple shape that is fully specified at compile time, or that has a storage class of **extern**. For example, **shapeof** in the example above refers to a fully specified shape.
- An array of shapes that is fully specified at compile time and whose right index is a constant expression. For example:

```
shape [256][512]Sarray[40];
int:(Sarray[17]) p1;
int:(Sarray[4-3]) p2;
```

- An indirection of an array of shapes that is fully specified at compile time, with a constant expression added to it. For example:

```
shape [512][256]Sarray[40];
int:(*(Sarray + 17)) p1;
int:(*(Sarray + 4 - 3)) p2;
```

The following are illegal:

```
shape Sarray1[40];
int:(Sarray1[17]) p1;     /* This is wrong */
```

Sarray1 is not fully specified; therefore, you can't declare **p1** to be a parallel variable of any of the elements of it.

```
shape [512][256]Sarray[40];
int:(Sarray[f(x)]) p1;    /* This is wrong */
```

In this case, **Sarray** is fully specified, but **f(x)** is not a constant expression, since it invokes a function whose result is not known until run time.

```
shape *ptr;
int:(*ptr) p1;     /* This is wrong */
```

In this case, **ptr** does not point to a fully specified shape.

9.8 The physical Shape

C* contains the predeclared shape name **physical**; **physical** is a new keyword that C* adds to standard 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 Connection Machine system. Note, therefore, that the number of positions in the shape is not known until run time. You can use **physical** as you would any other shape.

For example,

```
positionsof(physical);
```

returns the number of positions in shape **physical**, which is equal to the number of physical processors on which the program is running.

```
(int:physical)p1
```

casts **p1** to be an **int** of shape **physical**.

Chapter 10

Communication

This chapter describes methods you can use to perform communication among parallel data. For example:

- Sending values of parallel variable elements to other elements of the same or a different shape.
- Getting values of parallel variable elements that are of the same or a different shape.

C* provides two methods of communication:

- *General communication,* in which the value of any element of a parallel variable can be sent to any other element, whether or not the parallel variables are of the same shape. You can use parallel left indexing to perform general communication. Parallel left indexing is described in Section 10.1.
- *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. You can use the **pcoord** function, combined with parallel left indexing, to perform grid communication. The **pcoord** function is described in Sections 10.2 and 10.3.

In addition to the methods described in this chapter, C* includes a library of functions that provide an alternative way of performing grid and general communication; these functions are discussed in Part III of this manual. There are some differences in what you can accomplish using the different methods. but for most purposes the choice between the methods depends on individual preference.

10.1 Using a Parallel Left Index for a Parallel Variable

By now you should be familiar with the left indexing of a parallel variable to specify an individual element. For example, `[0]p1` specifies the first element of the 1-dimensional parallel variable `p1`. Similarly, if `s1` and `s2` are scalar variables, their values determine which element is specified by the 2-dimensional parallel variable `[s1][s2]d1`. But we have not yet covered the case in which a parallel variable is used as a left index for another parallel variable. If `p0` and `p1` are both 1-dimensional parallel variables, what does `[p0]p1` mean? If `d0`, `d1`, and `d2` are all 2-dimensional parallel variables, what does `[d0][d1]d2` mean?

Basically, 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. The example discussed below will help show how this works. (Note that this and other examples in this chapter do not represent valid shapes, because there are too few positions; we use these small shapes to make it easier to visualize what happens when you use a parallel left index.)

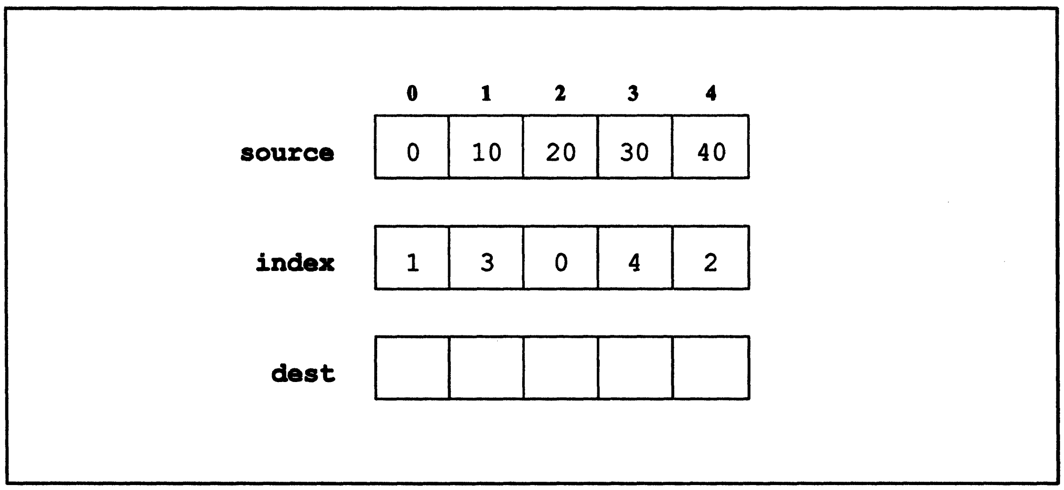

Figure 33. Three parallel variables

10.1.1 A Get Operation

Given the situation shown in Figure 33, what is the result of the following statement?

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

Let's look first at what goes into element 0 of **dest**. The value in element [0] of **index** is 1. This value is used as an index into the elements of **source**. The value in element 1 of **source** is 10. Therefore, element 0 of **dest** gets assigned the value 10. The way to think of this is that the LHS variable gets a value of the RHS variable, based on the value of the corresponding element of the index variable; we refer to this as a *get operation*. In C* code, what happens is this:

```
[0]dest = [1]source;
```

For element 1 of **dest**, the value of the **index** variable is 3. Therefore, element 1 of **dest** gets the value of element 3 of **source**, which is 30. In C* code:

```
[1]dest = [3]source;
```

And for the remaining elements:

```
[2]dest = [0]source;
[3]dest = [4]source;
[4]dest = [2]source;
```

It's important to note the difference between parallel left indexing and these serial statements. Parallel left indexing causes these assignments to occur *at the same time,* in parallel. In the serial statements, the result of an earlier statement could affect the result of a later one; this does not happen when all the statements are executed at the same time.

Figure 34 shows the results of the assignment statement for all elements of **dest**; the arrows show the process by which a value is assigned to **[0]dest**. The value of **[0]index** is 1, which causes **[0]dest** to get the value in **[1]source**.

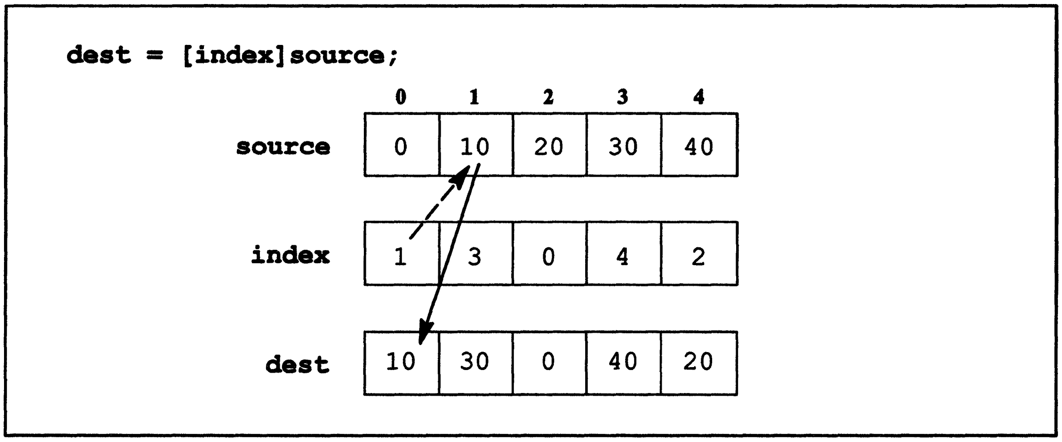

Figure 34. Parallel left indexing of a parallel variable—a get operation

10.1.2 A Send Operation

Here is another assignment statement that uses the data in Figure 33:

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

In this case, **index** is being used as an index for **dest**. In statements of this form, the RHS variable *sends* a value to the LHS variable, based on the value of the corresponding element of the index variable; we refer to this as a *send operation.*

Let's look at element 0 of **source**. The value in element 0 of the index variable **index** is 1; this value is used as an index into **dest**. The value in element 0 of **source**, 0, is sent to element 1 of **dest**. In C* code:

```
[1]dest = [0]source;
```

For element 1 of **source**, in the corresponding element, the value of **index** is 3; therefore, the value in element 1 of **source**, 10, is sent to element 3 of **dest**. In C* code:

```
[3]dest = [1]source;
```

The serial C* statements for the rest of the elements are:

```
[0]dest = [2]source;
[4]dest = [3]source;
[2]dest = [4]source;
```

Note once again, however, that parallel left indexing causes all these statements to be executed at the same time. The results are shown in Figure 35; the arrows show the process by which the value in **[0]source** is assigned to an element of **dest**. The value in **[0]index** is 1; therefore, **[0]source** sends its value to **[1]dest**.

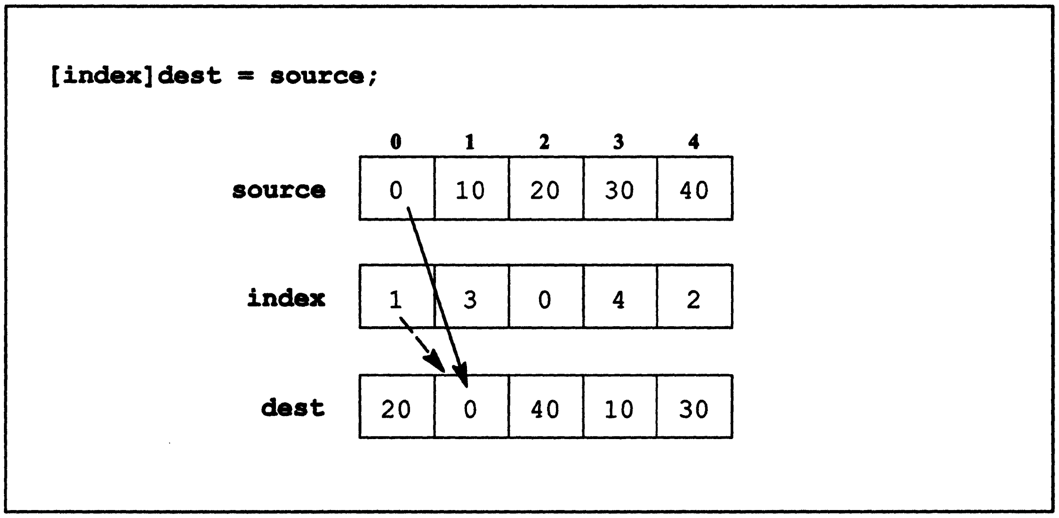

Figure 35. Parallel left indexing of a parallel variable—a send operation

10.1.3 Use of the Index Variable

The index variable would typically contain values that cause a meaningful rearrangement of the parallel variable it indexes. For example, if we use the values shown in Figure 36,

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

causes **dest** to contain the **source** values in reverse order; the arrows show the process by which **[0]dest** gets its value, based on the index in **index**.

The index variable cannot reference nonexistent elements of a parallel variable. For example, an index value of 5 in Figure 36 creates an error. If you choose the appropriate level of safety, you get a run-time error when you program tries to do this. Otherwise, the results are unpredictable.

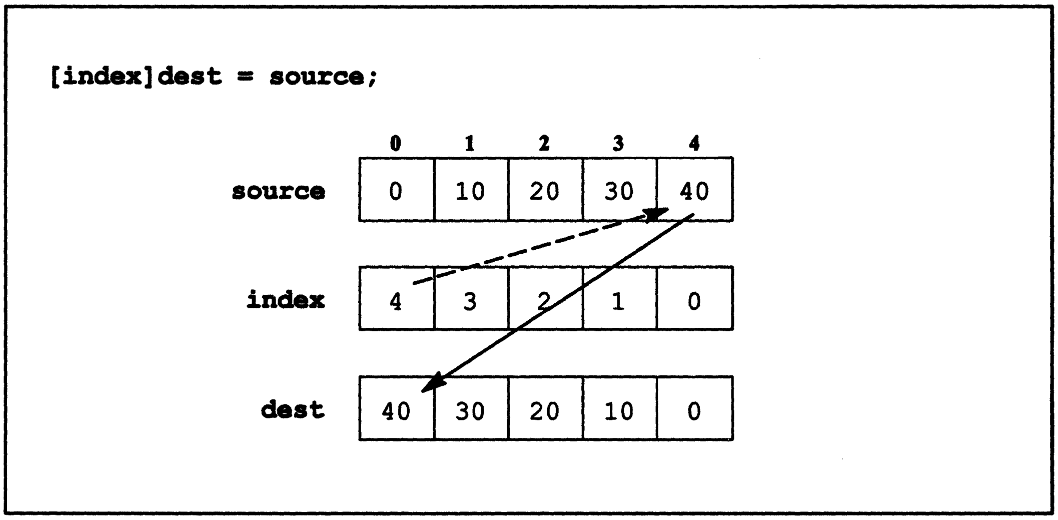

Figure 36. An index that reverses the order of a parallel variable

10.1.4 If the Shape Has More Than One Dimension

Parallel left indexing can be used if the parallel variable is of a shape with more than one dimension. In this case, however, you need to specify a left index for each axis of the shape. For example:

```
shape [128][512]ShapeA;
int:ShapeA dest, index0, index1, source;

main()
{
   with (ShapeA)
      dest = [index0][index1]source;
}
```

In this case, **source** is of the 2-dimensional shape **ShapeA**. Therefore, it requires two left indexes to specify the values to be assigned to **dest**. **index0** is used as the index for axis 0 of **source**, and **index1** is used as the index for axis 1 of **source**.

If one of the indexes is parallel and one or more are scalar, the scalar indexes are promoted to parallel in the current shape.

10.1.5 When There Are Potential Collisions

In the examples of parallel left indexing shown so far, the index variable, **`index`**, has had different variables in each element. Let's consider a situation, shown in Figure 37, where this is not true.

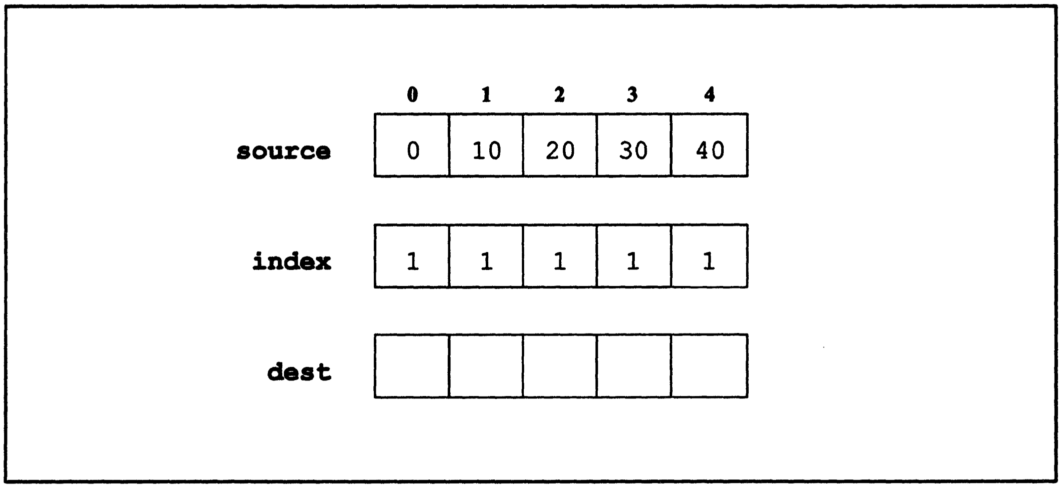

Figure 37. An index with the same value in each element

For a Get Operation

Using the data in Figure 37, the result of the following get operation is straightforward:

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

For each element of **`dest`**, the **`index`** index into **`source`** is 1. This means that the value in element 1 of **`source`**, 10, is assigned to each element of **`dest`**, as shown in Figure 38.

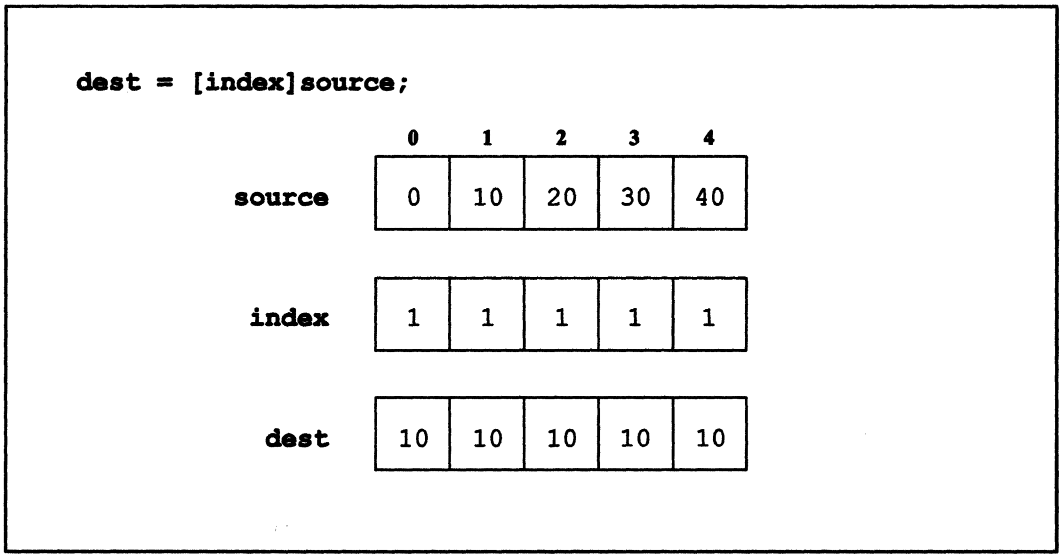

Figure 38. A get operation where the index has the same value in each element

It is equivalent to the following C* code:

```
[0]dest = [1]source;
[1]dest = [1]source;
[2]dest = [1]source;    /* ... and so on */
```

except that all operations are carried out at the same time, in parallel. There are no potential collisions in get operations.

For a Send Operation

If we try the following, however:

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

we have a problem. For each element of **`source`**, the index into **`dest`** is 1. This means that all the values of all the elements of **`source`** attempt to write into element 1 of **`dest`**. In serial C* code:

```
[1]dest = [0]source;
[1]dest = [1]source;
[1]dest = [2]source;    /* ... and so on */
```

This is an example of potential *collisions*, which could occur when more than one element tries to write into the same element at the same time. To avoid the collisions, C* chooses one of the **`source`** elements to assign to **`[1]dest`**. How it chooses the element is defined by the implementation.

You can use any C* reduction assignment operator in this situation. For example, we could specify the following:

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

This statement says: *If there is going to be a collision of* **`source`** *values assigned to any of the elements of* **`dest`**, *add the values of the* **`source`** *elements that would otherwise collide, then add this result to the value of the* **`dest`** *element.*

In cases where there are no collisions, the value of the **`source`** element is simply added to the value of the **`dest`** element. In the example, all the values of **`source`** are summed, and the result is assigned to element 1 of **`dest`**, as shown in Figure 39. (Note that if you *knew* all the index values were the same, it would be more efficient to use a simple unary reduction operator instead of doing parallel left indexing.)

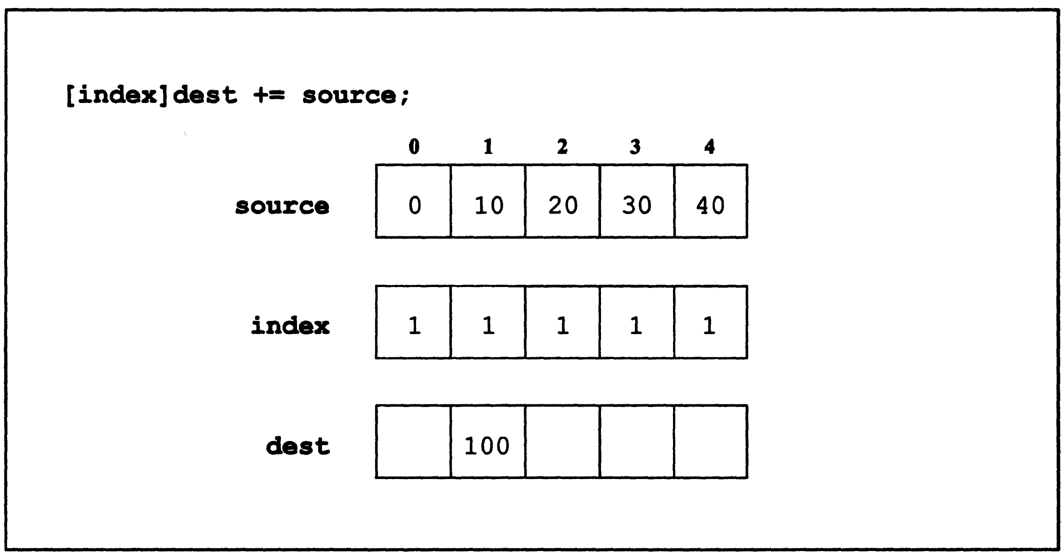

Figure 39. A reduction assignment when the parallel left index is on the LHS

The kind of reduction assignment operator you use specifies the way the colliding elements are combined. For example, the >?= operator selects the maximum value of the elements.

Note that the reduction occurs only for elements that would otherwise collide. Given the examples shown in the previous section, for example, the type of reduction assignment you use would not matter, because there are no possible collisions. This is consistent with the way parallel-to-scalar reduction operators work, because all values of the parallel variable will collide when they are assigned to a scalar variable; therefore, all must be included in the specified reduction operation.

To sum up:

- In a get operation, you don't have to consider using a reduction assignment operator, because there are no potential collisions.
- In a send operation, there may be potential collisions. If you simply use = instead of a reduction assignment operator, and there is a potential collision, C* picks one of the colliding values and assigns it to the element.

10.1.6 When There Are Inactive Positions

The examples of parallel left indexing shown so far have assumed that all positions are active. What happens when a **where** statement makes some positions inactive?

For a Get Operation

Consider the following get operation:

```
where (source < 30)
   dest = [index]source;
```

In this situation, the **where** statement deselects positions [3] and [4], using the data shown in Figure 40, but it deselects them only for *getting* purposes. Parallel variable elements in these positions cannot get values; however, elements in active positions can obtain values from them. The serial C* code would therefore be:

```
[0]dest = [1]source;
[1]dest = [3]source;
[2]dest = [0]source;
```

except that all operations occur at the same time. Figure 40 shows the results; the arrows show how **[1]dest** gets its value.

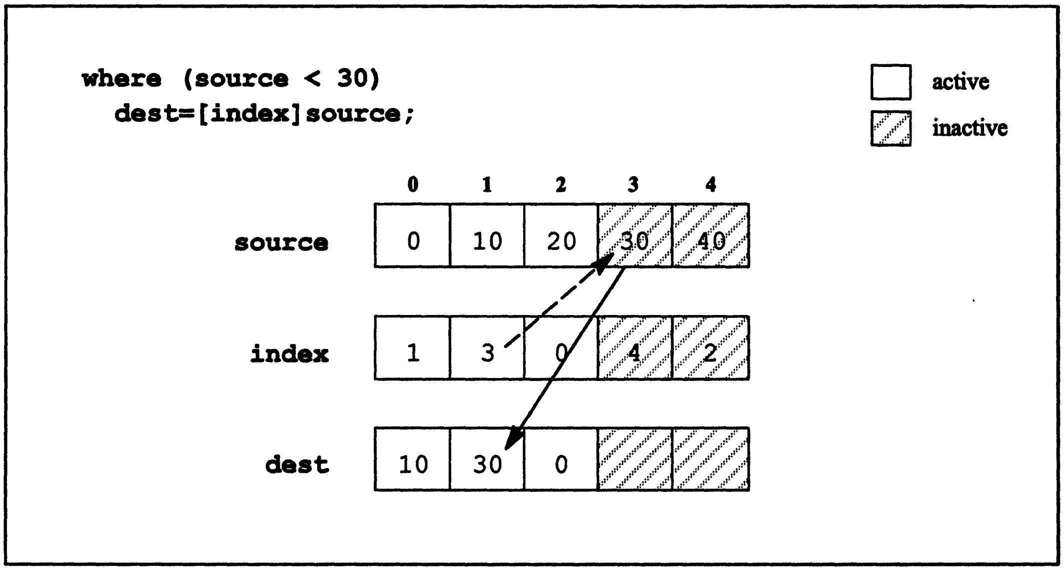

Figure 40. A get operation with inactive positions

Note these results:

- **[1]dest** gets a value from **[3]source**, even though position [3] is inactive.
- **[4]dest** does not get a value from **[2]source**, because position [4] is inactive.

For a Send Operation

Send operations work similarly:

```
where (source < 30)
   [index]dest = source;
```

The **where** statement "turns off" positions 3 and 4, as shown in Figure 41. But it turns them off only for *sending* purposes. Elements in inactive positions cannot send values, but they can receive values from elements in active positions. Thus, the serial C* version of this statement would be:

```
[1]dest = [0]source;
[3]dest = [1]source;
[0]dest = [2]source;
```

The results are shown in Figure 41; the arrows show how the value in **[1]source** is sent to **[3]dest**.

