A Reference Description of the C*® Language

James L. Frankel

Technical Report TR-253 as of May 16, 1991

Second printing, May 1993

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

Preface

This document served as the reference specification during the design and implementation of the C* compiler. It has been updated to reflect the resolution of syntax and semantic ambiguities, further clarifications on points of the language, corrections of errors, and extensions. This document does not necessarily describe the current state of the implementation of C*. For documentation on the current C* implementation, please see the C* documentation products available from Thinking Machines Corporation. They are the newest versions of *Getting Started in C**, *C* Programming Guide*, *C* User's Guide*, and *C* Release Notes*. An overview of the architecture of the Connection Machine® Systems is available in the appropriate *Connection Machine Technical Summary*.

Acknowledgments

Much of the current version of C* is based on the previous version of C*. The previous C* design and implementation was the work of Sam Kendall, John Rose, and Guy Steele. The design, implementation, documentation, and support of C* would not have been possible without the tireless work of the other members of the C* group who worked with me. The other members, past and present, are Mike Best, Rich Bowker, Steve Goldhaber, Harold Hubschman, Karen Jourdenais, Linda Seamonson, Josh Simons, Steve Sistare, Toby Weinberg, and summer members Will Cohen, Tom Cormen, and Robert Morris. Specifically, while I was responsible for the language design and the overall architecture and management of the project, Josh Simons was responsible for the compiler's front end, error mechanism, scripts, and source control system; Karen Jourdenais, Linda Seamonson, and Toby Weinberg were responsible for the middle end; Steve Sistare was responsible for the optimizer and storage allocation component; Steve Goldhaber was responsible for the code generator generator and the code generator; Karen Jourdenais was responsible for the run-time system and tests; the communications library and compiler support were provided by Mike Best; and the user documentation was written by Rich Bowker. Many people at Thinking Machines Corporation have been active in reviewing the designs and testing the new compiler. These people include Andrew Lumsdaine, Gary Oberbrunner, Jim Salem, Peter Schröder, L. Miguel Silveira, Craig Stanfill, and Lew Tucker. For management support and encouragement, Dick Clayton, Bob Millstein, and Ted Tabloski have made it possible for the group to work on producing this revised version of the C* language and compiler. In addition to those people listed above by name, there are many others whom I have neglected to list and without whose help the language and compiler would not have been possible.

J. L. F.

Table of Contents

1 Introduction

The C* language implemented for version 6.0 is different from the previous language accepted by the version 4.3 and 5.0 compilers. The improvements include cleaner treatment of data types, removal of the grid package and integration of grid operations into the language, access to scans and spreads, and a variety of other changes. Rather than focus on all the ways that the current version differs from the previous language, this document will present the complete C* language.

It is expected that the reader is well versed in C and, furthermore, that the reader is familiar with Standard C (ISO C Standard ISO/IEC 9899-1990 (E); ANSI C standard ANSI X3.159-1989). Although not mandatory, an understanding of the Connection Machine system and Connection Machine System Software is useful in understanding this document.

In addition to the extensions described within this technical report, C* accepts any Standard C program and correctly compiles it for execution — without any parallelism.

2 Goals of the C* Language, Version 6.0

- Continue in the C tradition of an efficient, fairly low-level systems programming language.
- Except for C* extensions, the language should follow Standard C.
- The language should support data parallel programming idioms that C programmers can understand and use effectively. It should be possible to write both operating systems and libraries in the language. Even though the language should be data parallel, compilers should be able to produce efficient code for both SIMD- and MIMD-type architectures.
- Efficiently allow access to all user-visible components of a distributed memory massively parallel system, such as the Connection Machine system (e.g., router, n-dimensional NEWS grid, scans, spreads, reductions). That is, the code produced by the compiler should be almost as efficient as can be achieved on the machine.
- Allow dynamic behavior in the allocation and deallocation of parallel storage.
- Allow layered object-oriented extensions (even though these are not included in the current language).
- Simplify previous C* pointer types and behavior (at least by guaranteeing the efficiency of those that are available).
- Allow access to low-level operations (such as Paris instructions; see the *Paris Reference Manual* for complete information on Paris) from C*. This is provided by means of function calls.