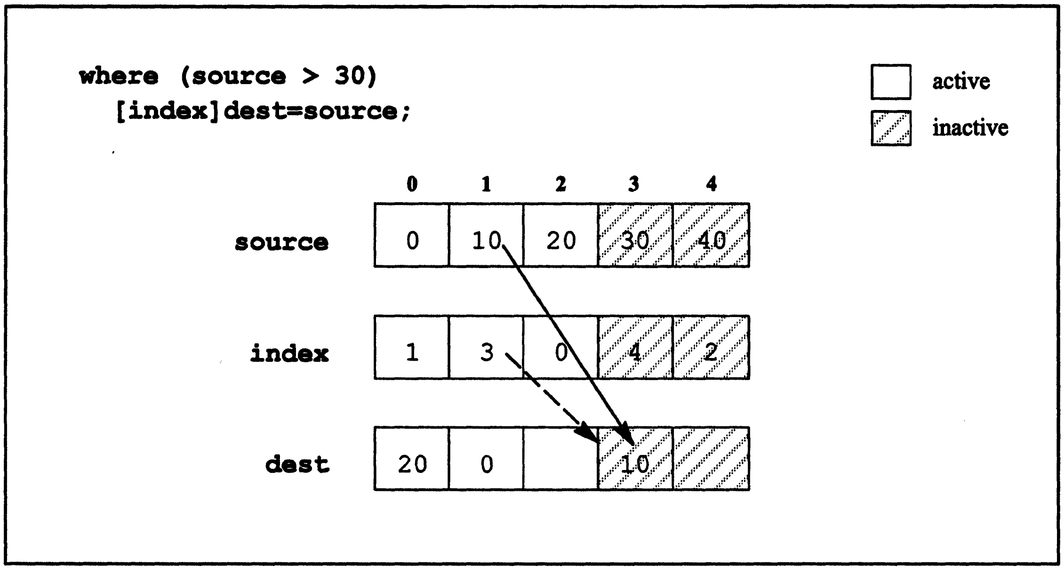

Figure 41. A send operation with inactive positions

Note these results:

- **[1]source** sends its value to **[3]dest**, even though position [3] is inactive, because position [1] is still active.
- **[4]source** does not send its value to **[2]dest** because position [4] is inactive.

One way to look at the concept of inactive positions in these situations is that the parallel variable without the parallel left index is the one doing the work (sending or getting). When a position is made inactive, it can't do work, but it can have work done to it. Thus:

- In a send operation, the inactive position can't send, but other positions can send to it.
- In a get operation, the inactive position can't get, but other positions can get from it.

Send and Get Operations in Function Calls

As we mentioned in Section 8.2, you should be careful about passing a parallel variable by value to a function that involves the parallel variable in a send or get operation. If there are inactive positions when the function is called, the results may not be what you expected.

For example, suppose we define the following function:

```
int:current get_op(int:current source, int:current index)
{
   return ([index]source);
}
```

If we use the data and the context from Figure 40, we get the results shown in Figure 42.

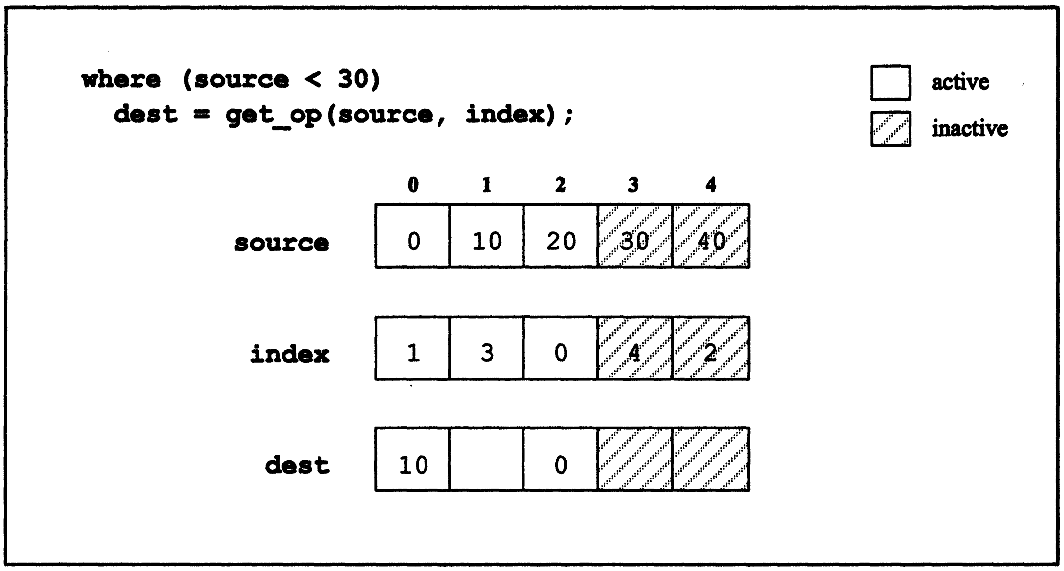

Figure 42. A function that includes a get operation

Note the difference in results between Figure 40 and Figure 42: In Figure 40, **[1]dest** got its value from **[3]source**, even though position [3] was inactive. In Figure 42, **[1]dest** receives an undefined value. This happens because the compiler makes a copy of a parallel variable when it is passed by value, and elements at inactive positions receive undefined values.

The solution is to pass **source** by reference. In that case, the compiler does not make a copy of the parallel variable, and the function can gain access to values at inactive positions.

Note that in send operations it is the **dest** parallel variable that should be passed by reference, since positions can send to an inactive destination.

10.1.7 Mapping a Parallel Variable to Another Shape

One use of the parallel left index is to map a parallel variable into another shape. Consider the situation shown in Figure 43.

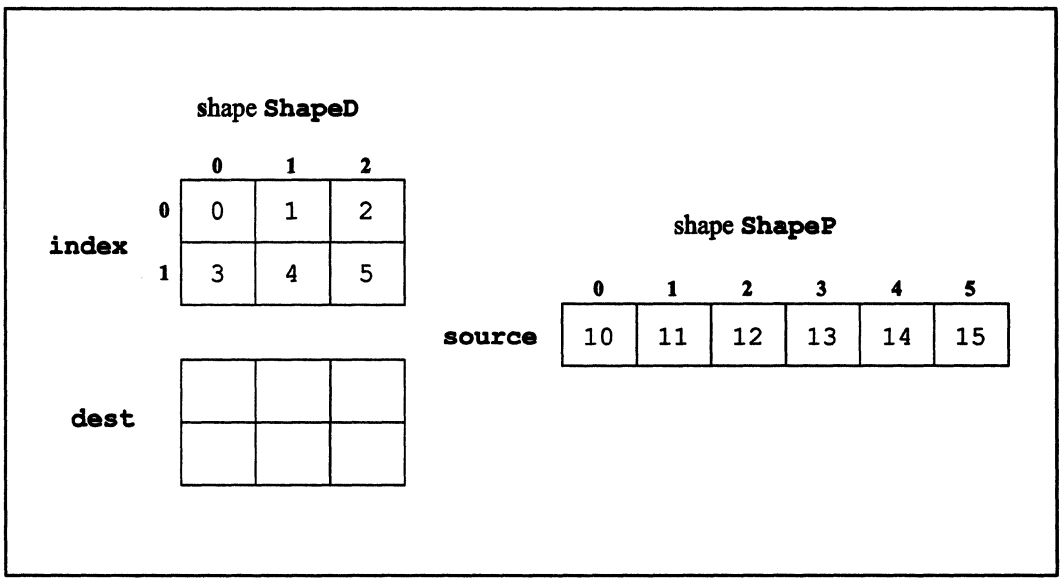

Figure 43. Two shapes

The statement:

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

has the same interpretation as before: Elements of **dest** get values of **source**, based on the value in the corresponding element of **index**. But in this situation, we are essentially mapping **source** into shape **ShapeD**, based on **index**. **ShapeD** must be the current shape.

Since the values in **index** are the same as the coordinates for **ShapeP**, the assignment is straightforward: the value of **index** for position [0][0] is 0; this value is used as an index into the elements of **source**. The value of element [0] of **source** is 10; therefore, 10 is assigned to element [0][0] of **dest**.

The mapping occurs only for the specified operation; it does not permanently affect the parallel variable being mapped. For example, **source** remains of shape **ShapeP** after the operation above.

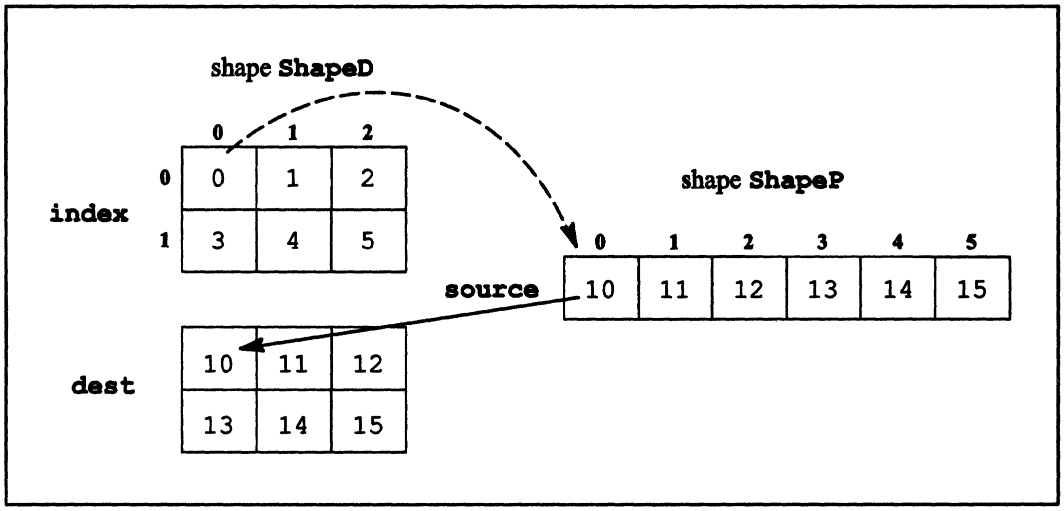

Figure 44. Mapping a parallel variable to another shape

If a parallel variable is not of the current shape, you can use a parallel left index to map it to the current shape and then operate on it. For example:

```
shape [64][64]ShapeD;
int:ShapeD index, dest;
shape [16384]ShapeP;
int:ShapeP source;

/* Code to initialize variables omitted. */

main()
{
   with (ShapeD) {
      [0][0]dest += source;        /* This doesn't work—source
                                      is the wrong shape. */
      [0][1]dest += [index]source;  /* This does work. */
```

```
        }
    }
```

Only active elements of a parallel left index participate in the indexing. If we add a **where** statement to the code example above and assume the data shown in Figure 43:

```
/* ... */
with (ShapeD) {
    where (index != 0)
        [0][0]dest += [index]source;
}
```

the value of element [0] of **source** is not included in the summation.

10.1.8 Limitation of Using Parallel Variables with a Parallel Left Index

A parallel variable with a parallel left index is a modifiable lvalue; therefore, it can appear as the left operand of assignment operators, as the operand of prefix or postfix ++ or --, and in all cases where an rvalue is needed. You cannot, however, take the address of it using the **&** operator. (In general, this would require a parallel pointer handle, which isn't supported in C*.)

10.1.9 What Can Be Left-Indexed

Parallel left indexing follows the general rules about performing parallel operations within the current shape; see Section 4.4. Specifically:

- If an expression is of the current shape, you can always left-index it.
- If an expression is not of the current shape, you can left-index it if it is any of the following:
    - A simple identifier.
    - A per-processor array that is not of the current shape, if it is right-indexed by a scalar value. (You cannot left-index an array that is not of the current

shape if it has a parallel right index, because that would require a parallel operation on a variable not of the current shape.)

- A parallel variable with the `&` operator applied to it to take its address.
- A member of a parallel structure or union that is not of the current shape (so long as the member is not an aggregate type, such as another structure or union).

10.1.10 An Example: Adding Diagonals in a Matrix

The example in this section uses a parallel left index and the `+=` reduction assignment operator to add diagonals in a matrix. It uses the data shown in Figure 45.

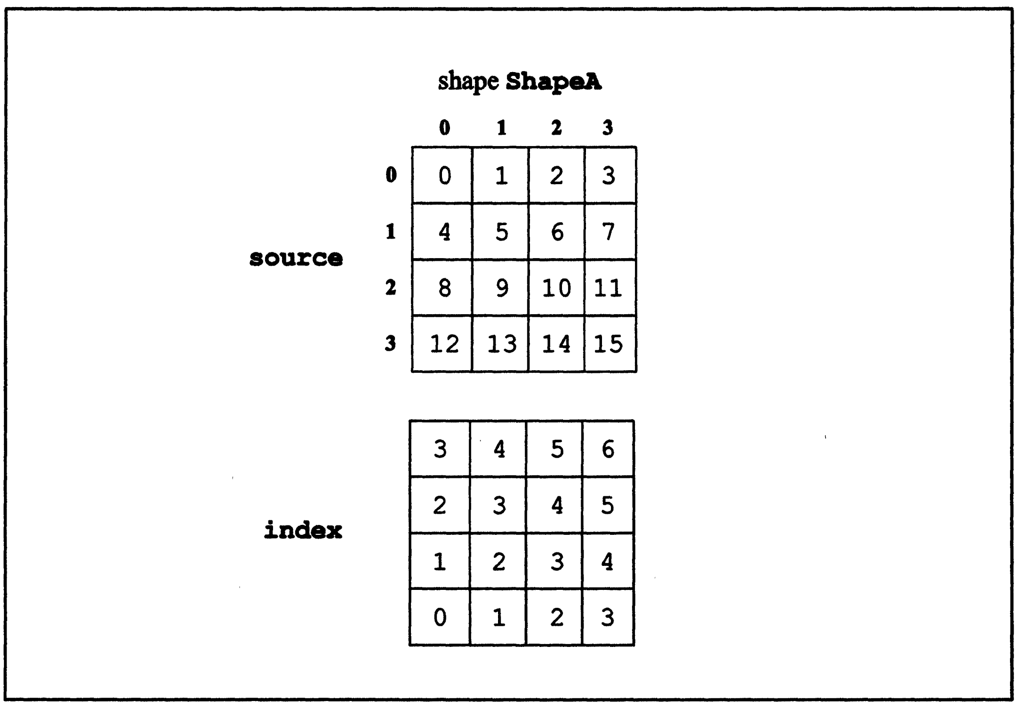

Figure 45. Two 4-by-4 parallel variables

The task is to add the values of `source` in the diagonals of the matrix. The following code accomplishes this.

```
shape [4][4]ShapeA;
shape [7]ShapeB;      /* Not legal shapes—for example purposes
                         only */
int:ShapeA source, index;
int:ShapeB dest = 0;

/* Code to initialize the parallel variables omitted */

main()
{
   with (ShapeA)
      [index]dest += source;
}
```

As you can see, the actual computation is quite simple, once the data has been set up properly. Let's look in detail at the statement:

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

First, note that the statement is legal, even though **dest** is not of shape **ShapeA**, since **dest** is left-indexed by a parallel variable that *is* of that shape. The statement says: *Use* **index** as *an index into* **dest** for *sending values of* **source***; if there are potential collisions, add the values of* **source**. So, for example, element [0][0] of parallel variable **source** is assigned to element [3] of **dest**, because the value of the corresponding element of **index** is 3. Element [1][1], element [2][2], and element [3][3] are also assigned to element [3] of **dest**. They are all added, thus avoiding collisions.

The other elements of **source** are also assigned to **dest**, based on the value of the corresponding elements of **index**. The result is the addition of the diagonals. Figure 46 shows the results, highlighting the values that go into **[3]dest**.

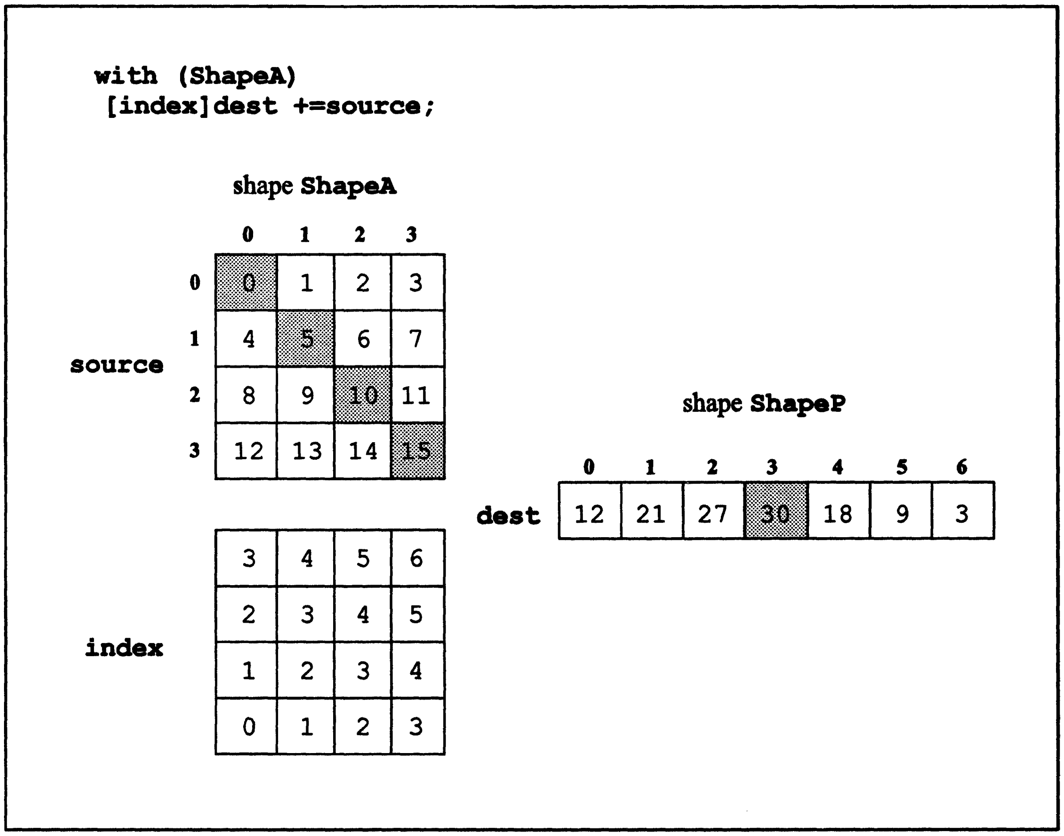

Figure 46. Using parallel left indexing to add the diagonals of a matrix

10.2 Using the pcoord Function

C* includes a new library function called **pcoord**, which is especially useful when combined with parallel left indexing. 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. For example,

```
shape [65536]ShapeA;
int:ShapeA p1;

main()
{
```

```
    with (ShapeA)
        p1 = pcoord(0);
}
```

initializes **p1** as shown in Figure 47.

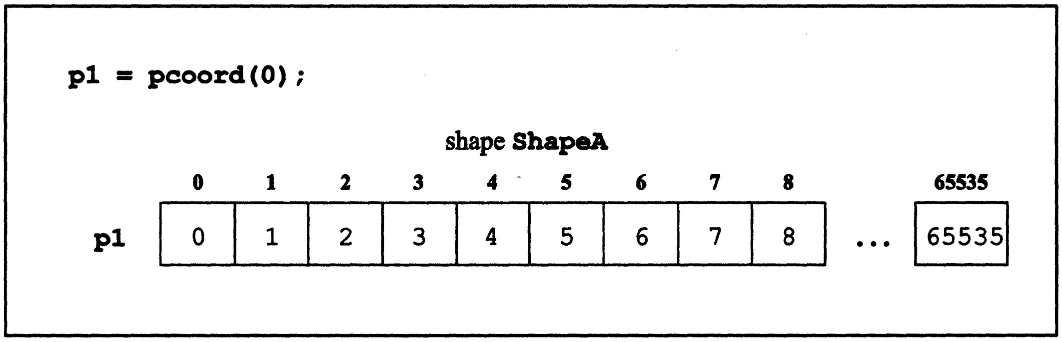

Figure 47. The use of **pcoord** with a 1-dimensional shape

Likewise, for a 2-dimensional shape,

```
shape [4][4096]ShapeB;
int:ShapeB p2;

main()
{
    with (ShapeB)
        p2 = pcoord(1);
}
```

initializes **p2** as shown in Figure 48.

p2 = pcoord(1);

p2	**0**	**1**	**2**	**3**		**4095**
0	0	1	2	3	...	4095
1	0	1	2	3	...	4095
2	0	1	2	3	...	4095
3	0	1	2	3	...	4095

Figure 48. The use of **pcoord** with axis 1 of a 2-dimensional shape

Similarly,

```
with (ShapeB)
   p2 = pcoord(0);
```

initializes **p2** as shown in Figure 49.

p2 = pcoord(0);

p2	**0**	**1**	**2**	**3**		**4095**
0	0	0	0	0	...	0
1	1	1	1	1	...	1
2	2	2	2	2	...	2
3	3	3	3	3	...	3

Figure 49. The use of **pcoord** with axis 0 of a 2-dimensional shape

The **pcoord** function provides a quick way of creating a parallel left index for mapping a parallel variable into another shape. For example:

```
shape [16384]ShapeA, [16384][4]ShapeB;
int:ShapeA source;

/* Code to initialize source omitted. */

main()
{
   with (ShapeB) {
      int:ShapeB index, dest;
      index = pcoord(0);
      dest = [index]source;
   }
}
```

Rather than assign the results of **pcoord** to a parallel variable, you can simply use it as the parallel left index itself:

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

The index of the specified axis of the current shape is generated by **pcoord**. This index is used as an index for selecting elements of a parallel variable of another shape. The values of these elements are assigned to elements of a parallel variable of the current shape.

10.2.1 An Example

This example uses **pcoord** to transpose a matrix—in other words, to turn its rows into columns and its columns into rows. For example, consider the simple 3-by-3 parallel variable called **matrix** shown on the left in Figure 50. (Note that this is an illegal parallel variable; we use it simply because it's easy to visualize.) The task is to turn it into the new matrix shown on the right.

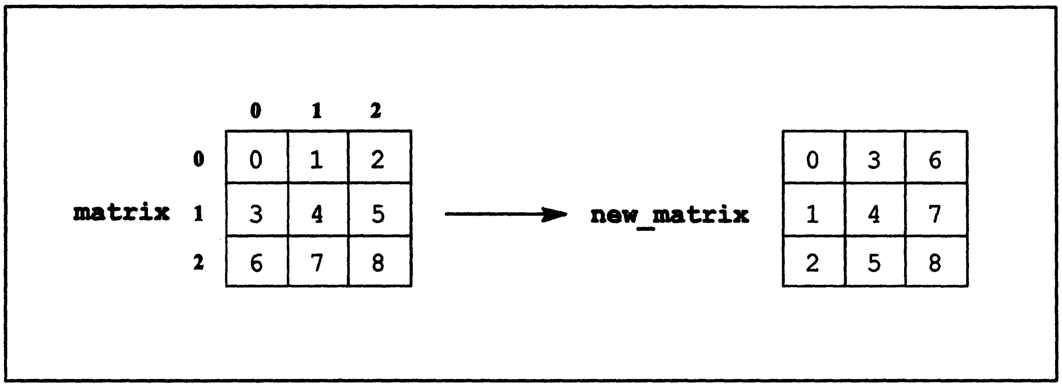

Figure 50. Transposing a 3-by-3 matrix

This can be done by reversing the axes for the parallel variable **matrix**. For example, **[0][1]matrix** (which contains the value 1) becomes element **[1][0]** of a new parallel variable. To do this for a 256-by-256 matrix, use **pcoord** as follows:

```
Shape [256][256]ShapeA;
int:ShapeA matrix, new_matrix;

main()
{
   with (ShapeA)
      [pcoord(1)][pcoord(0)]new_matrix = matrix;
}
```

The statement

```
[pcoord(1)][pcoord(0)]new_matrix = matrix;
```

says: *Assign each element of* **matrix** *to* **new_matrix**, *but reverse the axis numbering.* Thus, in serial C* code:

```
[0][0]new_matrix = [0][0]matrix;
[0][1]new_matrix = [1][0]matrix;
[0][2]new_matrix = [2][0]matrix;
[1][0]new_matrix = [0][1]matrix;   /* And so on */
```

except that all operations take place at the same time. This algorithm can be generalized for use in a function with any 2-dimensional parallel variable:

```
void transpose(float:current *matrixp,
      float:void *new_matrixp)
{
   [pcoord(1)][pcoord(0)]*new_matrixp = *matrixp;
}
```

Note the following about **transpose**:

- It passes two pointers to parallel variables. **matrixp** is a pointer to a parallel variable of the current shape; we pass a pointer rather than the parallel variable itself to avoid having to make a copy of the variable. **new_matrixp** is a pointer to a parallel variable of a new shape; we *must* pass a pointer in this case because we will be modifying the variable—therefore, it can't be passed by value.
- We use a second shape so that the function can work with a matrix that isn't square. For example, if the current shape is 256 by 512, make **new_matrixp** a pointer to a parallel variable of a shape that is 512 by 256.
- The variable pointed to by **matrixp** is assigned to the variable pointed to by **new_matrixp**, and this variable has its coordinates reversed.

10.3 The pcoord Function and Grid Communication

When used with parallel left indexing, **pcoord** provides the grid communication capabilities we discussed at the beginning of this chapter.

Consider the following statement:

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

This statement says: *Each active element of* **dest** *is to get the value of* **source** *that is in the position one coordinate higher along axis 0.* You can either add or subtract a scalar value from **pcoord** in the left index. Which operation you choose determines the direction of the communication; the value added or subtracted specifies how many positions along the axis the values are to travel. Note, however, that the values must stay within the border of the grid; the behavior is undefined if **dest** tries to get a nonexistent element of **source**.

You can use **pcoord** for a send operation as well as for a get operation; send and get operations are discussed in Section 10.1. For example:

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

This statement says: *Send the value of the* **source** *element to the* **dest** *element that is one position higher along axis 0.*

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 axes in this kind of statement provides redundant information. 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, the following is equivalent to the above statement:

```
dest = [. - 2][. + 1]source;
```

10.3.1 Grid Communication without Wrapping

As we noted above, behavior is undefined when elements try to get or send beyond the border of the grid. This means that the statements shown so far are not especially useful, because they do not solve this problem. What happens to the elements of **dest** in row 0 when they try to get from **[pcoord(0)-1]**—that is, from beyond the border of the grid?

For this kind of statement to work, you must first use a **where** statement to turn off positions that would otherwise get or send beyond the border of the grid. For example, if you want elements to get from 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 turn off positions 0 and 1, because elements in these positions would otherwise attempt to get nonexistent values. The following code accomplishes this:

```
where (pcoord(0) > 1)
   dest = [. - 2]source;
```

If you want to get from a parallel variable two coordinates higher along axis 0 (position 0 gets from position 2, and so on), you can use the **dimof** intrinsic function to determine the number of positions along the axis. For example:

```
where (pcoord(0) < (dimof(ShapeA, 0) - 2))
    dest = [. + 2]source;
```

Note that you must subtract 2 from the result returned by **dimof** to turn off the correct number of positions. If **dimof** returns 1024, the positions are numbered 0 through 1023. To turn off positions 1022 and 1023, you must subtract 2 from 1024 and specify that the result of calling **pcoord** is to be less than this.

10.3.2 Grid Communication with Wrapping

To perform grid communication in which the values "wrap" back to the other side of the grid, we once again need to use the **dimof** intrinsic function. Consider the following 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. For example, if **(. + 2)** is 502 in a 1024-position axis, the result of **(502 %% 1000)** is 502. When step 1 returns a value equal to or greater than the number of coordinates along the axis, step 2 achieves the desired wrapping. For example, element [1022] of **dest** attempts to get from element [1024] of **source**, which is beyond the border of the grid. 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.

Similarly,

```
dest = [(. - 2) %% dimof(ShapeA, 0)]source;
```

provides wrapping to the high end of the axis. For this statement, let's look at the case where `[0]dest` tries to get a value from the element of `source` that is two lower along axis 0. If there are 1024 coordinates along the axis, this produces the expression `(-2 %% 1024)` for the left index of `source`. Following the procedure for `%%` shown on page 52, we find that the result of this expression is 1022. This is the element of `source` from which `[0]dest` gets its value.

Note that you cannot use the standard C operator `%` to perform these operations, because different implementations of `%` can give different answers when one or both of its operands is negative. The `%%` operator guarantees that the sign of the answer is the same as the sign of the denominator, which is what is required.

Part III
C* Communication Functions

Chapter 11

Introduction to the C* Communication Library

Part III of this guide describes a set of C* library functions that provide different kinds of communication. For example, these functions allow you to:

- Send values of parallel variable elements to other elements of the same shape.
- Send values of parallel variable elements of one shape to elements of another shape.
- Perform different kinds of computation on values while sending them to elements of the same or a different shape.
- Send data from parallel variable elements to a front-end variable, and from a front-end variable to a parallel variable element.
- Send data from a parallel variable to a front-end array, or from a front-end array to a parallel variable.

Of course, you can perform similar kinds of communication using features of C* itself; see Chapter 10. These library functions supplement, and in many cases overlap, the communication features contained in the language itself. Several of them are particularly useful when the rank of a shape is not known until run time; in that situation, you cannot use left indexing to specify a parallel variable element, because you cannot specify values for all the axes when you write the program. The functions, however, provide a way to manipulate such data.

This chapter introduces the methods of communication available using C* library functions, and gives an overview of these functions.

Include the header file <cscomm.h> in programs that call any of the functions discussed in Part III. The functions are part of the C* run-time system, and are linked in to your program by default.

11.1 Two Kinds of Communication

There are two different kinds of communication in C*: *grid* and *general.*

11.1.1 Grid Communication

In grid communication, elements of parallel variables in the same shape communicate in regular patterns by using their coordinates. In other words, values of all elements in a parallel variable move the same number of positions in the same direction—for example, each element sends its value to the the element of another parallel variable that is two coordinates higher along axis 0.

The following functions implement grid communication:

- **from_grid**
- **from_grid_dim**
- **from_torus**
- **from_torus_dim**
- **to_grid**
- **to_grid_dim**
- **to_torus**
- **to_torus_dim**

In addition, the **pcoord** function, which we discussed in Chapter 10, can be used in certain kinds of grid communication.

Grid communication is discussed in Chapter 12.

11.1.2 General Communication

General communication allows any parallel variable element to send its value to any other element, whether or not they are of the same shape, and whether or not the pattern of communication is regular. It also allows the front end to read values from and write values to parallel variables. This kind of communication uses a position's *send address* rather than its coordinates. The send address is a combination of a position's shape and coordinates that uniquely identifies the position among all positions in all shapes. General communication is more versatile than grid communication, but it is also slower. It achieves the same result as parallel left-indexing a parallel variable; see Chapter 10.

General communication is implemented by the following C* functions:

- **`make_send_address`**
- **`send`**
- **`get`**
- **`read_from_position`**
- **`read_from_pvar`**
- **`write_to_position`**
- **`write_to_pvar`**
- **`make_multi_coord`**

These functions are discussed in Chapter 14.

11.2 Communication and Computation

Many C* functions perform computations or combining operations on the parallel values they transmit. Most of these functions involve grid communication. For example, the **`scan`** function lets you combine values of specified elements of a parallel variable along an axis of a shape. You can add these values, for example, multiply them, or take the minimum or maximum. The following C* library functions provide communication and computation:

- **`scan`**
- **`spread`**
- **`copy_spread`**

- **multispread**
- **copy_multispread**
- **enumerate**
- **rank**
- **reduce**
- **copy_reduce**
- **global**

These functions are discussed in Chapter 13.

Chapter 12

Grid Communication

As we mentioned in the previous chapter, there are two ways for data to be communicated from one position to another within a shape: by using the absolute address (called the *send address*) of the position, or by using the position's coordinates within the shape. Within-shape communication in regular patterns that uses positions' coordinates is referred to as *grid communication*, since the coordinates can be thought of as locating positions on an *n*-dimensional grid.

This chapter describes C* library functions that provide grid communication. These functions are faster than the general communication functions described in Chapter 14. If you use any of the functions discussed in this chapter, include the file `<cscomm.h>` in your program. You can also achieve grid communication by using the pcoord function, as described in Chapter 10.

All grid communication functions are overloaded so that they can be used with any arithmetic or aggregate data type.

12.1 Aspects of Grid Communication

There are several aspects to grid communication to consider before using these functions:

- Axis
- Direction
- Distance
- Border Behavior
- Behavior of Inactive Positions

12.1.1 Axis

Grid communication functions let parallel variable elements communicate along any axis of a shape. In a two-dimensional shape like Figure 51, for example, you can specify that elements communicate along axis 0 or along axis 1.

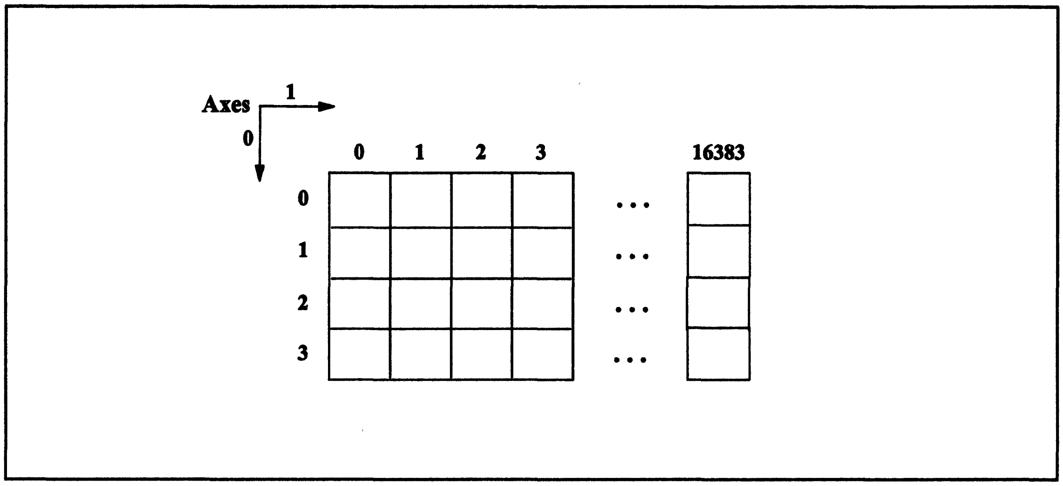

Figure 51. A two-dimensional shape

The functions **`from_grid`**, **`to_grid`**, **`from_torus`**, and **`to_torus`** allow communication along more than one axis—for example, an element could transmit a value to another element by sending it down axis 0, then across axis 1.

12.1.2 Direction

Parallel variable elements can also communicate in either direction along an axis using grid communication. In Figure 51, for example, parallel variable elements at position [0][2] can communicate along axis 1 with elements to the right (position [0][3]) or to the left (position [0][1]).

12.1.3 Distance

Parallel variables can communicate at any distance along an axis. For example, parallel variable elements at position [0][0] in Figure 51 can communicate with elements at position [0][16383].

12.1.4 Border Behavior

What happens when a parallel variable element at position [0][16383] in Figure 51 tries to get a value from the right—off the border of the grid? The behavior of grid communication at the border is handled in different ways by different functions. Specifically:

- In the functions **from_grid**, **from_grid_dim**, **to_grid**, and **to_grid_dim**, you can specify a value that the element is to receive when it tries to get a value from beyond the border. This value is referred to as the *fill value*.
- In the functions **from_torus**, **from_torus_dim**, **to_torus**, and **to_torus_dim**, the element receives the value from the opposite border of the grid—in this case, the element at position [0][16383] gets its value from position [0][0]. This is known as *wrapping*.

12.1.5 Behavior of Inactive Positions

What happens when positions in the grid are inactive? For example, a parallel variable element at position [0][0] tries to get the value of an element at position [0][1], but position [0][1] is inactive.

Different functions handle inactive positions in different ways, depending on whether parallel variables are seen as sending their values to other positions, or getting values from other positions. The distinction is the same one made for parallel left indexing; see Section 10.1.6. Specifically:

- In a get operation, a parallel variable element in an active position can get a value from an element in an inactive position, but an element in an inactive position cannot get a value from any position. The functions **from_grid**, **from_grid_dim**, **from_torus**, and **from_torus_dim** use get operations.

- In a send operation, a parallel variable element in an active position can send a value to an element in an inactive position, but an element in an inactive position cannot send its value. The functions **to_grid**, **to_grid_dim**, **to_torus**, and **to_torus_dim** use send operations.

  Note that the issue of getting from or sending to inactive positions requires passing some parallel variables in the grid communication functions by reference, rather than by value. See Chapter 10 for a discussion of this issue.

Table 3 summarizes the features of the grid communication functions.

Table 3. Features of grid communication functions

Function	Multiple Axes?	Wrapping?	Get or Send?
from_grid	Yes	No	Get
from_grid_dim	No	No	Get
from_torus	Yes	Yes	Get
from_torus_dim	No	Yes	Get
to_grid	Yes	No	Send
to_grid_dim	No	No	Send
to_torus	Yes	Yes	Send
to_torus_dim	No	Yes	Send

12.2 The from_grid_dim Function

Use the **from_grid_dim** function to communicate along one axis of a grid, without wrapping. **from_grid_dim** is a get operation, as described in Chapter 10.

12.2.1 With Arithmetic Types

Like all grid communication functions, **from_grid_dim** can be used with arithmetic data types, as well as with parallel structures and parallel arrays. The version of **from_grid_dim** for arithmetic data types has the following definition:

```
type:current  from_grid_dim (
    type:current *sourcep,
    type:current value,
    int axis,
    int distance);
```

where:

sourcep is a scalar pointer to the parallel variable from which values are to be obtained. The parallel variable can be of any arithmetic type; it must be of the current shape.

value is a parallel variable of the current shape whose values are to be used when elements try to get values from beyond the border of the grid. The parallel variable must be of the same arithmetic type as the parallel variable pointed to by **sourcep**.

axis specifies the axis along which the communication is to take place.

distance specifies how many positions along the axis the values are to travel. For example, if **distance** is 2, each parallel variable element gets a value from an element whose position is two greater along the specified axis. **distance** can be a negative number, which reverses the direction in which the data is to travel.

from_grid_dim returns the source values in their new positions. You can assign these values to a parallel variable of the current shape and of the same arithmetic type as the source parallel variable; this parallel variable can be viewed as the parallel variable that is doing the "getting."

Note the difference between **from_grid_dim** and the corresponding use of **pcoord** described in Chapter 10: **pcoord** does not provide a fill value when an element tries to get from beyond the border.

Examples

Figure 52 shows three parallel variables of the same shape (their shape, like others shown in the chapter, is smaller than would be legal in C*, so that it's easier to visualize what is happening).

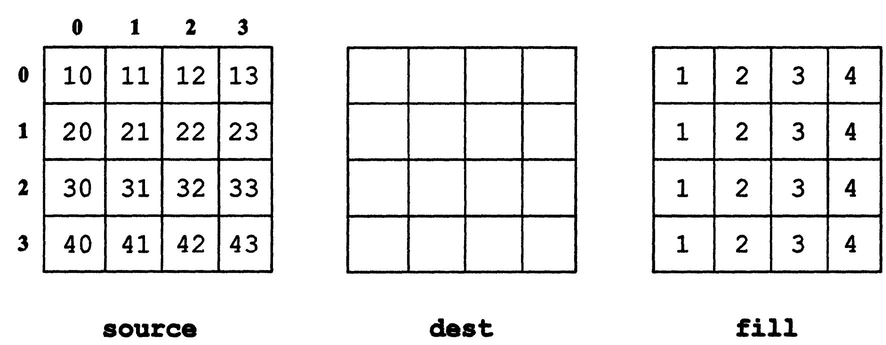

Figure 52. Three parallel variables of shape **ShapeA**

The goal is for **dest** to get values of the parallel variable pointed to by **sourcep** that are one position lower along axis 0. This is equivalent to scalar C* statements like the following (except that all operations happen at the same time):

```
[1][0]dest = [0][0]source;
[2][0]dest = [1][0]source;
[3][0]dest = [2][0]source;
[1][1]dest = [0][1]source;    /* . . . and so on */
```

In the case where **dest** tries to get a value of **source** from beyond the border (for example, the **dest** element at position [0][0]), it is to use the value from the corresponding element of **fill**.

The following code accomplishes this (for a shape of legal size):

```
#include <cscomm.h>

shape [256][256]ShapeA;
int:ShapeA source, dest, fill;

/* Code to initialize parallel variables omitted. */

main()
{
   with (ShapeA)
```

```
        dest = from_grid_dim(&source, fill, 0, -1);
}
```

Figure 53 shows the results.

Note that we use –1 for the **distance** argument, even though the values move to higher-numbered positions along the axis. As mentioned above, **from_grid_dim** is a get operation; in this case, the element in the higher-numbered position is viewed as getting the data from the lower-numbered position, and that is why a negative distance is used.

Note also the values of **fill** that are used when **dest** attempts to get from beyond the border of the grid.

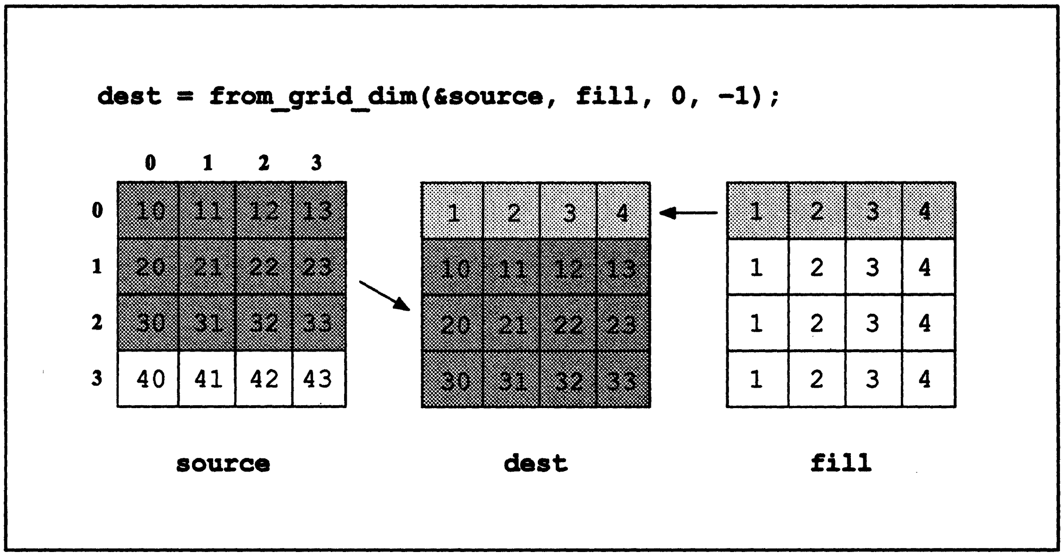

Figure 53. An example of the **from_grid_dim** function

Now let's take the data in Figure 53 and move the values in **dest** two positions lower along axis 1, but leaving them in **dest**. In scalar C* code:

```
[0][0]dest = [0][2]dest;
[0][1]dest = [0][3]dest;
[1][0]dest = [1][2]dest;   /*  . . . and so on */
```

In this case, the source parallel variable is the same as the destination parallel variable. This is legal. The following statement does the job:

```
dest = from_grid_dim(&dest, fill, 1, 2);
```

A positive integer is used for the distance, because the elements in the lower-numbered positions along the axis are getting data from the elements in the higher-numbered positions.

Figure 54 shows the results.

Note that the elements of **dest** at positions [*n*][2] and [*n*][3] (where *n* is any axis 0 coordinate) are assigned the values from the corresponding elements of **fill**, because they attempt to get values from beyond the border of the grid.

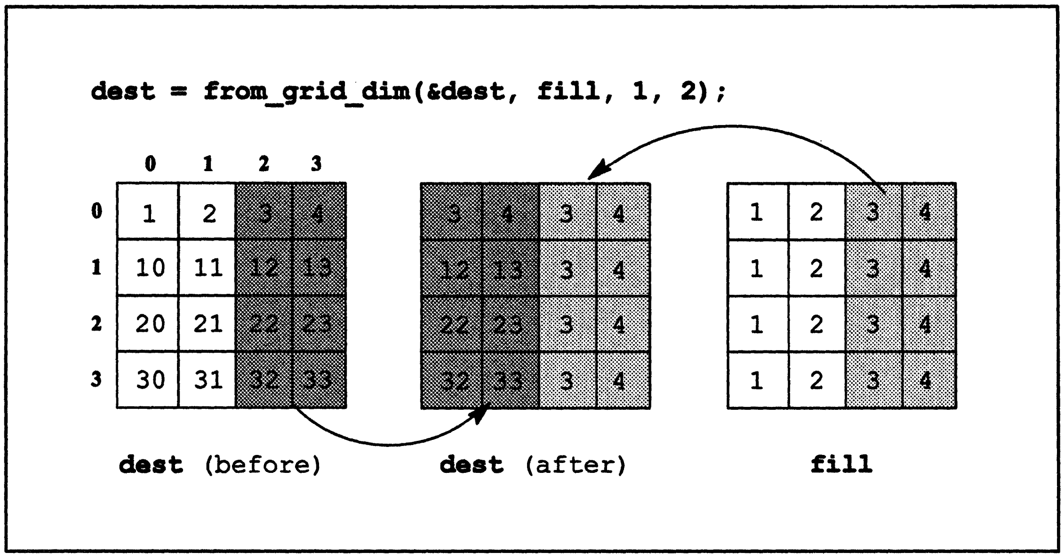

Figure 54. Another example of the **from_grid_dim** function

When Positions Are Inactive

Finally, let's see what happens when positions in a shape are inactive. The following code fragment makes position [2] inactive, using the simple data in Figure 55, and then calls **from_grid_dim**:

```
where (source != 7)
  dest = from_grid_dim(&source, fill, 0, -1);
```

Figure 55 shows the results.

```
where (source != 7)
  dest = from_grid_dim(&source, fill, 0, -1);
```

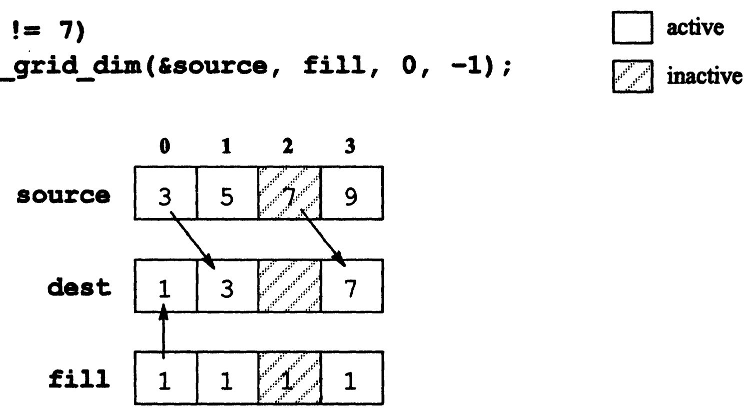

Figure 55. An example of **from_grid_dim** when a position is inactive

Since **from_grid_dim** is a get operation, the following rules apply:

- Elements at active positions can get values from elements at inactive positions.
- Elements at inactive positions cannot perform any gets at all.

Note how these rules are applied in Figure 55:

- Position [2] is inactive, so it doesn't get a value from position [1]. (It keeps the value it had before the operation.)
- Position [3] gets a value from position [2], even though position [2] is inactive.

12.2.2 With Parallel Data of Any Length

The definition of **from_grid_dim** for parallel data of any length is as follows:

```
void from_grid_dim (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int axis,
    int distance);
```

In this version, the location pointed to by **destp** gets values from the location pointed to by **sourcep**, using the **axis** and **distance** arguments to determine the axis for the communication and how many positions along the axis the values are to travel. If **destp** tries to get from beyond the border of the grid, it gets values from the corresponding location pointed to by **valuep** instead. The locations pointed to by **destp**, **sourcep**, and **valuep** are all **length** bits long.

You can use this version of **from_grid_dim** to transfer data that is larger than the standard data types—typically, this data would be in a parallel array or parallel structure. Note that there is no return value, and the destination is specified as the first argument to the function.

For example, in the following code, **dest_struct** gets the values of **source_struct** that are four coordinates higher along axis 0. When this takes **dest_struct** beyond the border of the grid, it gets the corresponding values of **value_struct**.

```
#include <cscomm.h>

shape [65536]ShapeA;
struct S {
   int a;
   int b;
};
struct S:ShapeA source_struct, dest_struct, value_struct;

main()
{
   with (ShapeA)
      from_grid_dim(&dest_struct, &source_struct,
         &value_struct,boolsizeof(source_struct), 0, 4);
}
```

12.3 The from_grid Function

The **from_grid** lets data travel along more than one axis of the grid. Like **from_grid_dim**, it is a get operation.

12.3.1 With Arithmetic Types

The definition of **from_grid** (for the version that takes arithmetic types) is:

```
type:current from_grid (
    type:current *sourcep,
    type:current value,
    int distance_along_axis_0, ... );
```

where **sourcep**, **value**, and the return value are defined as they were for **from_grid_dim**.

The argument **distance_along_axis_0** specifies how many positions along this axis the data is to travel. As with **from_grid_dim**, the sign of the integer (positive or negative) indicates the direction of travel along the axis. The ellipsis (. . .) indicates a variable number of arguments. Each argument is an **int** that represents the distance along succeeding axes that the data is to travel. You must include as many arguments as there are axes in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0.

from_grid lets you combine movement along different axes. For example, in the previous section we used two calls to **from_grid_dim** so that each **dest** element got the value from the **source** element that was one position lower along axis 0 and two positions higher along axis 1. The following call to **from_grid** accomplishes the same thing:

```
dest = from_grid(&source, fill, -1, 2);
```

The **-1** argument specifies the direction and distance of the communication along axis 0; the **2** argument specifies the direction and distance of the communication along axis 1. The movement along axis 1 takes place after the movement along axis 0. That is, the **dest** elements first get the **source** elements one position lower along axis 0; the **dest** elements that are two positions lower along axis 1 then gets these values from these other **dest** elements.

Note an important difference between the single **from_grid** call and the two **from_grid_dim** calls, however. With **from_grid**, the fill value is inserted only after *all* data movement is completed. No fill values are inserted when elements try to get from beyond the border in intermediate steps. This ensures that elements of the destination parallel variable receive fill values from corresponding elements of the fill parallel variable. But it yields a different result from consecutive **from_grid_dim** calls, where the fill value is inserted for each call.

Figure 56 shows the results of the **from_grid** call shown above on the data in Figure 52. Compare these results with those for the two **from_grid_dim** calls shown in Figure 54 (the arrow on the left shows that **[0][2]source** ends up at **[1][0]dest**).

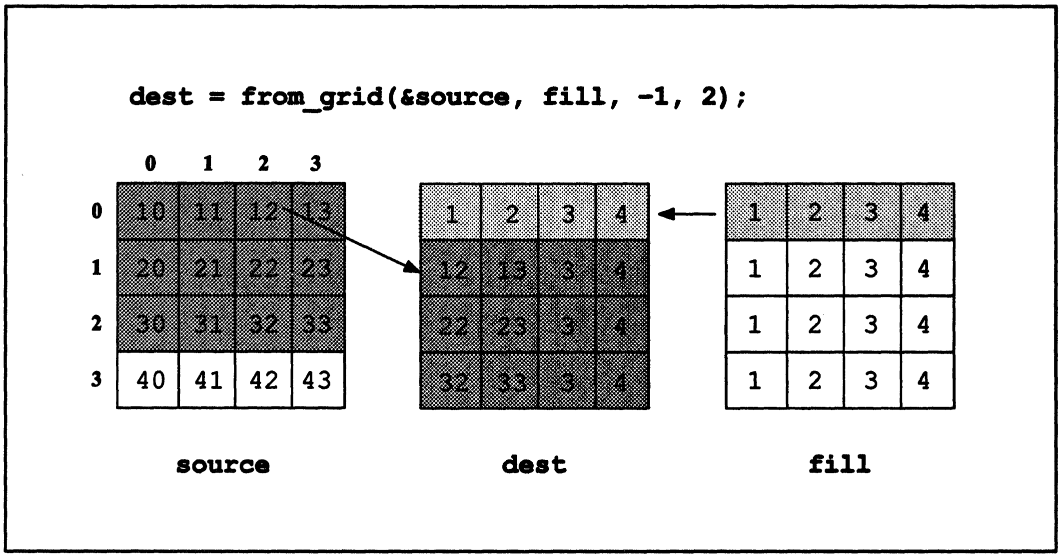

Figure 56. An example of the **from_grid** function

from_grid handles inactive positions in the same way that **from_grid_dim** does.

12.3.2 With Parallel Data of Any Length

Like **from_grid_dim**, **from_grid** has an overloaded version that can be used with parallel data of any length. Its definition is:

```
void from_grid (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int distance_along_axis_0, ... );
```

Once again, **destp**, **sourcep**, and **valuep** are pointers to parallel locations that are **length** bits long. Specify the data movement for each axis in the arguments

distance_along_axis_*n*. destp gets the value of **sourcep** based on these arguments, unless this brings it beyond the border of the grid, in which case it gets a value from the corresponding location pointed to by **valuep**.

12.4 The to_grid and to_grid_dim Functions

The **to_grid** and **to_grid_dim** functions are similar to **from_grid** and **from_grid_dim**, except that they are send operations instead of get operations. Both pairs of functions provide grid communication, with substitution of a fill value when the communication would otherwise go beyond the boundary of the grid. Both provide overloadings for arithmetic and aggregate types. The differences between the get operations and the send operations are:

- In the way the distance argument is interpreted
- In the way inactive positions behave

These differences are described in more detail below.

12.4.1 With Arithmetic Types

The definitions of **to_grid** and **to_grid_dim** (for the versions that take arithmetic types) are as follows:

```
void to_grid (
    type:current *destp,
    type:current source,
    type:current *valuep,
    int distance_along_axis_0, ... );

void to_grid_dim (
    type:current *destp
    type:current source,
    type:current *valuep,
    int axis,
    int distance);
```

where:

destp is a scalar pointer to the parallel variable to which values are to be sent. This parallel variable can be of any arithmetic type; it must be of the current shape.

source is the parallel variable that is to send its values. It can be of any arithmetic type; it must be of the current shape and of the same type as the parallel variable pointed to by **destp**.

valuep is a scalar pointer to a fill parallel variable whose values are to be used when elements of **source** try to send values to destinations beyond the border of the grid. It must be of the current shape and have the same type as **source**.

distance_along_axis_0
(for **to_grid**) specifies how many positions along axis 0 the values are to travel. For example, if **distance_along_axis_0** is 2, each parallel variable element of **source** sends a value to an element of the parallel variable pointed to by **destp** whose position is two greater along axis 0. Include a distance argument for each dimension in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0. The distance can be a negative number, which reverses the direction in which the data is to travel.

axis (for **to_grid_dim**) specifies the axis for the communication.

distance (for **to_grid_dim**) specifies how many positions along **axis** the values are to travel, as discussed in the description of **distance_along_axis_0**.

There is no return value.

Note the way that the **distance** argument is interpreted in send operations like **to_grid** and **to_grid_dim**. Specifying a positive integer for the distance sends values to higher-numbered positions. This is different from the behavior for get operations like **from_grid** and **from_grid_dim**, where specifying a positive integer for the distance gets values from higher-numbered positions.

When Positions Are Inactive

Since **to_grid** and **to_grid_dim** are send operations, the following rules apply when positions are inactive:

- Elements at active positions can send values to elements at inactive positions.
- Elements at inactive positions cannot send their values.
- Elements at border positions receive fill values even if they are inactive. This follows the general behavior of send operations, in which elements at inactive positions can be sent values.

Examples

The first example uses **`to_grid_dim`** to achieve the same result as the use of **`from_grid_dim`** shown in Figure 53. The goal is for **`source`** to send values to elements of **`dest`** that are one position higher along axis 0. When the sending goes beyond the border of the grid, values of the corresponding elements of **`fill`** are used instead. The following code accomplishes this:

```
to_grid_dim(&dest, source, &fill, 0, 1);
```

The results are shown in Figure 57.

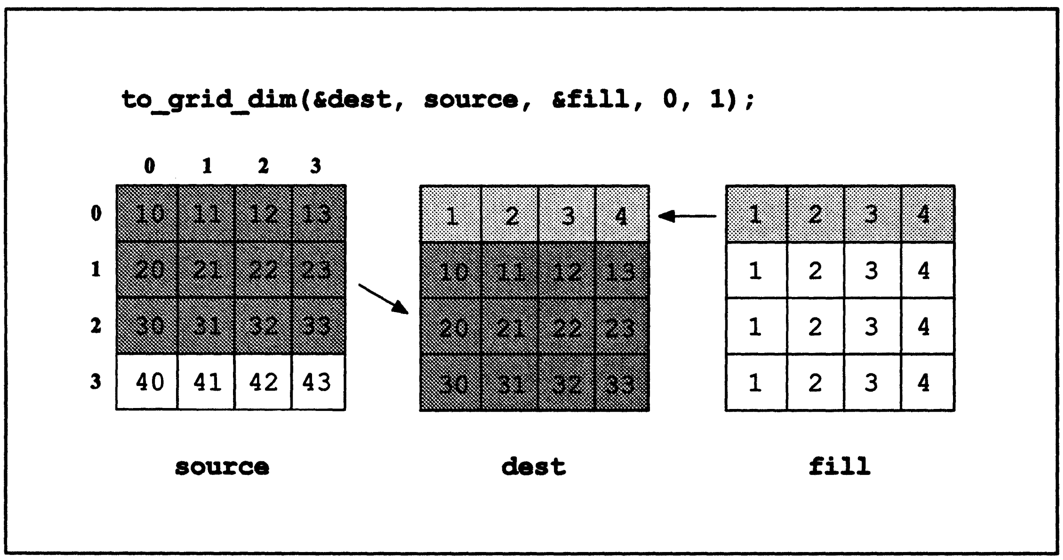

Figure 57. An example of the **`to_grid_dim`** function

Similarly, to obtain the same results as those shown in Figure 54 for **`for_grid_dim`**, use the following code:

```
to_grid_dim(&dest, dest, &fill, 1, -2);
```

These two calls to **to_grid_dim** are similar to the following call to **to_grid**:

```
to_grid(&dest, source, &fill, 1, -2);
```

Note, however, that, as with **from_grid**, the fill values for **to_grid** are inserted only after *all* data movement has occurred. In this case, this produces a slightly different result for the single **to_grid** call; see Figure 56.

In all cases, note that the difference from the corresponding **from_grid** or **from_grid_dim** call is that the sign of each distance argument is reversed.

The final example makes positions [0] and [2] inactive and then calls **to_grid_dim**:

```
where (source != 7)
  to_grid_dim(&dest, source, &fill, 0, 1);
```

Figure 58 shows the results.

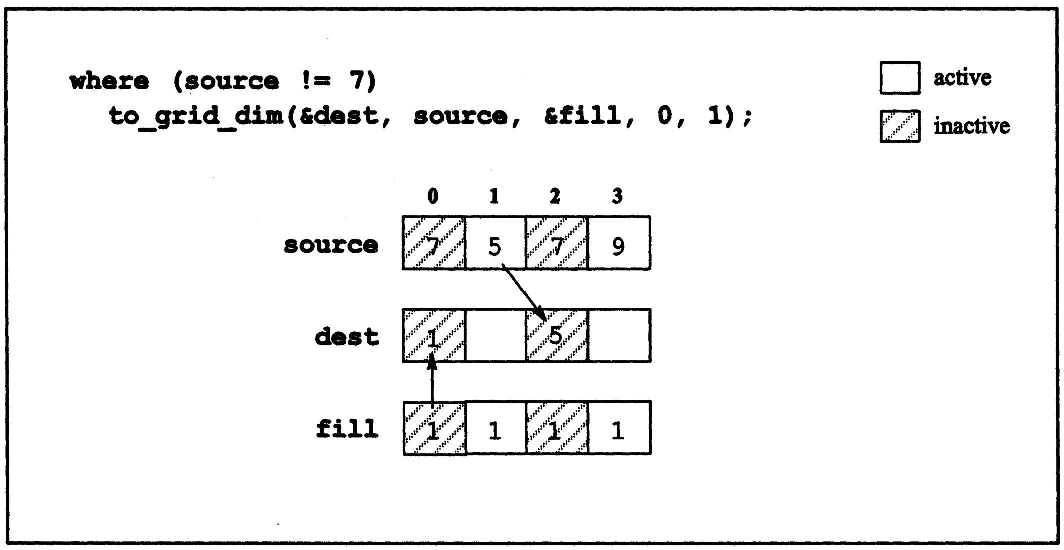

Figure 58. An example of **to_grid_dim** when position are inactive

Note how the rules for inactive positions and send operations are applied in Figure 58:

- **[0]source** and **[2]source** are at inactive positions, so they don't send their values to **[1]dest** and **[3]dest**.
- **[1]source** sends its value to **[2]dest**, even though position [2] is inactive.
- **[0]fill** sends its value to **[0]dest**, even though position [0] is inactive.

12.4.2 With Parallel Data of Any Length

The definitions of **to_grid** and **to_grid_dim** for parallel data of any length are as follows:

```
void to_grid (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int distance_along_axis_0, ... );

void to_grid_dim (
    void:current *destp,
    void:current *sourcep,
    void:current *valuep,
    int length,
    int axis,
    int distance);
```

These versions are useful if you want to transfer data in a parallel array or parallel structure. As with the corresponding versions of **from_grid** and **from_grid_dim**, the **length** argument specifies the length of the locations pointed to by **destp**, **sourcep**, and **valuep**. There is no return value, and the destination is specified as the first argument to the function.

12.5 The from_torus and from_torus_dim Functions

A *torus* is a doughnut-shaped surface. The C* "torus" functions (two more are discussed in the next section) use the grid as if it were wrapped into a torus, with the opposite borders of the grid connected. If a value is required from beyond the border, it comes from the other side of the grid. Thus, these functions don't need the fill value used in the "grid" functions, since there is never a case where an element will not be able to obtain a value because it is beyond the border.

Except for this difference, **from_torus** and **from_torus_dim** are equivalent to **from_grid** and **from_grid_dim**. As with the other grid functions, there are overloaded versions for use with all arithmetic and aggregate types.

12.5.1 With Arithmetic Types

The definitions of **from_torus** and **from_torus_dim** (for the versions that take arithmetic types) are as follows:

```
type:current from_torus (
    type:current *sourcep,
    int distance_along_axis_0, ... );

type:current from_torus_dim (
    type:current *sourcep,
    int axis,
    int distance);
```

Let's look at how the results change when we use these functions on data from previous sections.

For example, let's take the data from Figure 52 and use **from_torus_dim** instead of **from_grid_dim**. The goal is the same: **dest** elements are to get the values of **source** elements that are one position lower along axis 0:

```
dest = from_torus_dim(&source, 0, -1);
```

Note that **from_torus_dim** does not require a **valuep** argument, since values wrap from the other side of the grid. The results of this statement are shown in Figure 59. The arrows in the figure show the movement for two elements of **source**: **[0] [0] dest** wraps around to get the value of **[3] [0] source**, and **[2] [3] dest** gets the value of **[1] [3] source**.

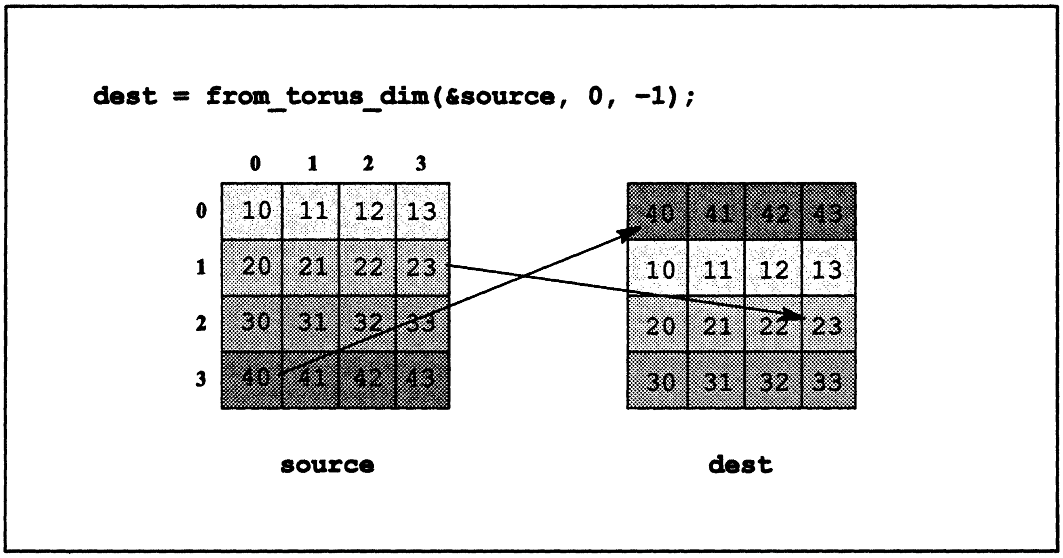

Figure 59. An example of the **from_torus_dim** function

Compare the results shown in Figure 59 with those for the equivalent **from_grid_dim** call, shown in Figure 53. The differences are only in the **dest** elements that are at position [0][*n*]. **from_grid_dim** puts the value of the corresponding element of **fill** into the **dest** element. **from_torus_dim** wraps around to the other side of the grid and has the **dest** elements get the values of the **source** elements at position [3][*n*].

Similarly, using the same **source** data, the following **from_torus** call:

```
dest = from_torus(&source, -1, 2);
```

produces the results shown in Figure 60. Compare these results with those shown in Figure 54, which are the results for the two **from_grid_dim** calls. Once again, **dest** elements that previously were assigned values of **fill** now get values of **source** elements from the other side of the grid. In Figure 60, the arrows show where the value of **[0] [3] source** ends up: after the movement along axis 0 **[1] [3] dest** gets it, and after the movement along axis 1 it ends up wrapping around to **[1] [1] dest**.

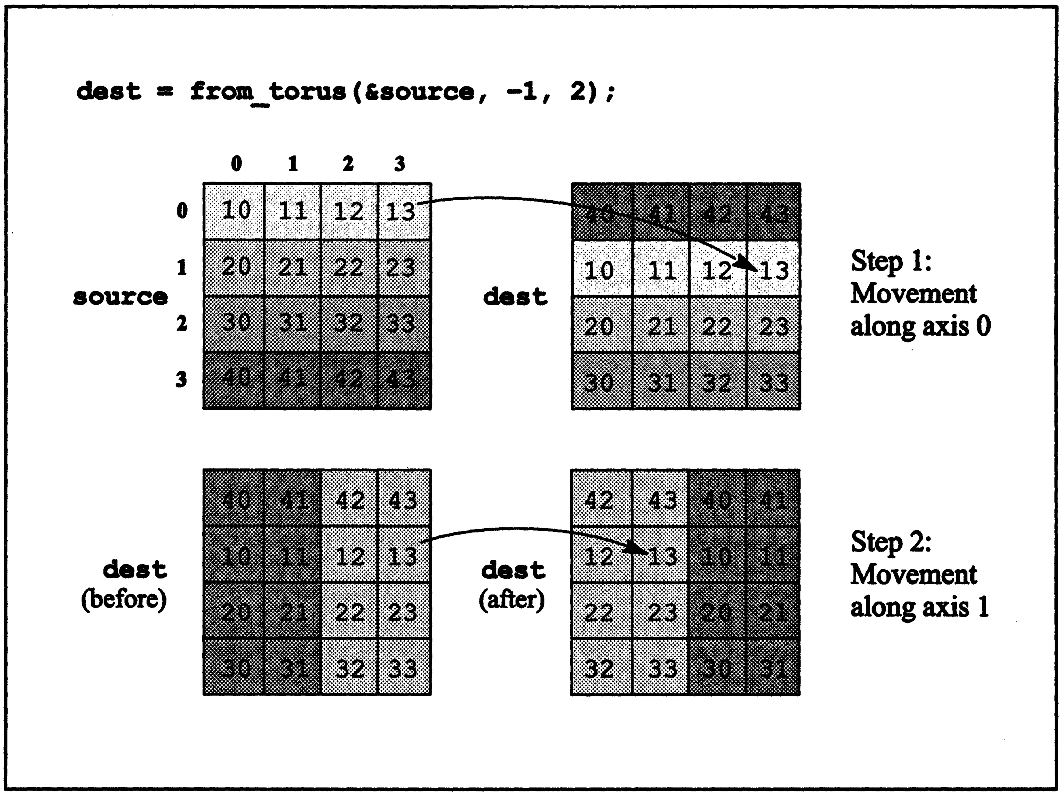

Figure 60. An example of the **from_torus** function

from_torus and **from_torus_dim** are both get operations, so their handling of inactive positions is the same as that of **from_grid** and **from_grid_dim**.

12.5.2 With Parallel Data of Any Length

The **from_torus** and **from_torus_dim** functions also have overloaded versions that can be used with parallel data of any length. Their definitions are:

```
void from_torus(
    void:current *destp,
    void:current *sourcep,
    int length,
    int distance_along_axis_0, ... );
```

```
void from_torus_dim (
    void:current *destp,
    void:current *sourcep,
    int length,
    int axis,
    int distance);
```

Note that these definitions are the same as those for **from_grid** and **from_grid_dim**, except that a **valuep** argument is not required, since values wrap when they go beyond the border of the grid.

12.6 The to_torus and to_torus_dim Functions

The **to_torus** and **to_torus_dim** functions are send operations that provide grid communication with wrapping to the other side of the grid. As with the other grid communication functions, the **_dim** version provides communication along one axis only, while the more general version provides communication along all axes. Both functions have overloaded versions for all arithmetic and aggregate types.

12.6.1 With Arithmetic Types

The **to_torus** and **to_torus_dim** functions have the following definitions when used with an arithmetic type:

```
void to_torus (
    type:current *destp,
    type:current source,
    int distance_along_axis_0, ... );

void to_torus_dim (
    type:current *destp,
    type:current source,
    int axis,
    int distance);
```

where:

destp is a scalar pointer to the parallel variable to which values are to be sent. This parallel variable can be of any arithmetic type; it must be of the current shape.

source is a parallel variable from which values are to be sent; it must be of the current shape and have the same arithmetic type as the parallel variable pointed to by **destp**.

distance_along_axis_0
(for **to_torus**) specifies how many positions along axis 0 the values of **source** are to travel. If the distance is 2, for example, **source** sends its value to the destination element whose position is two greater along axis 0. Include a distance argument for each dimension in the current shape. If the data is not to move along an axis, specify the distance for that axis as 0. The distance can be a negative number, which reverses the direction in which the data is to travel.

axis (for **to_torus_dim**) specifies the number of the axis along which the values of **source** are to be sent.

distance (for **to_torus_dim**) specifies how many positions along the axis the values of **source** are to be sent, as discussed in the description of **distance_along_axis_0**.

The behavior of inactive positions for **to_torus** and **to_torus_dim** is the same as it is for **to_grid** and **to_grid_dim**: elements of **source** at inactive positions cannot send values, but **source** can send values to elements at inactive positions.

Examples

The following code uses the **source** data also used in previous figures; it sends values of **source** to **dest** elements that are one position lower along axis 0:

```
to_torus_dim(&dest, source, 0, -1);
```

The results are shown in Figure 61. Compare these results to those for the comparable call to **from_torus_dim**, shown in Figure 59. The arrows in the figure show the movement of two elements of **source**: **[0][3]source** wraps around and sends its value to **[3][3]dest**; **[3][0]source** sends its value to **[2][0]dest**.

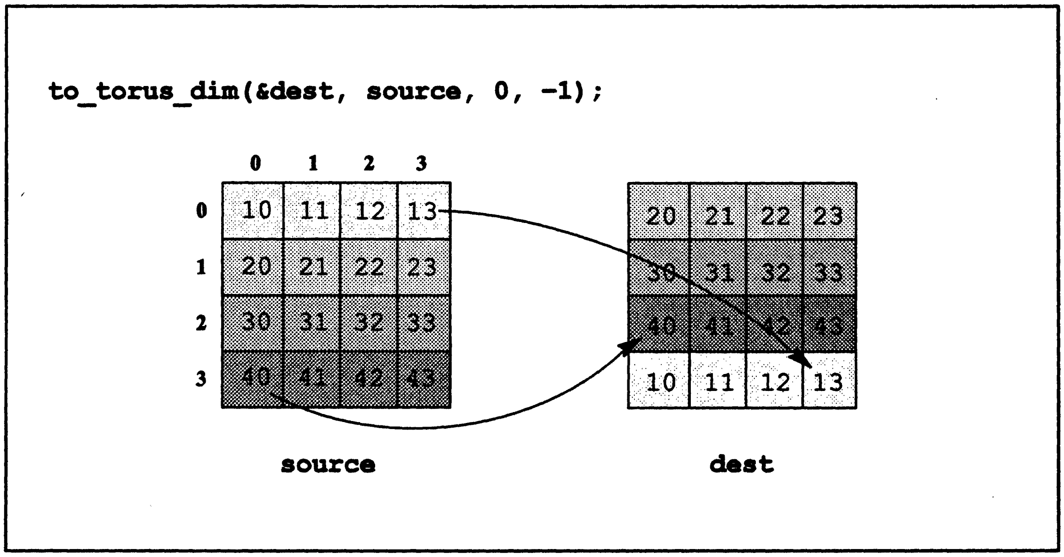

Figure 61. An example of the **to_torus_dim** function

to_torus is similar to **to_torus_dim**, except that you must specify the data movement for each axis, as you do for **from_torus** and **from_grid**. The following code uses the same **source** data used in previous figures:

```
to_torus(&dest, source, -1, 2);
```

The results are shown in Figure 62. Compare these results to those for the comparable call to **from_torus**, shown in Figure 60. The arrows in the figure show where **[0][3]source** ends up after the movement along axis 0 and axis 1.

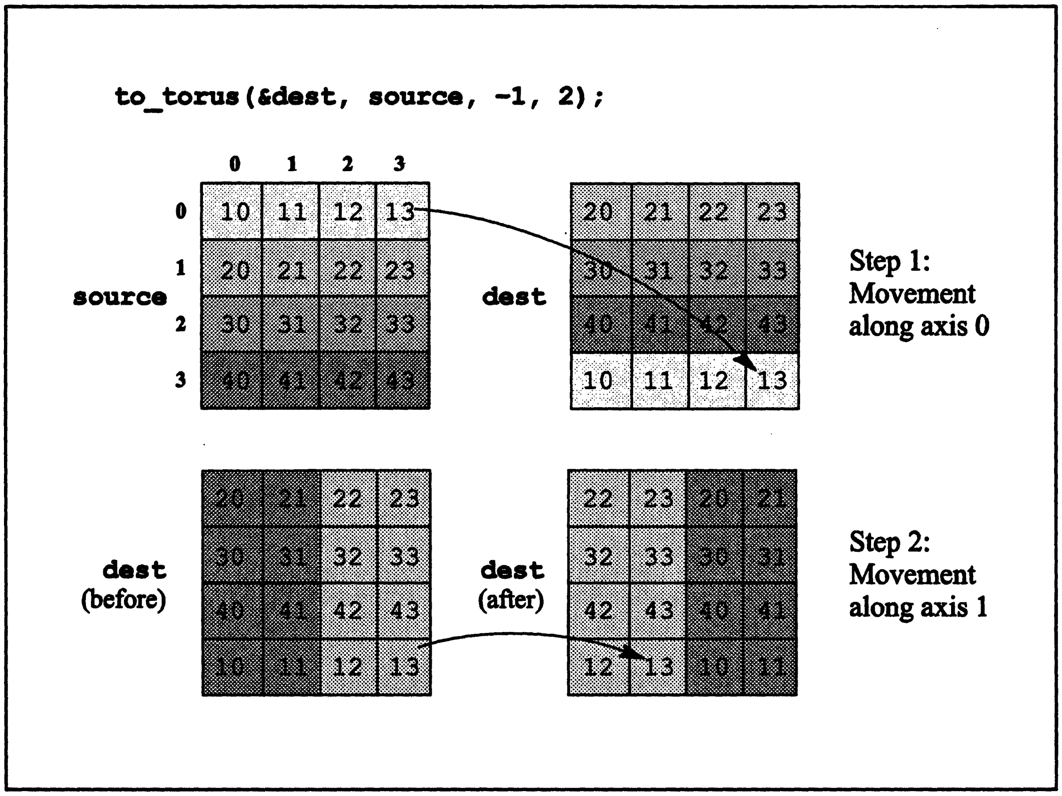

Figure 62. An example of the **to_torus** function

In the following example, we make a position inactive and call **to_torus_dim**:

```
where (source != 7)
  to_torus_dim(&dest, source, 0, 1);
```

Figure 63 shows the results for some sample data.

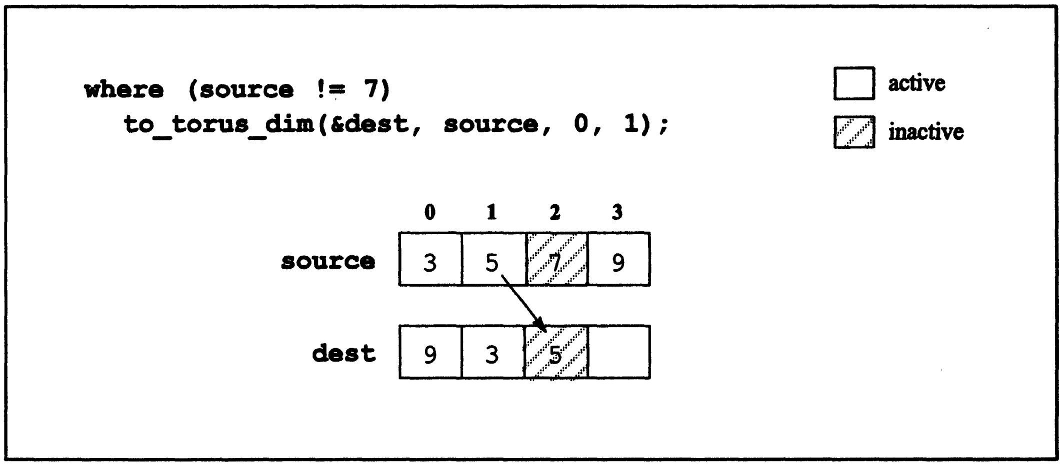

Figure 63. An example of **to_torus_dim** when a position is inactive

Note how the rules for send operations with inactive positions are applied in Figure 63:

- **[1]source** sends a value to **[2]dest**, even though position [2] is inactive.
- Position [2] is inactive, so **[2]source** doesn't send a value to **[3]dest**, which keeps its original value from before the call.

12.6.2 With Parallel Data of Any Length

The **to_torus** and **to_torus_dim** functions also have overloaded versions that can be used with parallel arrays or parallel structures. Their definitions are:

```
void to_torus(
    void:current *destp,
    void:current *sourcep,
    int length,
    int distance_along_axis_0, ... );

void to_torus_dim (
    void:current *destp,
    void:current *sourcep,
    int length,
    int axis,
    int distance);
```

Note that these definitions are the same as those for **from_torus** and **from_torus_dim**. But, as with the versions that use arithmetic types, the distance arguments are interpreted differently, and the behavior of inactive positions is different.

Chapter 13

Communication with Computation

This chapter discusses C* library functions that let you perform computations on parallel values that are being transmitted. Most of these functions use grid communication. The functions differ in the following ways:

- *The kinds of computation that are available for each function.* See Section 13.1.
- *The way in which parallel variable elements are selected.* For example, some functions let you divide the parallel variable elements into groups called *scan classes*. You can then operate on each scan class independently. See Section 13.2.
- *The way in which the function reports the results of the computation.* For example, **`scan`** provides a running total of its computations; **`spread`** provides only the final result.

Include the file **`<cscomm.h>`** when calling any of the functions discussed in this chapter.

13.1 What Kinds of Computation?

The **`scan`**, **`reduce`**, **`spread`**, **`multispread`**, and **`global`** functions let you specify a *combiner type* that indicates the kind of computation or combining you want carried out on the parallel data. Each of these functions is overloaded for some subset of the following combiner types:

Table 4. Combiner types

Combiner	Meaning
CMC_combiner_max	Take the largest value among the specified parallel variable elements.
CMC_combiner_min	Take the smallest value among the specified elements.
CMC_combiner_add	Add the values of the specified elements.
CMC_combiner_copy	Copy the values of the specified elements.
CMC_combiner_multiply	Multiply the values of the specified elements.
CMC_combiner_logior	Perform a bitwise logical inclusive OR on the specified elements.
CMC_combiner_logxor	Perform a bitwise logical exclusive OR on the specified elements.
CMC_combiner_logand	Perform a bitwise logical AND on the specified elements.

These combiner types are also used by the **send** function, which is described in the next chapter.

13.2 Choosing Elements

Several of the C* functions discussed in this chapter provide methods for choosing the subsets of parallel variable elements on which they are to operate. The terminology we use in referring to these subsets of elements comes from **scan**, which is the most general of the functions that use these methods.

13.2.1 The Scan Class

Two positions belong to the same *scan class* if their coordinates differ only along a specified axis. The following functions use the concept of a scan class: **scan**, **reduce**, **copy_reduce**, **spread**, **copy_spread**, **enumerate**, **rank**, and **multispread**.

To see how scan classes work, consider the 2-dimensional shape shown in Figure 64. (This and other shapes in this chapter are smaller than legal size in C*, so that they are easy to visualize.)

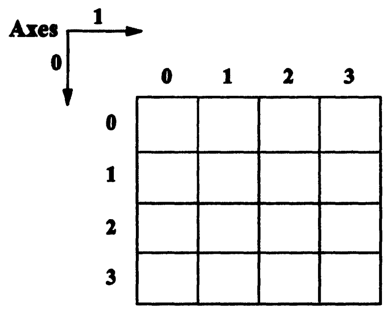

Figure 64. A 4-by-4 shape

If you specify axis 0 as an argument to one of the functions listed above, you get the scan classes shown in Figure 65. Positions [0][0], [1][0], [2][0], and [3][0] differ only in their coordinates for axis 0; therefore, they belong to the same scan class. Position [0][1] does not belong to this scan class, because it has a different axis 1 coordinate; it belongs to a scan class with positions [1][1], [2][1], and [3][1].

Thus, specifying axis 0 for this shape creates four separate scan classes, each of which is a column of positions through axis 0 in the shape. Functions like **scan** operate on each of these scan classes independently.

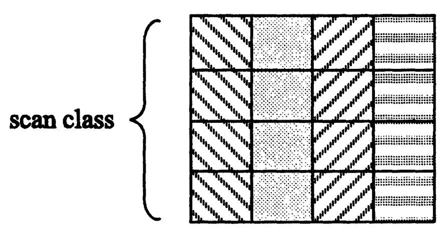

Figure 65. Scan classes for axis 0 of a 2-dimensional shape

Specifying axis 1, on the other hand, creates four different scan classes, each one consisting of a row of positions through axis 1 in the shape, as shown in Figure 66.

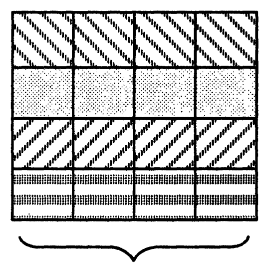

Figure 66. Scan classes for axis 1 of a 2-dimensional shape

If you have a 1-dimensional shape, there is, of course, only one axis you can specify, and only one scan class for the shape. You can, however, subdivide a scan class, as we discuss below.

If you have a 3-dimensional shape, specifying an axis once again gives you a set of scan classes consisting of the rows of positions that cross this axis. For example, in a 2-by-2-by-2 shape, specifying axis 0 creates the following four scan classes:

[0][0][0] and [1][0][0]

[0][1][0] and [1][1][0]

[0][0][1] and [1][0][1]

[0[[1][1] and [1][1][1]

To operate on more than one dimension in a multi-dimensional shape (for example, on planes of positions instead of rows of positions), you must use the **multispread** or **copy_multispread** function; these functions are discussed in Section 13.8.

The Scan Subclass

Only active positions participate in computations within a scan class. The active positions within a scan class are referred to as the *scan subclass.*

13.2.2 The Scan Set

There may be times when you want a function to operate independently on different parts of a scan subclass. The **scan**, **enumerate**, and **rank** functions let you do this by subdividing a scan subclass into *scan sets.*

To create scan sets, declare a `bool`-sized parallel variable of the shape on which the function is to operate, and initialize it to 0. This parallel variable is referred to as the *sbit*; it is used as the **sbit** argument to the functions listed above. Assign a 1 to an element of this parallel variable to mark the beginning of a scan set at that element's position. In the simplest case, the scan set for each position starts either at the beginning of the scan subclass, or at the nearest position below it in the scan subclass that has its sbit set to 1.

Figure 67 shows a 1-dimensional shape divided into scan sets. In the figure, the scan set for position 1, for example, consists of positions 0 and 1 (the scan subclass starts at position 0, so the scan set starts there also, even if the sbit for that position isn't set to 1). The scan set for position 7 consists of positions 5, 6, and 7, since **[5]sbit** is set to 1, thus starting a new scan set.

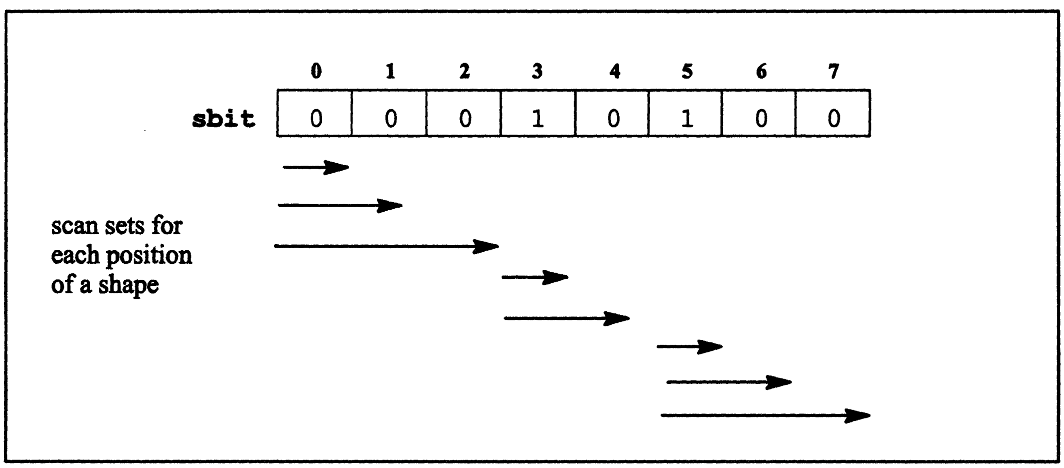

Figure 67. Scan sets in a 1-dimensional shape

Note than scan sets include only active positions; see Section 13.2.3, however, for a more in-depth discussion of inactive positions and scan sets.

To show how scan sets work, let's use an example in which we keep a running total of the values in the parallel variable **data** (this is a scan operation, as discussed in Section 13.3). The results are shown in Figure 68.

	0	1	2	3	4	5	6	7
sbit	0	0	0	1	0	1	0	0
data	0	1	2	3	4	5	6	7
running_total	0	1	3	3	7	5	11	18

Figure 68. An operation that provides a running total, using scan sets

In the example, `[1]`**running_total** contains the sum of `[0]`**data** and `[1]`**data**, since 0 and 1 are the positions in its scan set. `[3]`**running_total** contains only the value in `[3]`**data**, since `[3]`**sbit** is set to 1, thus starting a new scan set in this position.

You actually have more flexibility than this in how you can divide up scan subclasses:

- Whether an operation is *inclusive* or *exclusive* affects the way scan sets are interpreted; see "Inclusive and Exclusive Operations," below. The example in Figure 68 shows an inclusive operation.

- There are two ways of interpreting the sbit; see Section 13.2.3. In particular, this affects the way scan classes are divided when there are inactive positions, and when an operation proceeds in a downward direction. The example in Figure 68 shows an operation that proceeds in an upward direction.

Inclusive and Exclusive Operations

The way in which scan sets work when you are performing a particular operation depends on whether the operation is *inclusive* or *exclusive*. (NOTE: In this section, we are ignoring the effect of *segment bits* and *start bits*; these are discussed in the next section.)

In an *inclusive* operation (specified by **`CMC_inclusive`**), an element participates in the operation for its position—in other words, the scan set for a position contains that position. As we mentioned, Figure 68 shows the results of an inclusive operation.

In an *exclusive* operation (specified by **`CMC_exclusive`**), the scan set for an element does not contain the element itself—in other words, it does not participate in the operation for its position. Figure 69 shows the results of an exclusive operation, using the same data as that shown in Figure 68.

	0	1	2	3	4	5	6	7
sbit	0	0	0	1	0	1	0	0
data	0	1	2	3	4	5	6	7
running_total	0	0	1	0	3	0	5	11

Figure 69. An exclusive operation on scan sets

Note the difference between the two results. In the inclusive operation, for example, **`[2]running_total`** receives the running total for **`[0]data`**, **`[1]data`**, and **`[2]data`**; in the exclusive operation, **`[2]running_total`** receives the running total only for **`[0]data`** and **`[1]data`**. When there are no preceding elements in the scan set (for example, in **`[3]running_total`**), the element receives the identity for the operation.

13.2.3 Segment Bits and Start Bits

There are two different kinds of sbits: *segment bits* and *start bits*. Use the **smode** argument to the **scan**, **enumerate**, or **rank** function to specify which kind of sbit you want, as discussed below.

If smode Is CMC_segment_bit

If the value of the **smode** argument is **CMC_segment_bit**, the sbit is considered a *segment bit*, and it divides a scan subclass into segments, as follows:

- An sbit element set to 1 starts a new segment, whether or not the element appears in an active position.
- The way in which the segment bit divides the scan subclass is not affected by the direction of the operation.
- Operations in one segment never affect values of elements in another segment.

If smode Is CMC_start_bit

If the value of the **smode** argument is **CMC_start_bit**, the sbit is considered a *start bit*, and scan classes are divided as follows:

- An sbit element set to 1 divides a scan subclass *only if its position is active*.
- The division *is* affected by the direction of the operation. When the direction is downward, the division occurs from the higher coordinate to the lower coordinate.
- When an operation is *exclusive*, the position whose sbit element is set to 1 will receive a value from the preceding scan set.

These differences between segment bits and start bits are discussed below.

Inactive Positions

When the sbit is a segment bit, a new scan set is created, even though the position where it starts is inactive. Figure 70 shows an example (the scan sets displayed are for positions [2], [4], and [7]).

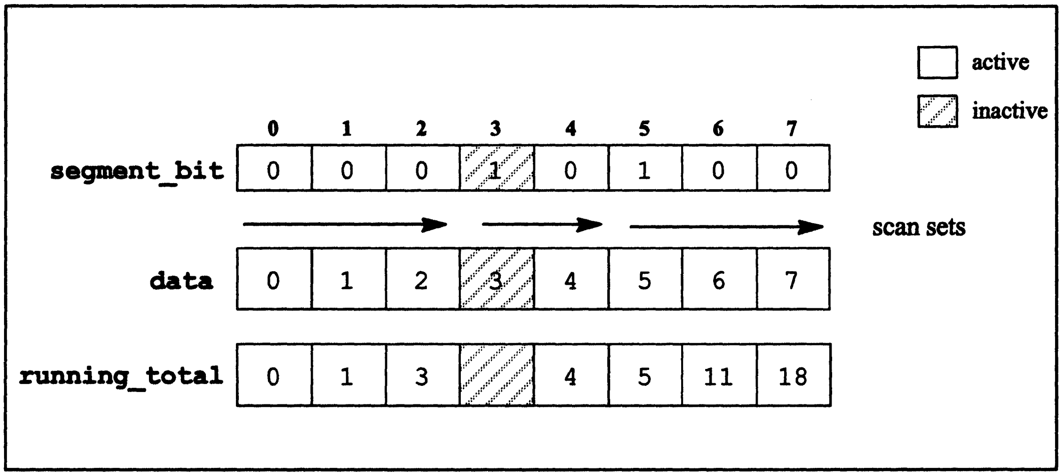

Figure 70. An inclusive operation in an upward direction
on segment-bit scan sets, with an inactive position

Note that position [3] does not participate in the operation, even though it starts a new scan set.

A start bit does not start a scan set if its position is inactive. Figure 71 is an example. Note that the scan set for position [4] begins at position [0], not at position [3], as in Figure 70.

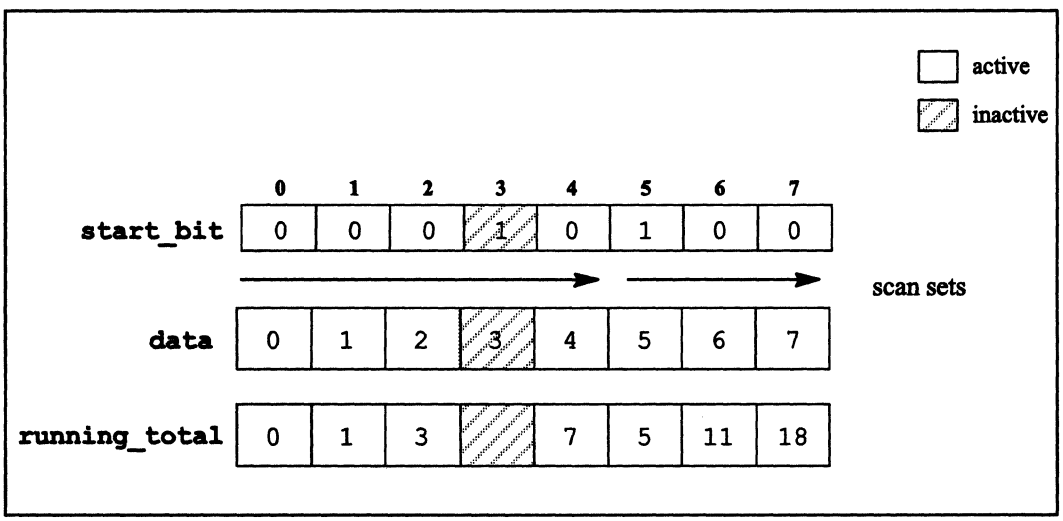

Figure 71. An inclusive operation in an upward direction
on start-bit scan sets, with an inactive position

The Direction of the Operation

When the direction of the operation is *upward*, it proceeds from lower-numbered positions to higher-numbered positions along the scan subclass. Both kinds of sbits divide the scan subclass in the same way when the direction is upward (provided that all positions are active); see Figure 67 for an example. You specify an upward direction with the argument **CMC_upward**.

When the direction of the operation is *downward* (specified by the argument **CMC_downward**), the operation proceeds from higher-numbered positions to lower-numbered positions along the scan subclass. In this case, segment bits divide the scan subclass in the same way as the sbits shown in Figure 67; however, since the operation proceeds in a downward direction, this means that a segment bit *ends* a scan set, and the operation begins again in the position with the next lowest coordinate. Figure 72 is an example; it shows the scan sets for positions [0], [3], and [5].

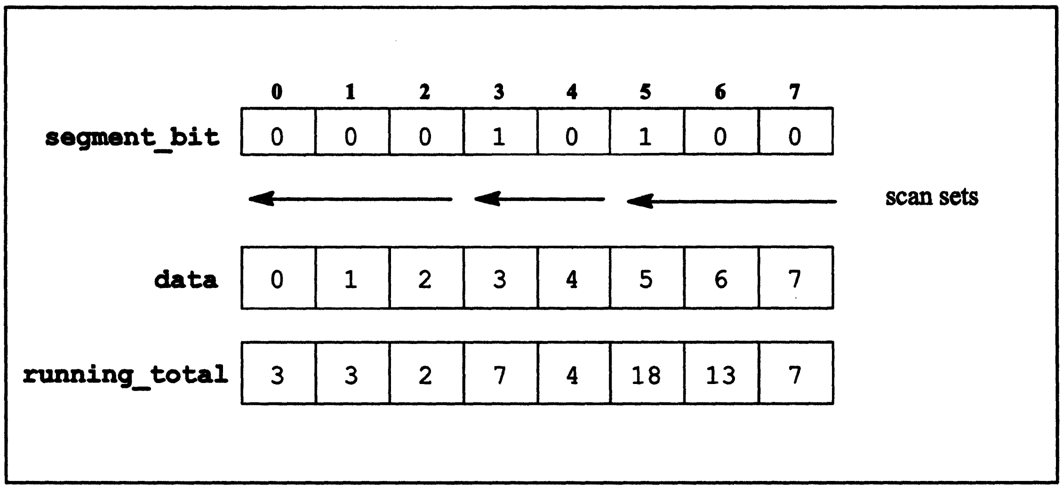

Figure 72. An inclusive operation in a downward direction on segment-bit scan sets

Start-bit scan sets, however, follow the downward direction; in other words, start bits start scan sets, rather than ending them. Figure 73 is an example; it shows the scan sets for positions [0], [4], and [6].

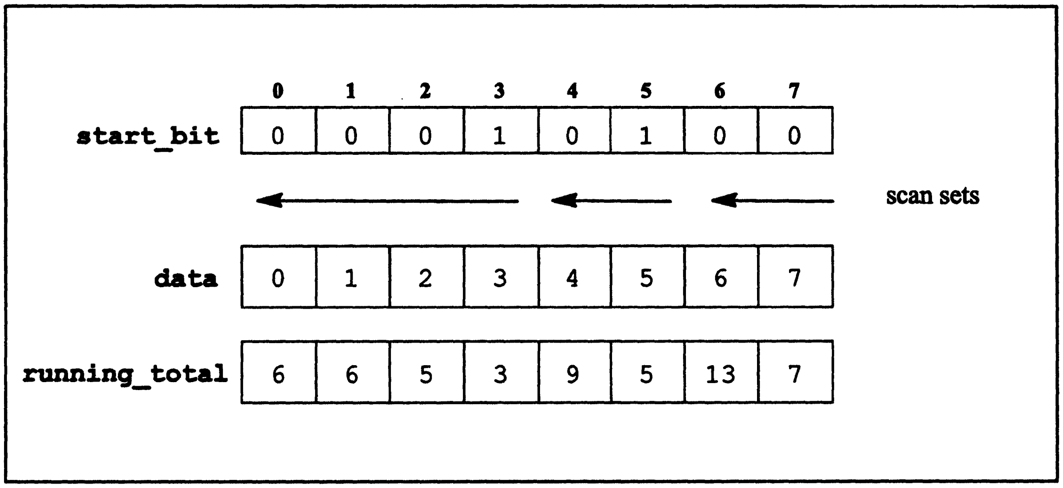

Figure 73. An inclusive operation in a downward direction
on start-bit scan sets

Data from Another Scan Set

In exclusive operations on start-bit scan sets, the first position in a scan set receives the result of the operation for the *preceding* scan set, if there is one. Figure 74 is an example.

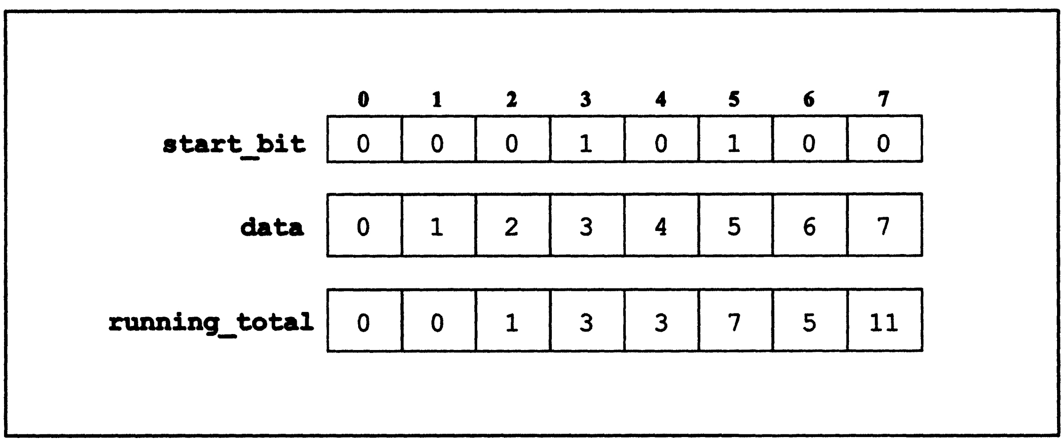

Figure 74. An exclusive operation in an upward direction
with start bits

Compare these results with those shown in Figure 69, which assumes that the sbit is a segment bit. **[3]running_total** and **[5]running_total** receive the results from the preceding scan set, rather than 0. **[0]running_total** still receives 0 (the identity for the operation) because there is no preceding scan set.

What constitutes a "preceding" scan set depends on the direction of the operation, of course. In a downward direction, scan sets with higher-numbered coordinates along the axis precede scan sets with lower-numbered coordinates.

13.3 The scan Function

Use the **scan** function to provide running results for operations on the scan sets you specify.

The definition of **scan** is as follows:

```
type:current scan (
    type:current source,
    int axis,
    CMC_combiner_t combiner,
    CMC_communication_direction_t direction,
    CMC_segment_mode_t smode,
    bool:current *sbitp,
    CMC_scan_inclusion_t inclusion);
```

where:

source is the parallel variable whose values are to be used in the operation. It must be of the current shape, and it can have any arithmetic type.

axis specifies the axis along which the scan class or classes are to be created; see Section 12.2.

combiner specifies the type of operation that **scan** is to carry out. Possible values are listed in Section 12.1.

direction specifies the direction of the operation. Possible values are **CMC_upward** and **CMC_downward**.

smode specifies whether the sbit is a segment bit or a start bit; see Section 13.2.3. Possible values are **CMC_start_bit**, **CMC_segment_bit**, and **CMC_none**. Specify **CMC_none** if there is no sbit.

sbitp is a scalar pointer to a **bool**-sized parallel variable of the current shape. This parallel variable is the sbit, which creates scan sets for the operation. Specify **CMC_no_field** if there is no sbit.

inclusion specifies whether the operation is exclusive or inclusive; see "Inclusive and Exclusive Operations," above. Possible values are **CMC_exclusive** and **CMC_inclusive**.

The function returns the result of the scan in a parallel variable of the current shape and with the same type as **source**.

The types **CMC_combiner_t**, **CMC_communication_direction_t**, **CMC_segment_mode_t**, and **CMC_scan_inclusion_t** are defined by the compiler.

The **scan** function provides a running result of the operation you specify on the parallel variable you specify. If you assign this result to a parallel variable of the current shape, each element of the parallel variable receives the running result for its position. The operation is carried out independently for each scan set.

13.3.1 Examples

The following example adds the values of **data** in an upward direction and assigns the running result to **running_total**; there is no sbit, and the operation is inclusive. The results are shown in Figure 75.

```
running_total = scan(data, 0, CMC_combiner_add,
    CMC_upward, CMC_none, CMC_no_field, CMC_inclusive);
```

```
running_total = scan(data, 0, CMC_combiner_add,
  CMC_upward, CMC_none, CMC_no_field, CMC_inclusive);
```

	0	1	2	3	4	5	6	7
data	4	7	9	5	3	5	9	6
running_total	4	11	20	25	28	33	42	48

Figure 75. An example of the **scan** function with no sbit

The following example assigns the minimum value of **data** in the scan set to **running_min**. The direction is downward, the operation is inclusive, and the sbit is a start bit. The results are shown in Figure 76.

```
running_min = scan(data, 0, CMC_combiner_min, CMC_downward,
  CMC_start_bit, &start_bit, CMC_inclusive);
```

```
running_min = scan(data, 0, CMC_combiner_min,
  CMC_downward, CMC_start_bit, &start_bit,
  CMC_inclusive);
```

	0	1	2	3	4	5	6	7
start_bit	0	0	0	0	1	0	0	0
data	4	7	9	5	3	5	9	6
running_min	3	3	3	3	3	5	6	6

Figure 76. An example of the **scan** function with a start bit and a downward direction

Note that you would get a different result in this example if the sbit were a segment bit, since segment bits and start bits behave differently when the direction is downward.

The following example multiplies the values of **data** in the scan set and assigns the product to **running_product**. The direction is upward, the operation is exclusive, and the sbit is a segment bit. The results are shown in Figure 77.

```
running_product = scan(data, 0, CMC_combiner_multiply,
  CMC_upward, CMC_segment_bit, &segment_bit, CMC_exclusive);
```

```
running_product = scan(data, 0,
  CMC_combiner_multiply, CMC_upward, CMC_segment_bit,
  &segment_bit, CMC_exclusive);
```

	0	1	2	3	4	5	6	7
segment_bit	0	0	0	0	1	0	0	0
data	4	7	9	5	3	5	9	6
running_product	1	4	28	252	1	3	15	135

Figure 77. An example of the **scan** function using a segment bit and an exclusive operation

These examples are of a 1-dimensional shape, which by definition has only one scan class. If a shape has more than one dimension, more than one scan class is created, and **scan** carries out the operation on all scan subclasses (or scan sets, if the sbit is used) at the same time.

The destination parallel variable can be the same as the source parallel variable. In other words, a statement like the following is legal:

```
data = scan(data, 0, CMC_combiner_add, CMC_upward, CMC_none,
  CMC_no_field, CMC_inclusive);
```

In this case, the elements of **data** are overwritten with the results of the operation.

13.4 The reduce and copy_reduce Functions

13.4.1 The reduce Function

Use the **reduce** function to put the result of an operation into a single parallel variable element in each scan subclass.

The **reduce** function has the following definition:

```
void reduce (
    type:current *destp
    type:current source,
    int axis,
    CMC_combiner_t combiner,
    int to_coord);
```

where:

destp is a scalar pointer to a parallel variable, of the current shape and of any arithmetic type. One element of each scan subclass of this parallel variable receives the result of the operation.

source is a parallel variable (of the current shape) whose values are to be used in the operation. It must be of the same type as the parallel variable pointed to by **destp**.

axis specifies the axis along which the scan class or classes are to be created; see Section 12.2.

combiner specifies the type of operation that **reduce** is to carry out. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**.

to_coord specifies the coordinate of the parallel variable pointed to by **destp** that is to receive the result of the operation.

Note the following differences between **reduce** and **scan**:

- **reduce** puts the final result of the operation into a single parallel variable element of the scan subclass; it does not produce a running result.

- **reduce** does not use scan sets; therefore, it does not have the arguments **smode** and **sbit**.
- Copying with reduction is handled as a separate function, which is discussed below.

Elements of **source** that are at inactive positions do not participate in the operation. If a position specified by **to_coord** is inactive, that element of **dest** does not receive the result.

dest can be the same parallel variable as **source**; the result simply overwrites the value(s) in the specified element(s).

An Example

The following statement puts the maximum value of **data** into element 0 of **max**. The results are shown in Figure 78.

```
reduce(&max, data, 0, CMC_combiner_max, 0);
```

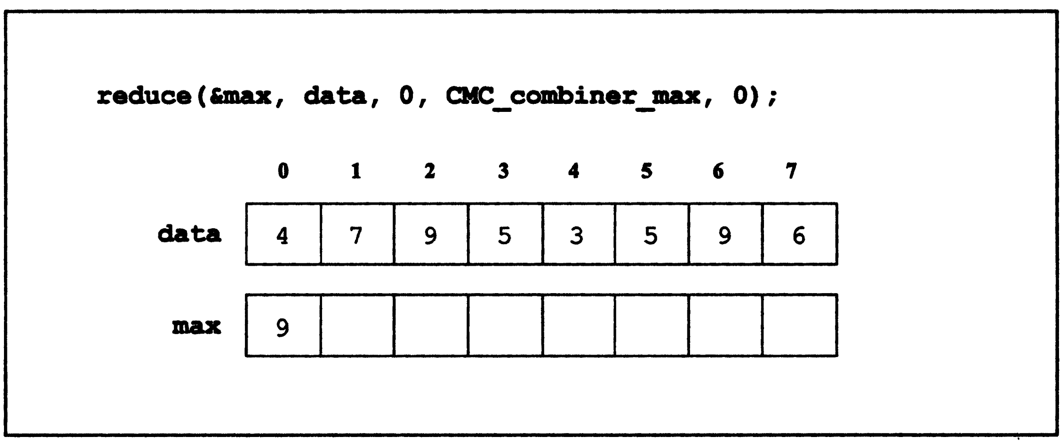

Figure 78. An example of the **reduce** function

Incidentally, this statement is virtually equivalent to the following C* statement:

```
[0]max = >?= data;
```

But note the following:

- If position [0] were inactive, the assignment statement above would work; if you used **reduce**, the reduction would not take place.
- The equivalence holds only for 1-dimensional shapes. In shapes with more dimensions, **reduce** carries out its operation separately for each scan subclass, whereas the reduction assignment carries out its operation once for all elements of the parallel variable.

13.4.2 The copy_reduce Function

Use the **copy_reduce** function to copy a value from one parallel variable element of a scan subclass to another parallel variable element.

The definition of **copy_reduce** is as follows:

```
void copy_reduce (
    type:current *destp
    type:current source,
    int axis,
    int to_coord,
    int from_coord);
```

The arguments are the same as for the **reduce** function, except that there is a **from_coord** argument instead of a combiner. **from_coord** specify the element of **source** from which the value is to be copied. It is copied into the **to_coord** element of the parallel variable pointed to by **destp** for each scan subclass. If either **from_coord** or **to_coord** specifies an inactive position, the copying does not take place for that scan subclass.

An Example

The following example copies the values of elements in row 1 of **data** into elements of row 0 of **copy**.

```
copy_reduce(&copy, data, 0, 0, 1);
```

The results for some sample values are shown in Figure 79.

```
copy_reduce(&copy, data, 0, 0, 1);
```

data

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23

copy

10	11	12	13

Figure 79. An example of the **copy_reduce** function

If the example of **copy_reduce** shown in Figure 79 were applied to a 1-dimensional shape, it would be equivalent to:

```
[0]copy = [1]data;
```

If position [0] were inactive, however, the results would be different. **[0]copy** would get the result from **[1]data** if you used the assignment statement above; it would not get the value if you used **copy_reduce**.

13.5 The spread and copy_spread Functions

13.5.1 The spread Function

Use the **spread** function to place the result of an operation into all the elements of a specified parallel variable in a scan subclass.

The **spread** function has the following definition:

```
type:current spread (
    type:current source,
    int axis,
    CMC_combiner_t combiner);
```

where:

source is a parallel variable (of the current shape) whose values are to be used in the operation. It can have any arithmetic type.

axis specifies the axis along which the scan class or classes are to be created; see Section 13.2.

combiner specifies the type of operation that **spread** is to carry out. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**. See Section 13.1.

spread returns its result in a parallel variable of the current shape; the parallel variable has the same type as **source**. This destination parallel variable can be the same as the source parallel variable, in which case the elements of the source parallel variable are overwritten with the result.

The **spread** function "spreads" the result of an operation into all active elements of the destination parallel variable in a scan subclass. Like **reduce**, **spread** does not use scan sets, and it does not have a **CMC_combiner_copy** operation; copying is handled by the **copy_spread** function, as discussed below.

Inactive positions do not participate in the operation.

An Example

The following code adds the values of the elements in **data** in the scan subclasses of axis 1, and assigns the result to **total**. The results for sample data are shown in Figure 80.

```
total = spread (data, 1, CMC_combiner_add);
```

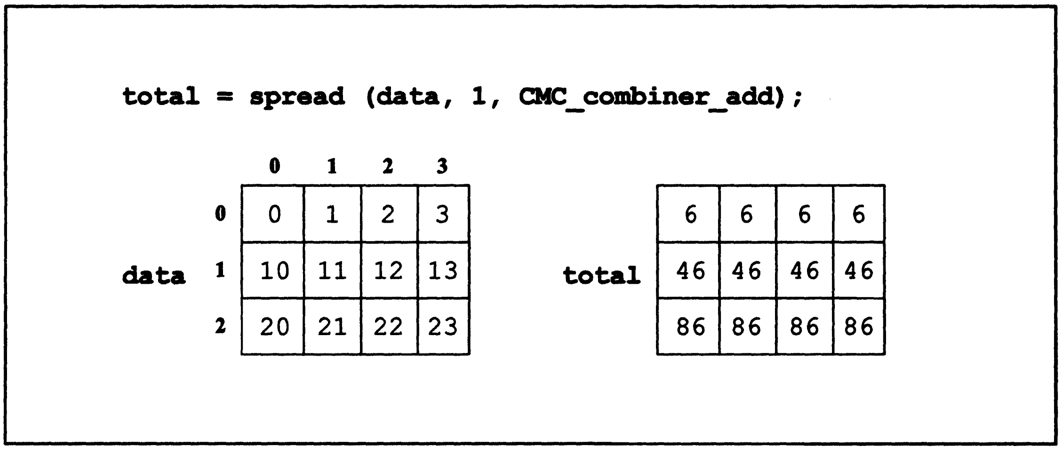

Figure 80. An example of the **spread** function

13.5.2 The copy_spread Function

Use the **copy_spread** function to copy a value from an element of a parallel variable in a scan subclass to all elements of a parallel variable in the scan subclass.

The **copy_spread** function has the following definition:

```
type:current copy_spread (
    type:current *sourcep,
    int axis,
    int coordinate);
```

where:

sourcep is a scalar pointer to a parallel variable, one value of which is to be copied.

axis specifies the axis along which the scan class or classes are to be created.

coordinate is the coordinate along **axis** that specifies the source parallel variable element whose value is to be copied.

The function returns a parallel variable of the current shape and the same arithmetic type as the parallel variable pointed to by **sourcep**, containing the results of the operation.

If a specified element of the source parallel variable is inactive, its value is copied. However, inactive positions of the destination parallel variable do not receive a result.

An Example

The following code copies the value from element [*n*][1] of **data** to elements of **copy** in the same scan subclass along axis 1. The results are shown in Figure 81.

```
copy = copy_spread(&data, 1, 1);
```

```
copy = copy_spread(&data, 1, 1);
```

data

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23

copy

1	1	1	1
11	11	11	11
21	21	21	21

Figure 81. An example of the **copy_spread** function

Note that, for a 1-dimensional shape, the above statement is equivalent to this statement:

```
copy = [1]data;
```

unless position [1] is inactive. In that case, the assignment statement works; **copy_spread**, however, would not copy **[1]data**.

13.6 The enumerate Function

Use the **enumerate** function to place in each active element of a parallel variable the size of its scan set. As we discuss in more detail below, **enumerate** is a generalized version of the **pcoord** function.

The **enumerate** function has the following definition:

```
unsigned int:current enumerate (
    int axis,
    CMC_communication_direction_t direction,
    CMC_scan_inclusion_t inclusion,
    CMC_segment_mode_t smode,
    bool:current *sbitp);
```

All the parameters for **enumerate** have the same meanings and take the same values as the corresponding parameters for the **scan** function; see Section 13.3. Like **scan**, **enumerate** lets you specify a direction, an sbit, and whether the operation is to be exclusive or inclusive. Note, however, that the return value is an **unsigned int** of the current shape.

If you specify **CMC_inclusive**, **enumerate** includes each position in calculating the size of the scan set for that position. If you specify **CMC_exclusive**, **enumerate** does not include the position in calculating the size of its scan set.

An inactive position does not receive a value and is not included in the calculation of values for other positions; see the third example, below.

13.6.1 Examples

The first example does an exclusive enumerate in an upward direction, ignoring the sbit, and assigning the result to **number**. The results are shown in Figure 82.

```
number = enumerate(0, CMC_upward, CMC_exclusive, CMC_none,
   CMC_no_field);
```

```
number = enumerate(0, CMC_upward, CMC_exclusive, CMC_none,
  CMC_no_field);
```

	0	1	2	3	4	5	6	7
number	0	1	2	3	4	5	6	7

Figure 82. An example of the **enumerate** function without an sbit

This is exactly equivalent to the following use of **pcoord**:

```
number = pcoord(0);
```

Both functions initialize each parallel variable element to its coordinate along the axis. The **enumerate** function, however, is more versatile than **pcoord**. In the next example, **enumerate** uses the sbit as a start bit and proceeds in a downward direction, using the inclusive mode:

```
number = enumerate(0, CMC_downward, CMC_inclusive,
  CMC_start_bit, &start_bit);
```

The results are shown in Figure 83.

```
number = enumerate(0, CMC_downward,
  CMC_inclusive, CMC_start_bit, &start_bit);
```

	0	1	2	3	4	5	6	7
start_bit	0	0	0	0	1	0	0	0
number	5	4	3	2	1	3	2	1

Figure 83. An example of the **enumerate** function with a start bit and a downward direction

In the following example, the sbit is a segment bit, the enumerate is exclusive, the direction is upward, and position 2 is inactive. The results are shown in Figure 84.

```
where (p1 != 9)
   number = enumerate(0, CMC_upward, CMC_exclusive,
    CMC_segment_bit, &segment_bit);
```

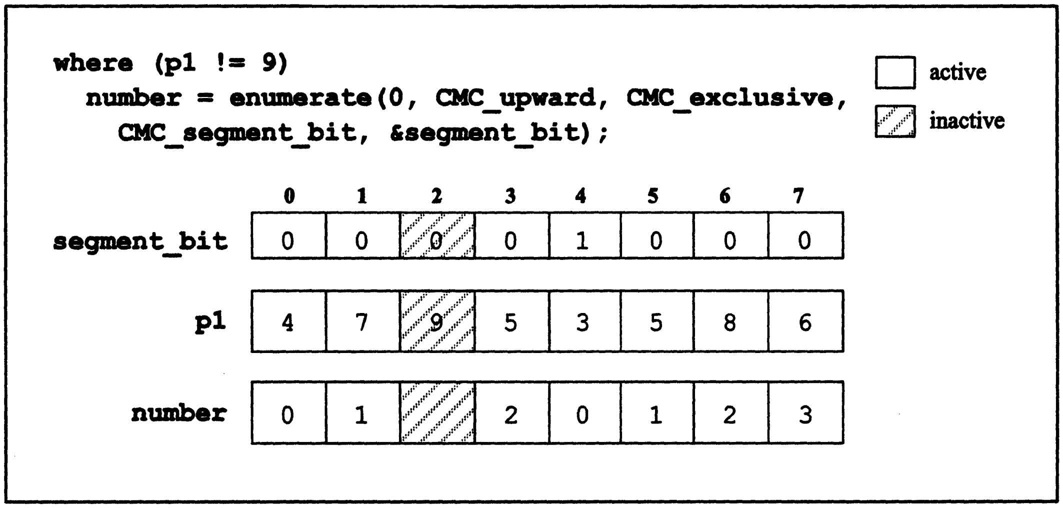

Figure 84. An example of the **enumerate** function using a segment bit and an exclusive operation, with an inactive position

Note that the inactive position is not included in the enumeration.

13.7 The rank Function

Use the **rank** function to produce a numerical ranking of the values of parallel variable elements in a scan set.

The definition of **rank** is as follows:

```
unsigned int:current rank (
    type:current source,
    int axis,
    CMC_communication_direction_t direction,
    CMC_segment_mode_t smode,
    bool:current *sbitp);
```

The parameters for **rank** have the same meanings and take the same values as the corresponding parameters for the **scan** function; see Section 13.3. Like **scan** and **enumerate**, **rank** lets you specify a direction and an sbit. It does not, however, let you specify that its operation is exclusive; the operation is inclusive by default. Like the **enumerate** function, **rank** returns an **unsigned int** of the current shape.

The **rank** function returns, for each active position, the rank of the value of the specified parallel variable at that position in its scan set. Inactive positions are not included in the determination of the rank for other positions, and they do not receive a rank themselves. The ranking is from 0 to $n-1$, where n is the total number of positions in the scan set. The ranks are assigned as follows:

- When the direction is *upward*, the lowest value is assigned rank 0.
- When the direction is *downward*, the highest value is assigned rank 0.
- If more than one element has the same value, their ranks are assigned arbitrarily within the range of ranks they represent.

13.7.1 Examples

The first example has no sbit and ranks the values of **data** in a upward direction; it assigns the ranks to **data_rank**. The results are shown in Figure 85.

```
data_rank = rank(data, 0, CMC_upward, CMC_none, CMC_no_field);
```

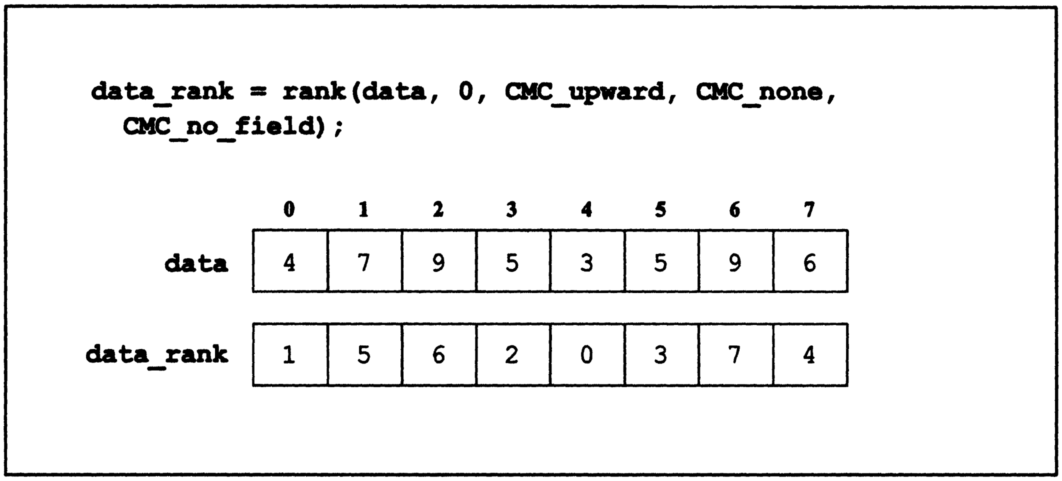

Figure 85. An example of the **rank** function with no sbit

In the next example, the sbit is a segment bit, the direction is downward, and position 1 is inactive. The results are shown in Figure 86.

```
where (data != 7)
    data_rank = rank(data, 0, CMC_downward, CMC_segment_bit,
       &segment_bit);
```

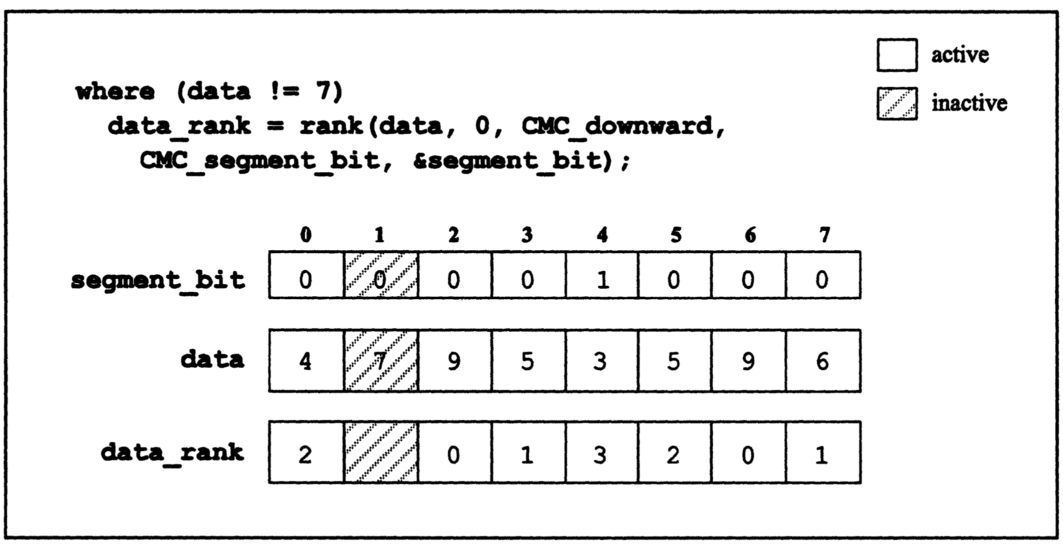

Figure 86. An example of the **rank** function using a segment bit
and a downward direction, with an inactive position

The final example uses **rank** along with parallel left indexing to actually reorder parallel variable elements according to their rank:

```
[rank(data, 0, CMC_upward, CMC_none, CMC_no_field)]sorted =
data;
```

In this example, **data** sends values to **sorted**, using the return values from **rank** as an index. The key here is to have **rank** operate on the parallel variable that is doing the sending. The results are shown in Figure 87.

```
[rank(data, 0, CMC_upward, CMC_none,
   CMC_no_field)]sorted = data;
```

	0	1	2	3	4	5	6	7
data	4	7	9	5	3	5	9	6
sorted	3	4	5	5	6	7	9	9

Figure 87. Using **rank** as a parallel left index to reorder parallel variable elements according to their ranks

Note how values move in the example: **[0]data**, for example, has a rank of 1; therefore, its value (4) is sent to **[1]sorted**.

You can also achieve the same result using the **make_send_address** and **send** functions along with **rank**; see Section 14.3.3.

13.8 The multispread Function

The **multispread** function is like the **spread** function, except that you can use it to spread the result of an operation along more than one axis at the same time. This is useful in shapes that have more than two dimensions. For example, in a 3-dimensional shape, you can use **spread** to spread results along any one of the dimensions; **multispread** lets you spread results through entire planes of positions instead of along a single dimension.

To see how this works, consider the simple 8-position 2-by-2-by-2 shape shown in Figure 88.

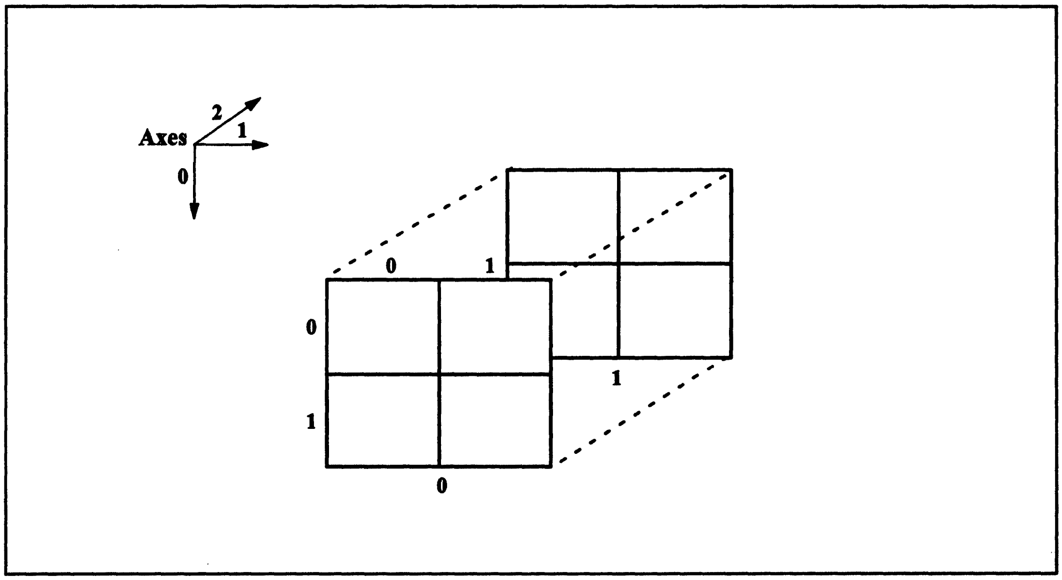

Figure 88. A 3-dimensional shape

As we mentioned in Section 12.2, specifying axis 0 creates four scan classes for this shape:

[0][0][0] and [1][0][0]

[0][1][0] and [1][1][0]

[0][0][1] and [1][0][1]

[0][1][1] and [1][1][1]

In each scan class, the positions differ only along axis 0. These scan classes are shown in Figure 89.

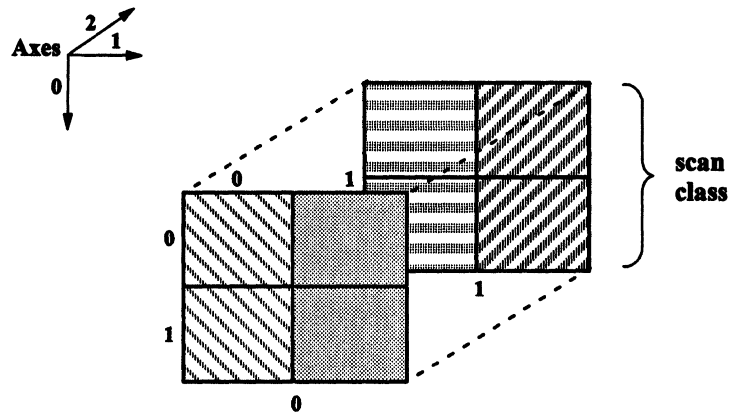

Figure 89. Scan classes in a 3-dimensional shape

For the **multispread** function, you can specify more than one axis along which the positions can differ. In this case, let the positions differ along axes 0 and 1; axis 2 is fixed. This results in two sets of positions:

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

and:

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

Figure 90 shows these two sets of positions. The sets of positions in which the positions are allowed to differ along more than one axis are called *hyperplanes*. Scan classes are therefore a subset of hyperplanes; in this subset, the positions can differ along only one axis. The **multispread** function operates on any kind of hyperplane.

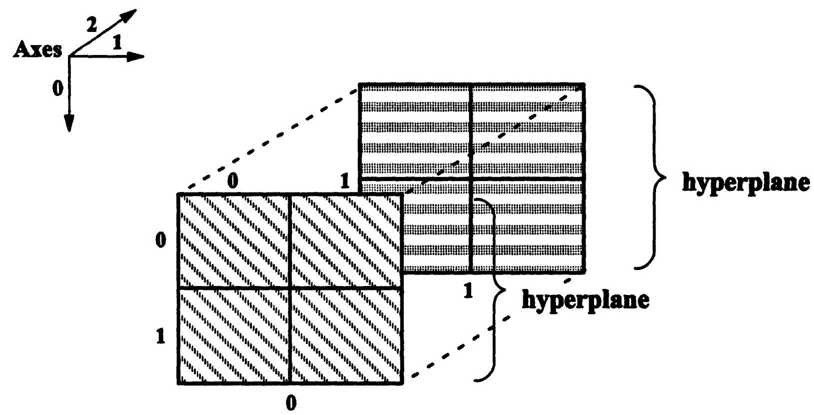

Figure 90. Hyperplanes in a 3-dimensional shape

The **multispread** function has the following definition:

```
type:current multispread (
    type:current source,
    int axis_mask,
    CMC_combiner_t combiner);
```

The only difference in this definition from that of **spread** is the **axis_mask** parameter. The **axis_mask** parameter is a bit mask that specifies the axes along which the positions in a hyperplane are allowed to differ. For example, use a bit mask of 3 to specify axes 0 and 1; use 6 to specify axes 1 and 2.

The following example assumes a 3-dimensional shape like the one shown above. In it, the values of **source** in the hyperplanes described by axes 0 and 1 are added, and the results are spread to all elements of **dest** in the same hyperplane.

```
dest = multispread(source, 3, CMC_combiner_add);
```

13.8.1 The copy_multispread Function

There is also a **copy_multispread** function, comparable to the **copy_spread** function, but available for use on hyperplanes instead of scan classes. Using **copy_multispread**, however, requires an understanding of send addresses, which are discussed in the next chapter. We therefore defer discussion of this function until Section 14.5.

13.9 The global Function

Use the **global** function to perform reduction operations on a parallel variable and assign the result to a variable on the front end.

The **global** function has the following definition:

```
type global (
    type:current source,
    CMC_combiner_t combiner);
```

where:

source is a parallel variable (of the current shape and any arithmetic type) upon whose values the reduction operation is to be performed.

combiner specifies the reduction operation. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, and **CMC_combiner_logand**; see Section 13.1 for definitions of these values.

The function returns a scalar variable of the same type as **source**.

The **global** function provides an alternative method for performing certain reduction operations. For example, the following two statements are equivalent (where **s1** is a scalar variable and **p1** is a parallel variable of the same type):

```
s1 = |= p1;
```

and:

```
s1 = global(p1, CMC_combiner_logior);
```

Both do a bitwise inclusive OR of **p1** and assign the result to **s1**.

Note that **global** does not have a **combiner** value for the reduction assignment operator **-=** (negative of the sum of the parallel values).

The **global** function operates only on active positions.

Chapter 14

General Communication

The C* communications functions we have discussed so far have required that the source and destination parallel variables be of the current shape (except for **global**, where the destination is a scalar variable), and that the communication be in regular patterns—that is, all elements transfer their values the same number of positions in the same direction. In this chapter, we introduce functions that allow communication in which:

- One of the parallel variables need not be of the current shape, and
- The communication need not be in a regular pattern.

The **get** and **send** functions described in this chapter provide communication comparable to that offered by parallel left indexing; see Chapter 10.

The **read_from_position** function described in this chapter provide communication comparable to that offered by assigning a scalar-indexed parallel variable to a front-end variable; **write_to_position** is comparable to assigning a front-end variable to a scalar-indexed parallel variable. The **read_from_pvar** function reads data from a parallel variable into a front-end array; **write_to_pvar** writes data from a front-end array to a parallel variable.

Include the header file **<cscomm.h>** when calling any of the functions discussed in this chapter.

14.1 The make_send_address Function

Grid communication requires knowing the coordinates of parallel variable elements in the shape. More information is required for general communication. Specifically, you need to supply a *send address* for a parallel variable element's position. This send address, along with a position's shape, uniquely identifies a position among all positions in all shapes; thus, you can use this address when an element of the current shape is communicating with an element that is of a different shape.

Use the **make_send_address** function to obtain a send address for one or more positions. **make_send_address** is an overloaded function that has different versions depending on the following:

- *Whether you want to return a single address or multiple addresses.* Multiple addresses are returned as a parallel variable of the current shape.
- *Whether you specify axis coordinates for the position in a varargs list or in an array.* The choice is the same as that for the **allocate_shape** function, which we discussed in Chapter 9. If you know the rank of the position's shape, it is easier to use the varargs version. If the rank will not be known until run time, you must use an array.

14.1.1 Obtaining a Single Send Address

To obtain a send address for a single position, use **make_send_address** with one of the following formats:

```
CMC_sendaddr_t make_send_address (
    shape s,
    int axis_0_coord, ...);
```

or:

```
CMC_sendaddr_t make_send_address (
    shape s,
    int axes[]);
```

where:

s is the shape to which the position whose address you are obtaining belongs.

axis_0_coord
(in the first version) specifies the position's coordinate along axis 0. Specify as many coordinates as there are axes in the shape.

axes[ ] (in the second version) is an array that contains the position's coordinates.

The function returns a scalar value (of type **CMC_sendaddr_t**) that is the send address of the position. This address is returned even if the position is inactive.

Note that the shape you specify in the parameter list need not be the current shape.

An Example

The following code calculates the send address of position [77][44] in shape **image** and assigns this address to the variable **addr** on the front end:

```
CMC_sendaddr_t addr;
addr = make_send_address(image, 77, 44);
```

14.1.2 Obtaining Multiple Send Addresses

To obtain send addresses for more than one position, use **make_send_address** with one of the following formats:

```
CMC_sendaddr_t:current make_send_address(
    shape s,
    int:current axis_0_coord, ...);
```

or:

```
CMC_sendaddr_t:current make_send_address (
    shape s,
    int:current axes[]);
```

These formats are the same as the ones shown in Section 14.1.1, except that the **axis_*n*_coord** arguments take parallel `int`s of the current shape, and the function returns a parallel variable of the current shape.

The value in each element of the parallel variable you specify for an axis of shape **s** represents a coordinate along that axis. The corresponding elements of the parallel variables that represent all the axes of the shape therefore fully specify a position in shape **s**. The function returns the send address for each position specified in this way. These send addresses are returned as the values of elements of a parallel variable that is of the current shape.

For example, if you specify **p1** as the **axis** argument for a 1-dimensional shape **s**, and **[0]p1** contains the value 4, then the send address of position [4] of shape **s** is returned in element [0] of a parallel variable of the current shape.

You cannot mix scalar values and parallel values in the argument list. If you want to use a scalar value (for example, because you only want the send addresses of positions whose coordinate for axis 1 is 3), do one of the following:

- Use a separate assignment statement to assign 3 to a parallel variable; or
- Use a cast in the argument list to explicitly promote 3 to a parallel value.

When Positions Are Inactive

If a position in the current shape is inactive, that position does not participate in the operation. In other words, the function does not return the send address specified by that position's parallel variable elements.

If elements specify a position in shape **s** that is inactive, the send address for that position *is* returned.

An Example

Figure 91 shows an example of **make_send_address**, using parallel variables of the 1-dimensional shape **t** to map parallel variables of the 2-dimensional shape **s**.

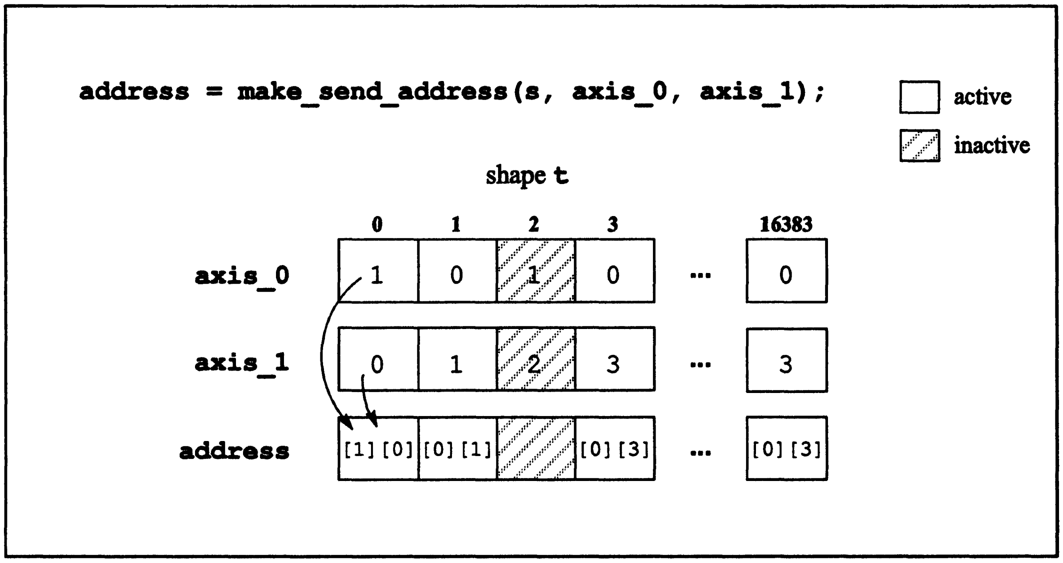

Figure 91. An example of the **make_send_address** function

Note the following in Figure 91:

- Two elements contain the same send address; this is legal.
- Position [2] is inactive; therefore, element [2] of **address** does not obtain the send address specified by the values in **[2]axis_0** and **[2]axis_1**.

The values of the elements that specify coordinates for an axis must be within the range of these coordinates. If, for example, shape **s** has 256 positions along axis 0, a value of 256 or greater in an element of **axis_0** would produce a run-time error, depending on the safety level.

14.2 Getting Parallel Data: The get Function

Use the **get** function to get values from a parallel variable when grid communication is not possible—that is, when communicating between shapes, or when the communication is not in a regular pattern. The **get** function is overloaded for both arithmetic and aggregate types.

14.2.1 Getting Parallel Variables

The **get** function has the following definition when used with arithmetic types:

```
type:current get (
    CMC_sendaddr_t:current send_address,
    type:void *sourcep,
    CMC_collision_mode_t collision_mode);
```

where:

send_address
: is a parallel variable of the current shape. The parallel variable contains send addresses for positions in a shape that need not be the current shape; see Section 14.1. They must, however, be of the same shape as the parallel variable pointed to by **sourcep**.

sourcep
: is a scalar pointer to a parallel variable (of any shape) from which values are to be returned. The parallel variable pointed to by **send_address** specifies which values are to be returned and where they are to be assigned.

collision_mode
: specifies what to do if more than one destination parallel variable element tries to get from the same element of the source parallel variable. Possible values are **CMC_collisions**, **CMC_no_collisions**, **CMC_few_collisions**, and **CMC_many_collisions**. See "Collisions in Get Operations," below.

The **get** function returns a parallel variable of the current shape. It has the same arithmetic type as the parallel variable pointed to by **sourcep**, and it contains the values of the parallel variable pointed to by **sourcep** in the positions specified by **send_address**.

The **get** function works like a get operation using a parallel left index; see Chapter 10. A destination parallel variable obtains values of the source parallel variable, using the parallel variable **send_address** as an index. Thus, given the following:

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
int:ShapeA axis_0, axis_1, dest;
int:ShapeB source;
```

The following two code fragments have the same results:

```
with (ShapeA) {
   CMC_sendaddr_t:ShapeA address;
   address = make_send_address(ShapeB, axis_0, axis_1);
   dest = get(address, &source, CMC_collisions);
}
```

and:

```
with (ShapeA)
   dest = [axis_0][axis_1]source;
```

The **get** function is more general, however:

- You can use **get** even if the rank of the shape from which you want to get values is not known until run time. Parallel left indexing requires that you know the rank of the shape when you write the program.
- The **get** function lets you control how collisions are handled; see below.
- The **get** function also lets you get parallel arrays. See Section 14.2.2, below.

If there are inactive positions in **ShapeA** in the first example above, elements of **dest** at these positions do not get values from **source**. The status of the positions in **ShapeB** does not matter; the active elements of **dest** get the values from the positions for which **address** has send addresses, whether or not these positions are active. Once again, this behavior is the same as that for get operations with parallel left indexing.

Collisions in Get Operations

The collisions we have talked about in previous chapters occur when two elements try to send to the same element at the same time. Get operations also have collisions, however; these occur when more than one parallel variable element tries to get a value from the same element at the same time. Unlike send collisions, get collisions are permitted in C*; they are handled automatically by get operations in the language. The **get** function and its **collision_mode** argument, however, gives you some control over how collisions are handled.

We recommend using the **CMC_collisions** option of **collision_mode** for most applications. This is the method used by get operations in the language itself. The other options may be useful in special circumstances:

- If there is no possibility of collisions, you can specify **CMC_no_collisions**; currently, this option uses the same code as **CMC_collisions**. However, future implementations of the **get** function may increase the performance of **CMC_no_collisions**.

- **CMC_many_collisions** and **CMC_few_collisions** can be useful if your application is memory-intensive and risks running out of storage (you can tell this if, for example, your program doesn't run with 4K physical processors, but does run with 8K processors). **CMC_collisions** requires memory for two aspects of its operation: to store the paths it takes in doing gets for each position, and to store colliding addresses. If it runs out of memory, it switches over and tries the algorithm used by **CMC_many_collisions**, which is slower but requires less memory. Under these circumstances, the operation would be faster if you specified **CMC_many_collisions** to begin with, thus avoiding the time spent trying the **CMC_collisions** algorithm.

  If **CMC_collisions** takes a long time due to memory limitations and the get has few collisions, **CMC_few_collisions** may be faster. In this case, the get operation iterates separately over each collision, saving the memory required to store the colliding addresses.

14.2.2 Getting Parallel Data of Any Length

You can also use the **get** function to obtain values from parallel locations of any length—typically, parallel structures or parallel arrays.

This version of the **get** function has the following definition:

```
void  get (
    void:current *destp,
    CMC_sendaddr_t:current *send_addressp,
    void:void *sourcep,
    CMC_collision_mode_t collision_mode,
    int length);
```

where:

destp is a scalar pointer to a parallel location of the current shape. This location obtains values from **sourcep**, based on the index in the parallel variable pointed to by **send_addressp**.

send_addressp
is a scalar pointer to a parallel variable of the current shape. The parallel variable contains send addresses for positions in a shape that need not be the current shape. See Section 14.1.

sourcep is a scalar pointer to a parallel location; it need not be of the current shape. The parallel variable pointed to by **send_addressp** specifies positions of this location. Data is to be gotten from these positions.

collision_mode
specifies what to do if more than one destination parallel variable element tries to get from the same element of the source parallel variable. Possible values are **CMC_collisions**, **CMC_no_collisions**, **CMC_few_collisions**, and **CMC_many_collisions**. See "Collisions in Get Operations," above.

length specifies the length in bits of the parallel location pointed to by **sourcep**.

This version of the **get** function lets you obtain data that is larger than the standard data types; typically, this data would be in a parallel structure or parallel array. For example:

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
struct S {
   int a;
   int b;
};
int:ShapeA axis_0, axis_1;
struct S:ShapeA dest_struct;
struct S:ShapeB source_struct;

main()
{
   with (ShapeA) {
      CMC_sendaddr_t:ShapeA address;
      address = make_send_address(ShapeB, axis_0, axis_1);
      get(&dest_struct, &address, &source_struct,
```

```
            CMC_collisions, boolsizeof(source_struct));
    }
}
```

dest_struct, of shape **ShapeA**, gets data from individual positions of the structure **source_struct**, of shape **ShapeB**, based on the send addresses stored in **address**. Note the use of the intrinsic function **boolsizeof** to obtain the length, in bits, of **source_struct**.

14.3 Sending Parallel Data: The send Function

Use the **send** function to send parallel data when grid communication is not possible—that is, when communicating between shapes, or when the communication is not in a regular pattern. The **send** function is overloaded for both arithmetic and aggregate types.

14.3.1 Sending Parallel Variables

The **send** function has the following definition when used with arithmetic types:

```
type:current send (
    type:void *destp,
    CMC_sendaddr_t:current send_address,
    type:current source,
    CMC_combiner_t combiner,
    bool:void *notifyp);
```

where:

destp is a scalar pointer to a parallel variable to which values are to be sent. It can be of any arithmetic type and any shape.

send_address
is a parallel variable of the current shape. The parallel variable contains send addresses for positions in the shape of the parallel variable pointed to by **destp**. This shape need not be the current shape; see Section 14.1.

source is a scalar pointer to a parallel variable from which values are to be sent. It must be of the current shape, and it must have the same type as the parallel variable pointed to by **destp**.

combiner specifies how **send** is to handle collisions. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, **CMC_combiner_logand**, and **CMC_combiner_overwrite**. All of these are defined in Section 13.1 except **CMC_combiner_overwrite**. If you specify **CMC_combiner_overwrite** and more than one value is sent to a parallel variable element, one of the values is chosen arbitrarily and stored in the element, and the rest of the values are discarded.

notifyp is a scalar pointer to a **bool**-sized parallel variable of the same shape as the parallel variable pointed to by **destp**. Initialize it to 0 before using it. When an element of the **destp** parallel variable receives a value, the corresponding element of the parallel variable pointed to by **notifyp** is set to 1. If you do not want to use a notify bit, specify **CMC_no_field** for this argument.

send returns the source.

Using the **send** function is roughly equivalent to performing a send operation with parallel left indexing; see Chapter 10. The **source** parallel variable sends values to the **destp** parallel variable, using **send_address** as an index. The combiners are equivalent to reduction assignment operators. **CMC_combiner_overwrite** has the same effect as the = operator, when the parallel right-hand side is cast to the type of the scalar left-hand side.

There are some differences, however, between the **send** function and send operations with parallel left indexing:

- The **send** function can be used when the rank of the shape of the destination parallel variable is not known until run time.
- The **send** function lets you include a notify bit, which provides notification that a value has been received by an element of the destination parallel variable.
- There is not a complete correspondence between the combiners and the reduction assignment operators. For example, there is no combiner that is equivalent to the -= reduction assignment operator.
- The **send** function has an overloaded version that lets you send parallel arrays; see Section 14.3.2, below.

Inactive Positions

Inactive positions are treated in the same way they are treated by send operations with parallel left indexes:

- An element in an inactive position in the current shape does not send a value.
- Destination parallel variable elements receive values even if they are in inactive positions.

In addition, the notify bit can be set even in an inactive position.

An Example

The following code sends values from elements of **source** to elements of **dest**.

```
#include <cscomm.h>

shape [16384]ShapeA;
shape [2][16384]ShapeB;
int:ShapeA axis_0, axis_1, source;
int:ShapeB dest;
bool:ShapeB notify_bit = 0;

/* Code to initialize parallel variables omitted. */

main()
{
   with (ShapeA) {
      CMC_sendaddr_t:ShapeA address;
      address = make_send_address(ShapeB, axis_0, axis_1);

      where (source < 9)
         send(&dest, address, source, CMC_combiner_min,
               &notify_bit);
   }
}
```

Some sample results are shown in Figure 92. The arrows show what happens to the value at **[3]source**, based on the send address in **[3]address**.

Note the following points in these results:

- Position [2] of **ShapeA** is inactive; therefore, **[2]source** does not send its value.
- The **CMC_combiner_min** combiner causes the 3 from **[0]source**, rather than the 5 from **[1]source**, to be sent to **[1][0]dest**.
- The notify bit is set in the two positions that receive values.

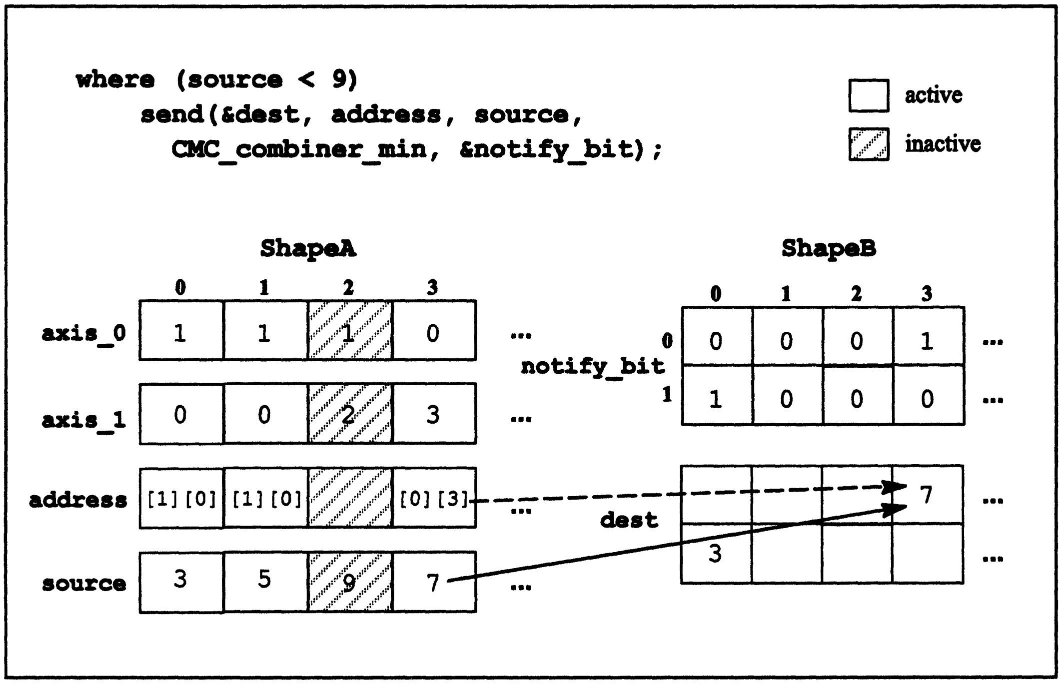

Figure 92. An example of the **send** function

14.3.2 Sending Parallel Data of Any Length

You can also use the **send** function to send parallel data of any length—typically a parallel structure or parallel array.

This version of the **send** function has the following definition:

```
void:current * send (
    void:void *destp,
    CMC_sendaddr_t:current *send_addressp,
    void:current *sourcep,
    int length,
    bool:void *notifyp);
```

where:

destp is a scalar pointer to a parallel location to which data is to be sent. **void:void** specifies that **destp** points to a location that can be of any type and of any shape.

send_addressp
is a scalar pointer to a parallel variable of the current shape. The parallel variable contains send addresses for positions in the shape of the parallel variable pointed to by **destp**.

sourcep is a scalar pointer to a parallel location from which data is to be sent. It must be of the current shape.

length specifies the length in bits of the location whose beginning is pointed to by **sourcep**.

notifyp is a scalar pointer to a **bool**-sized parallel variable of the same shape as the location pointed to by **destp**. When data is written to a position pointed to by **destp**, the corresponding element of the parallel variable pointed to by **notifyp** is set to 1. If you do not want to use a notify bit, specify **CMC_no_field** for this argument.

send returns a pointer to the source.

This version of the **send** function lets you send data that is larger than the standard data types; typically, this data would be in a parallel structure or parallel array. The data is sent from the source location to the destination location, using the parallel variable pointed to by **send_addressp** as an index to determine the destination.

Note that this version of **send** does not include a **combiner** argument. This version uses the **CMC_combiner_overwrite** option, and arbitrarily chooses a position of the array or structure if there would otherwise be a collision.

For example:

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
struct S {
   int a;
   int b;
};
int:ShapeA axis_0, axis_1;
bool:ShapeB notify_bit = 0;
struct S source_struct:ShapeA, dest_struct:Shape_B;

main()
{
   with (ShapeA) {
      CMC_sendaddr_t:ShapeA address;
      address = make_send_address(ShapeB, axis_0, axis_1);
      send(&dest_struct, &address, &source_struct,
        boolsizeof(source_struct), &notify_bit);
   }
}
```

The values of individual positions of the parallel structure **source_struct**, of shape **ShapeA**, are sent to **dest_struct**, of shape **ShapeB**, based on the send addresses stored in **address**. Note the use of the intrinsic function **boolsizeof** to obtain the length, in bits, of **source_struct**.

14.3.3 Sorting Elements by Their Ranks

You can use **send**, along with the **make_send_address** and **rank** functions, to reorder elements of a parallel variable by the ranks of their values. Note that this is also possible with parallel left indexing, as described in Section 13.7.1.

In the following example, we rearrange salary data for employees:

```
#include <cscomm.h>

shape [16384]employees;
struct employee {
   int id;
   int salary;
```

```
   };
struct employee:employees staff;

main()
{

/* Code to initialize salaries and ids omitted. */

   with (employees) {
      int:employees order;
      CMC_sendaddr_t:employees address;

      /* Determine ranks of salary values. */

      order = rank(staff.salary, 0, CMC_upward, CMC_none,
                   CMC_no_field);

      /* Create send addresses, using salary ranks as
         the index. */

      address = make_send_address(employees, order);

      /* Send employee data for each employee to new
         positions, based on the salary ranks. */

      send(&staff, &address, &staff, boolsizeof(staff),
              CMC_no_field);
   }
}
```

The code proceeds as follows:

1. It declares the shape, and declares and initializes the parallel structure. (The initialization of **staff.salary** and **staff.id** is omitted.)

2. It calls **rank** to return the ranks of the elements of **staff.salary**. The results (assuming only a 5-position shape) are shown in Figure 93.

3. It calls **make_send_address** to return send addresses, using the salary ranks as the index. Upon return, **[0]address** contains the send address of position [1] of shape **employees**, **[1]address** contains the send address of position [0] of **employees**, and so on.

4. It then calls **send** to send the variables in the parallel structure to new positions, based on the send addresses. The result is that the values are rearranged as shown in Figure 94.

```
order = rank(staff.salary, 0, CMC_upward, CMC_none,
  CMC_no_field);
```

shape **employees**

	0	1	2	3	4
staff.id	50	51	52	53	54
staff.salary	530	230	616	614	800
order	1	0	3	2	4

Figure 93. Using the **rank** function to rank elements of a parallel variable

```
address = make_send_address(employees, order);
send(&staff, &address, &staff, boolsizeof(staff),
    CMC_no_field);
```

shape **employees**

	0	1	2	3	4
staff.id	51	50	53	52	54
staff.salary	220	530	614	616	800
order	1	0	3	2	4

Figure 94. Using **make_send_address** and **send** to reorder the elements of parallel variables by rank

14.4 Communicating with the Front End

This section discusses C* communication functions that provide general communication between the front end and parallel variables on the CM.

14.4.1 From the CM to the Front End

The read_from_position Function

Use the **read_from_position** function to read a value from a parallel variable element (not necessarily of the current shape) and assign it to a front-end variable. This function is overloaded for any arithmetic type.

The **read_from_position** function has the following definition:

```
type read_from_position (
    CMC_sendaddr_t send_address,
    type:void *sourcep);
```

where:

send_address
is the send address of a position from which a value is to be read.

sourcep is a scalar pointer to the parallel variable from which a value is to be read; the parallel variable can be of any shape and any arithmetic type.

Before calling **read_from_position** (or as part of the **read_from_position** call), you must use the single-address version of **make_send_address** to store a send address on the front end; see Section 14.1. The **read_from_position** function uses this send address to specify the position, and it uses **sourcep** to specify the parallel variable. It returns the value obtained from the parallel variable element at that position. The value is returned even if the position is inactive.

Since **read_from_position** deals with a scalar value, it does not have to be called within the scope of a **with** statement, and the source parallel variable does not have to be of the current shape.

This function, in combination with **make_send_address**, produces the same result as assigning a scalar-indexed parallel variable to a front-end variable. For example:

```
scalar = [7]p1;
```

You can use **read_from_position** even when the rank of the shape is not known until run time, however.

The following example reads the value from element [16][4] of parallel variable **p1**, which is of shape **image**. It assigns the value to the scalar variable **s1**.

```
#include <cscomm.h>

shape [256][256]image;
float:image p1;
CMC_sendaddr_t address;
float s1;

main()
{
   address = make_send_address(image, 16, 4);
   s1 = read_from_position(address, &p1);
}
```

Note that the call to **make_send_address** can also be made from within **read_from_position**'s argument list:

```
s1 = read_from_position(make_send_address(image, 16, 4), &p1);
```

The read_from_pvar Function

Use the **read_from_pvar** function to read the values of active elements of a parallel variable and assign them to a front-end array. This function is overloaded for any arithmetic type. It has the following definition:

```
void read_from_pvar (
    type *destp,
    type:current source)
```

where:

destp is a pointer to the scalar array to which values are to be written.

source is a parallel variable of the current shape from which values are to be read. Both **source** and the array pointed to by **destp** must have the same arithmetic type.

The values in **source** are written into the specified front-end array. Values in inactive elements are not copied; array elements that correspond to inactive positions receive undefined values. Typically, the front-end array will have the same number of elements and dimensions as the source parallel variable. It cannot have fewer elements than the source parallel variable.

The following example copies the values in **p1** to the front-end array **fe_array**:

```
#include <cscomm.h>

shape [16384]ShapeA;
int:ShapeA p1;
int fe_array[16384];

main()
{
   /* Initialization of p1 omitted */

   with (ShapeA)
      read_from_pvar(fe_array, p1);
}
```

14.4.2 From the Front End to the CM

The write_to_position Function

Use the **write_to_position** function to write a value from the front end to a parallel variable element (not necessarily of the current shape). The **write_to_position** function has the following definition:

```
type write_to_position (
    CMC_sendaddr_t send_address,
    type:void *destp,
    type source);
```

where:

send_address
is the send address of the position to which a value is to be written.

destp is a scalar pointer to the parallel variable to which a value is to be written; the parallel variable can be of any shape and any arithmetic type.

source is the front-end variable whose value is to be sent to the destination parallel variable element. Both **source** and the parallel variable pointed to by **destp** must have the same arithmetic type.

The function returns the value of **source**.

As with **read_from_position**, you must use the single-address version of **make_send_address** to store a send address on the front end; see Section 14.1. **write_to_position** uses this send address to specify the position, and it uses **destp** to specify the parallel variable. It sends the value in **source** to the element specified by these arguments. The value is written into this element even if the element's position is inactive.

write_to_position does not have to be called within the scope of a **with** statement, and the destination parallel variable does not have to be of the current shape.

This function, when used along with **make_send_address**, produces the same result as assigning a front-end variable to a scalar-indexed parallel variable. For example:

```
[7]p1 = scalar;
```

You can use **write_to_position** even when the rank of the shape is not known until run time, however.

The following example reverses the example for **read_from_position** in the previous section. It assigns the value of the scalar variable **s1** to element [16][4] of parallel variable **p1**, which is of shape **image**.

```
#include <cscomm.h>

shape [256][256]image;
float:image p1;
CMC_sendaddr_t address;
float s1;

main()
{
   address = make_send_address(image, 16, 4);
```

```
   write_to_position(address, &p1, s1);
}
```

The write_to_pvar Function

Use the **write_to_pvar** function to write data from a front-end array to a parallel variable of the current shape. The function is overloaded for any arithmetic type. It has the following definition:

```
type:current write_to_pvar (
    type *sourcep)
```

where **sourcep** is a pointer to a scalar array from which data is to be written.

The function returns a parallel variable of the current shape containing the values in the front-end array. If there are inactive positions in the shape at the time the function is called, the values in these inactive positions are not overwritten. The front-end array typically has the same number of elements and dimensions as the current shape; it cannot have fewer elements.

The following example reverses the example for **read_from_pvar** shown in the previous section. The front-end array **fe_array** writes its values to the parallel variable **p1**:

```
#include <cscomm.h>

shape [16384]ShapeA;
int:ShapeA p1;
int fe_array[16384];

main()
{
   /* Initialization of fe_array omitted */

   with (ShapeA)
      p1 = write_to_pvar(fe_array);
}
```

14.5 The make_multi_coord and copy_multispread Functions

As we mentioned at the end of Chapter 13, the **copy_multispread** function is comparable to the **copy_spread** function, except that you use it on hyperplanes instead of scan classes.

copy_multispread takes as one of its arguments a *multicoordinate*. The multicoordinate specifies which element of the parallel variable is to be spread through each hyperplane. For example, in the discussion of **multispread** in Chapter 13, we saw that, if we allowed positions to differ along axes 0 and 1 while keeping axis 2 fixed, we created the following two hyperplanes (for a 2-by-2-by-2 shape):

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

and:

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

Choosing an individual element in these hyperplanes requires that you specify only two of the three coordinates, since the third (the coordinate for axis 2) is fixed (it is [0] in the first hyperplane, [1] in the second). The multicoordinate specifies what the coordinates are along the axes that are not fixed. If the multicoordinate specifies [0] for axis 0 and [0] for axis 1, for example, then position [0][0][0] is chosen for the first hyperplane, and [0][0][1] is chosen for the second hyperplane.

To obtain this multicoordinate for a position, use the **make_multi_coord** function. You can then use the multicoordinate in the call to **copy_multispread**. The multicoordinate specifies the desired position in each hyperplane.

make_multi_coord is an overloaded function. It provides three different ways of specifying a position:

- By including the position's coordinates as arguments to the function.

- By specifying an array that contains these coordinates. Use this version if the shape's rank will not be known until run time.
- By specifying the position's send address.

The three versions of **make_multi_coord** have the following definitions:

```
CMC_multicoord_t make_multi_coord (
   shape s,
    unsigned int axis_mask,
    int axis_0_coord, ... );
```

or:

```
CMC_multicoord_t make_multi_coord (
    shape s,
    unsigned int axis_mask,
    int axes[]);
```

or:

```
CMC_multicoord_t make_multi_coord (
    shape s,
    unsigned int axis_mask,
    CMC_sendaddr_t send_address);
```

where:

s specifies the shape for which the multicoordinate is to be obtained.

axis_mask is a bit mask that specifies the axis or axes along which positions in a hyperplane are allowed to differ. Bit 1 corresponds to axis 0, bit 2 to axis 1, and so on. For example, use a bit mask of 3 to specify axes 0 and 1; use 6 to specify axes 1 and 2; use 5 to specify axes 0 and 2.

axis_0_coord
(in the first version) specifies the coordinates of a position in shape **s** along axis 0. Specify as many coordinates as there are axes in the shape.

axes[] (in the second version) is an array that contains the position's coordinates. Specify as many coordinates as there are axes in the shape.

send_address
(in the third version) is the send address for a position in shape **s**. Any position will do.

In all versions, the function returns the multicoordinate for the specified position with the specified axis mask.

The definition of **copy_multispread** is as follows:

```
type:current copy_multispread (
    type:current *sourcep,
    unsigned int axis_mask,
    CMC_multicoord_t multi_coord);
```

where:

sourcep is a scalar pointer to a parallel variable from which values are to be copied. The parallel variable can be of any arithmetic type; it must be of the current shape.

axis_mask is a bit mask that specifies the axis or axes along which positions in a hyperplane are allowed to differ.

multi_coord
specifies the coordinates that determine the elements of the source parallel variable from which values are to be copied.

The function copies the value from each specified element to each active position in that element's hyperplane. It returns a parallel variable containing these values; the parallel variable is of the current shape and has the same arithmetic type as **source**. Values of inactive elements are copied.

14.5.1 An Example

For example, given the following declarations:

```
#include <cscomm.h>

CMC_sendaddr_t address;
CMC_multicoord_t multi_coord;
```

```
shape [128][128][128]ShapeA;
int:ShapeA source, dest;
```

then:

```
address = make_send_address(ShapeA, 0, 0, 1);
```

obtains the send address for position [0][0][1] in shape **ShapeA** and assigns it to the scalar **int address**.

```
multi_coord = make_multi_coord(ShapeA, 3, address);
```

obtains the multicoordinate for this position along axes 0 and 1 (specified by the value 3 for the **axis_mask** argument) and assigns it to the **multi_coord**.

```
with (ShapeA)
  dest = copy_multispread(&source, 3, multi_coord);
```

takes each element of parallel variable **source** specified by the axis mask (3) and the multicoordinate (**multi_coord**) and copies its value into the elements of parallel variable **dest** in the same hyperplane. In other words (for a 2-by-2-by-2 shape):

- The value in **[0][0][0]source** is assigned to **[0][0][0]dest**, **[1][0][0]dest**, **[0][1][0]dest**, and **[1][1][0]dest**.
- The value in **[0][0][1]source** is assigned to **[0][0][1]dest**, **[1][0][1]dest**, **[0[1][1]dest**, and **[1][1][1]dest**.

Appendixes

Appendix A

Improving Performance

This appendix describes ways to improve the performance of C* programs. In some cases, it repeats information included in the body of this guide; in other cases (for example, the discussion of **`allocate_detailed_shape`**), it presents information not discussed elsewhere in the guide. Other performance information may be included in the release notes.

A.1 Declarations

A.1.1 Use Scalar Data Types

If data is scalar, declare it as a regular C variable, so that it is stored on the front end. In other words, do not store constants in parallel variables.

A.1.2 Use the Smallest Data Type Possible

To save storage on the CM, use the smallest data types possible for parallel variables. For example, if the parallel variable is a flag, declare it as a **`bool`**. If it is to have values only from –4 to 17, declare it as a **`signed char`**.

A.1.3 Declare float constants as floats

Declaring **float constants** as **floats** (that is, with the final *f*) reduces the number of conversions that the compiler must make, thereby speeding up the program. For example,

```
float:ShapeA p1, p2;
p1 = p2 * 4.0f;
```

is better than writing the code with just "4.0".

A.2 Functions

A.2.1 Prototype Functions

Using ANSI function prototyping speeds up a program by reducing the number of conversions. For example, a call to an unprototyped function with a **char** will promote the argument to an **int**. The called function must then convert the **int** back to a **char**.

A.2.2 Use current instead of a Shape Name

If a program is to be run with safety on, it is more efficient to define a function to take a parallel variable of the current shape as an argument, rather than a parallel variable of a specified shape. In the latter case, the compiler must take the additional step of determining that the specified shape is current.

A.2.3 Use everywhere when All Positions Are Active

If a function contains statements that are to operate on all positions, regardless of the context in which they are called, you may be able to increase performance by enclosing the function's statements in an **everywhere** statement. The explicit use of **everywhere** lets the compiler use faster instructions that ignore the context.

NOTE: This technique can also work with a program's **main** function.

A.2.4 Pass Parallel Variables by Reference

In function calls, pass a parallel variable by reference (that is, take its address and pass the pointer) if passing the parallel variable by value is not required.

A.3 Operators

A.3.1 Avoid Parallel &&, ||, and ?: Operators Where Contextualization Is Not Necessary

As discussed in Chapter 5, the parallel versions of the `&&`, `||`, and `?:` operators perform implicit contextualization. If you do not require this aspect of the operators' behavior, your code will run faster if you can avoid using them.

For example, if **`p1`** and **`f(p1)`** are known to be 0- or 1-valued, then

```
p2 = p1 & f(p1);
```

is much more efficient than

```
p2 = p1 && f(p1);
```

The former statement avoids contextualization, and it avoids doing a logical conversion of its operands, because it assumes that the two operands have logical values.

Similarly,

```
where ( (p1 < p2) & (p2 < p3) )
```

is more efficient than a version that uses the logical AND operator. The "less-than" relational expressions have logical values; therefore, the use of the logical AND (and the resulting contextualization) is not required.

A.3.2 Avoid Promotion to ints by Assigning to a Smaller Data Type

As discussed in Chapter 5, the compiler evaluates an expression at the precision of the variable to which the expression is assigned, provided that the results are the same as if standard ANSI promotion rules were followed. Otherwise, smaller data types such as **bools** and **chars** are promoted to **ints** when used in expressions. Therefore, explicitly assigning the result of an expression involving these data types to a variable of the same data type will increase performance.

A.3.3 Assign a "where" Test to a bool

When using the **where** statement, it is more efficient to first store the test in a **bool**, and then use the **bool** in the **where**. This is a notable case of the situation discussed in Section A.3.2. For example, the following code:

```
int:current x, y;
  where ((x>1) && (y<2)) {
     /* ... */}
```

is more efficient when it is rewritten as follows:

```
int:current x,y;
bool:current b;

b = (x>1) && (y<2);
  where (b) {
    /* ... */}
```

A.4 Communication

To get the best performance in programs in which parallel variables send values to and receive values from other parallel variables, do the following:

1. If possible, put parallel variables that are to communicate in the same shape.

2. Use grid communication functions instead of general communication functions or the language features (like parallel left indexing) that are the equivalent of general communication functions.

3. Use send operations instead of get operations for general communication.

4. If the program has known, stable patterns of communication that use one axis more than another, use **`allocate_detailed_shape`** to weight the axes.

Some of these points are covered in more detail below.

A.4.1 Use Grid Communication Functions instead of General Communication Functions

As mentioned in Part III of this guide, grid communication is faster than general communication. Therefore, your program will run faster if parallel variables that are to communicate are in the same shape, and you use the grid communication functions for send and get operations.

A.4.2 Use Send Operations instead of Get Operations

For general communication, send operations are up to twice as fast as get operations, and use less storage. If possible, use communication functions and C* code that perform send operations rather than get operations.

In grid communication, send operations and get operations have the same cost.

A.4.3 The allocate_detailed_shape Function

Typically, programs use the C* intrinsic function **`allocate_shape`** to dynamically allocate shapes. If, however, your program has known, stable patterns of communication, you may be able to improve the performance of your program by using the intrinsic function **`allocate_detailed_shape`** instead; this function lets you weight the axes of the shape according to the relative frequency of communication along the axes. C* can then lay out the shape on the CM to optimize performance based on these weights.

Like **allocate_shape**, **allocate_detailed_shape** is overloaded. In one version, you use a variable arguments list to specify each dimension of the shape. In the other, the information about the dimensions is included in an array that is passed as an argument to the function; this format is useful if the program will not know the rank until run time.

Include the header file **<cm/cmtypes.h>** when you call **allocate_detailed_shape**.

The variable-arguments format of the function is as follows:

```
CMC_Shape_t allocate_detailed_shape (
    shape *shapep,
    int rank,
    unsigned long length,
    unsigned long weight,
    CM_axis_order_t ordering,
    unsigned long on_chip_bits,
    unsigned long off_chip_bits, ...
)
```

where:

shapep is a pointer to a shape. The remaining arguments specify this shape, and the function returns this shape.

rank specifies the number of dimensions in the shape.

length is the number of positions along axis 0.

weight is a number that indicates the relative frequency of communication along the axis. For example, weights of 1 for axis 0 and 2 for axis 1 specify that communication occurs about half as often along axis 0. Only the relative values of the **weight** arguments for the different axes matter; for example, weights of 5 for axis 0 and 10 for axis 1 specify the same communication as weights of 1 and 2, or 3 and 6. Specifying the same values for different axes indicates that they have the same level of communication.

ordering specifies how coordinates are mapped onto physical CM processors for the axis. There are three possible values: **CM_news_order**, **CM_send_order**, and **CM_fb_order**.

The value **CM_news_order** specifies the usual mapping, in which positions with adjacent coordinates are in fact represented in neighboring processors on the CM. Specifying any other order slows down grid communication considerably.

The value **CM_send_order** specifies that a position with a lower coordinate than another position also has a smaller send address. This ordering is rare, but it is used in certain applications.

Use the value **CM_fb_order** only if your shape is an image buffer and is to be moved to a framebuffer. For details, see Chapter 1 of the *Generic Display Interface Reference Manual.*

You can specify a different ordering for each axis.

on_chip_bits
off_chip_bits
can be used to specify the mapping of positions to physical processors *only if* the values of the **weight** argument for all axes are the same. Specify 0 for the value of each of these arguments if you use different values for the **weight** argument. For information on how to specify other values for **on_chip_bits** and **off_chip_bits**, consult the description of the **create-detailed-geometry** instruction in the *Paris Reference Manual.*

Include values for **length**, **weight**, **ordering**, **on_chip_bits**, and **off_chip_bits** for as many axes as are specified by **rank**.

The array format of **allocated_detailed_shape** is as follows:

```
CMC_Shape_t allocate_detailed_shape (
   shape *shape_ptr
   int rank,
   CM_axis_descriptor_t axes[]
)
```

where **axes** is an array that contains descriptors for each axis in the shape to be allocated. You can fill in the information about each axis by calling the C* library function **fill_axis_descriptor**, which is defined as follows:

```
void fill_axis_descriptor (
   CM_axis_descriptor_t axis,
   unsigned long length,
   unsigned long weight,
   CM_axis_order_t ordering,
   unsigned long on_chip_bits,
```

```
    unsigned long off_chip_bits
)
```

where **axis** is an array element that corresponds to the axis being described, and the remaining arguments are defined as above.

As an intrinsic function, **allocate_detailed_shape** can be used as an initializer at file scope. Thus, you can do the following:

```
#include <cm/cmtypes.h>

shape s = allocate_detailed_shape(&s, 2, 256, 2, CM_news_order,
  0, 0, 512, 1, CM_news_order, 0, 0);
```

This statement fully specifies a 256-by-512 shape **s**, for which you expect communication to occur twice as often along axis 0 as along axis 1.

A.5 Parallel Right Indexing

Parallel right indexing, as described in Chapter 7, becomes less efficient as the range of the array indexes increases.

For users familiar with Paris: The performance of parallel right indexing is comparable to **aref** and **aset** calls, rather than **aref32** and **aset32** calls.

A.6 Paris

Although generally not necessary, it may be possible to improve performance by calling Paris, the CM parallel instruction set, from within a C* program. For details on how to do this, see Chapter 2 of the *C* User's Guide*.

Glossary

active — Of elements and positions: Participating in parallel operations. Parallel operations within a **where** statement are carried out only on parallel variable elements left active by the **where** statement.

axis — A dimension of a shape. Axes are numbered starting with 0 and are read from left to right in a left index. For example, if a shape is declared as "**[256][512]ShapeA**", shape **ShapeA** has 256 positions along axis 0 and 512 positions along axis 1.

bool — An unsigned single-bit integer data type.

collision — An attempt by more than one parallel variable element to send values to or get a value from the same element at the same time. C* provides mechanisms for avoiding collisions.

combiner type — In communication functions: The type of operation to be carried out by the function—for example, add values, multiply them, or perform a bitwise logical AND.

context — The active positions of a shape as set by a **where** statement.

coordinate — A number that identifies a position or an element along an axis. For example, the coordinates of parallel variable element **[6][14]p1** are 6 for axis 0 and 14 for axis 1.

corresponding elements — Elements of different parallel variables that are at the same position. Corresponding elements have the same coordinates and the same shape.

current shape — The shape on whose parallel variables parallel operations can be performed. The **with** statement selects the current shape.

current *predeclared shape name* A shape name that C* equates to the current shape. Variables declared to be of shape **current** (for example, in a function) are of the shape that is current when the declaration is made.

direction In communication functions: The direction along an axis in which a function is to perform its operation. An upward direction is from lower-numbered coordinates to higher; a downward direction is from higher-numbered coordinates to lower.

element An individual data point of a parallel variable. A parallel variable has one element at each position in its shape.

exclusive operation In communication functions: An operation that excludes the first position of a segment-bit scan set, and that includes the first position of a start-bit scan set in the operation for the preceding scan set. Compare *inclusive operation.*

general communication Communication in which any parallel variable element can send a value to or get a value from any other element, whether or not their positions are in the same shape. Compare *grid communication.*

get operation An operation in which a parallel variable gets values from another parallel variable. For example: "**dest = [index]source;**".

grid communication Communication in which a parallel variable sends values to or gets values from another parallel variable in the same shape, using the coordinates of the parallel variable's elements. Compare *general communication.*

hyperplane In communication functions: A set of positions whose coordinates are allowed to differ along more than one axis. Compare *scan class.*

inactive Of elements and positions: Not participating in parallel operations.

inclusive operation In communication functions: An operation that includes the first position of the scan set. Compare *exclusive operation.*

intrinsic function — A function that is defined as part of the language.

left indexing — A method of specifying an element or elements of a parallel variable, or the dimension(s) of a shape, using values in brackets to the left of the variable or shape's name.

multicoordinate — A value obtained by the **make_multi_coordinate** function that specifies which element of a parallel variable is to be spread through each hyperplane for the **copy_multispread** function.

notify bit — In the **send** function: a **bool**-sized parallel variable, each element of which can be set when the corresponding element of the destination parallel variable receives a value.

parallel operation — An operation carried out on more than one element of a parallel variable at the same time.

parallel variable — A variable consisting of multiple data points, called *elements*, arranged in a specified shape. The declaration "**int:ShapeA p1;**" declares **p1** to be an **int**-length parallel variable of shape **ShapeA**. Compare *scalar variable*.

pcoord *function* — An intrinsic function that returns a parallel variable whose elements are initialized to their coordinates along a specified axis.

physical *shape* — A shape predeclared by C*. It is a 1-dimensional shape, with the number of positions equal to the number of physical processors allocated to the program at run time.

position — An area of a shape that can contain parallel variable elements. A shape declared as **[8192]ShapeB** contains 8192 positions, arranged along one dimension. A parallel variable of a given shape has an element in each position of that shape.

predeclared shape name — A shape name provided as part of the language. The three predeclared shape names are **current**, **physical**, and **void**.

promotion — Changing a scalar variable into a parallel variable by replicating the value of the scalar variable in each position of the shape.

rank — The number of dimensions of a shape. A shape declared as **[512][256]ShapeA** has rank 2. A shape can have up to 31 dimensions.

reduction operator — An operator that reduces a parallel variable to a single scalar value by performing a combining operation. For example, the reduction operator **+=** adds the values of active elements of a parallel variable.

region — In C* debugging: A specified subset of a shape's positions on which certain debugging functions are to operate.

sbit — In communication functions: A **bool**-sized parallel variable. An element of an sbit, when set to 1, marks the beginning of a scan set at the element's position. An sbit can be interpreted as a *segment bit* or as a *start bit*, depending on the value of the **smode** argument to the function.

scalar variable — A standard C variable, having only one value. Compare *parallel variable.*

scan class — In communication functions: A set of positions whose coordinates differ only along a specified axis. Compare *hyperplane*, *scan set.*

scan set — In communication functions: A subset of a scan class, the beginning of which is marked by an sbit.

segment bit — In communication functions: The interpretation of an sbit when the value of the **smode** argument is **CM_segment_bit**. When an sbit is a segment bit: 1) the sbit starts a scan set when the value of its element is 1, whether or not it is in an active position; 2) scan sets are not affected by the direction of the operation; and 3) operations in one scan set never affect values of elements in another scan set. Compare *start bit.*

send address — An address that, along with a position's coordinates, uniquely identifies that position among all positions in all shapes.

send operation An operation in which a parallel variable element sends a value to another element. For example: "**[index]dest = source;**".

shape A template for parallel data. A shape is declared in a **shape** statement and consists of a number of positions organized in up to 31 dimensions. All parallel variables must have a shape, and no parallel operations can be carried out unless a shape is made current by a **with** statement.

shape-valued expression An expression that can be resolved to a shape name, and can be used anywhere a shape name is used. For example, "**shapeof(p1)**" returns the name of the parallel variable **p1**'s shape and can be used in place of that shape's name.

start bit In communication functions: The interpretation of an sbit when the value of the **smode** argument is **CM_start_bit**. When an sbit is a start bit: 1) an sbit starts a scan set only when the value of its element is 1 and the element's position is active; 2) when the direction is downward, scan sets are created from the higher coordinate to the lower coordinate; and 3) in an exclusive operation, the position whose sbit element is 1 receives a value from the preceding scan set, if there is one. Compare *segment bit.*

torus A doughnut-shaped surface. C* "torus" communication functions use a grid as if it were wrapped into a torus, with the opposite borders of the grid connected. An element that requires a value from beyond the border gets it from the other side of the grid.

void *predeclared shape name* An extension of the ANSI keyword **void**. It specifies a shape without indicating what the shape's name is. The **void** predeclared shape name can be used only as the target shape of a scalar-to-parallel pointer.

where *statement* A statement that sets the context for parallel operations within its body. For example, "**where (p1 == 4)**" causes parallel operations to be carried out only on elements in positions where the parallel variable **p1** is equal to 4.

with *statement*	A statement that chooses the current shape. Parallel operations within the body of a **with** statement must (with some exceptions) be carried out on parallel variables of the current shape.
wrapping	In communication functions: Obtaining values from the other side of the grid.

Index

Symbols

A

B

C