2.1 Programming Abstraction Presented by the Language

The language presents an abstraction of the machine known as the *global view*, which treats parallel data as a new entity that is acted upon by new or overloaded operators and statements. That is, parallel variables are seen as monolithic vectors or arrays that are operated on as a whole. In particular, unlike previous versions of C*, version 6.0 does not support a local view — where the programmer can imagine writing a C program for an individual processor, yet the program runs on all processors seemingly independently. This change in view was required to present a parallel programming abstraction that did not contain hidden execution costs. The local view requires that invisible multiprocessing (support for multiple virtual program counters) and synchronization code be generated by the compiler. This was in conflict with the desire for an efficient, fairly low-level language.

2.2 Terminology

This language specification uses the term *scalar* in a manner different from the C Standard's usage of the term. This document's use of the terms is consistent with the standard meanings of the terms *scalar* and *parallel* in the parallel processing industry. In the Standard C document, *scalar* is used to refer to "arithmetic types and pointer types ... collectively." That is, it refers to any integral, floating-point, or pointer type. In this specification, the term is used to refer to all *non-parallel* types. Therefore, all traditional C data types are referred to as scalar data types. When a traditional C data type is qualified to be parallel, it is referred to as a *parallel data type*. When the Standard C use of *scalar* is mentioned in this specification, the term *Standard-C-scalar* will be used.

2.3 New Reserved Words

C* has added the following reserved words to Standard C: **allocate_detailed_shape**, **allocate_shape**, **bool**, **boolsizeof**, **current**, **dimof**, **everywhere**, **overload**, **pcoord**, **physical**, **positionsof**, **rankof**, **shape**, **shapeof**, **where**, and **with**.

2.4 New Operators

C* has added the following operators to Standard C: **<?**, **<?=**, **>?**, **>?=**, **%%**, and **boolsizeof**. In addition, several operators have new overloadings. The index operator ([and]) may now be used as a unary prefix operator. The compound assignment operators may now be used as unary prefix operators. Many operators have new overloadings to work with parallel types.

3 Parallel Data Allocation

3.1 Shape

Unlike previous versions of C*, version 6.0 allocates parallel data only when such data is tagged with a *shape*, which specifies the rank, dimensions, and layout of *parallel* data. **shape** is a new reserved word in C*, and is added to the list of acceptable *type-specifier*s in Standard C. A new construct, called *left indexing*, is the means used to declare shapes. Left indexing is simply using the traditional C brackets to specify an index that precedes rather than follows the indexed expression. Examples of shape specifications follow:

```
shape Sa, [10]Sb, [50][30]Sc;
shape [30][50]Sd, []Se, [][]Sf;
```

The names of shapes and variables declared throughout this document are meant to be both consistent and cumulative. Therefore, all references to a variable or shape are to one that is previously declared in this document. For clarity, variables, shapes, types, and other identifiers may be redeclared in this document; all such redeclarations are identical to prior declarations.

Shape identifiers have the same scope as non-label identifiers in C. [See the discussion of scope in the C Standard §3.1.2.1.] Shape identifiers are in the same name space as *ordinary identifiers*. [See the discussion of name spaces of identifiers in the C Standard §3.1.2.3.] This class includes variables, functions, typedef names, and enumeration constants.

Shapes **Sa**, **Se**, and **Sf** in the shape declarations above are not *fully specified.* Shape **Sa** does not even have its rank specified — it is said to be *fully unspecified.* Shapes **Se** and **Sf** have their ranks specified, but not their dimensions — they are said to be *partially specified.* Shapes **Sb** and **Se** are of rank one; shapes **Sc**, **Sd**, and **Sf** are of rank two. It is invalid to specify some but not all dimensions of a shape. Thus, the language does *not* allow a shape to be specified as **shape [][10]Sg;**. All of the shapes specified above may be used in the allocation of parallel data once they have become fully specified. Shape **Sb** is a shape with 10 *positions*. The term *position* is used to refer to a potential slot within the framework established by a shape. Left indices are numbered from left to right starting with index 0. Index 0 corresponds to the *row* number and index 1 corresponds to the *column* number. For example, index 0 of **Sc** is 50, and index 1 of **Sc** is 30; index 0 of **Sd** is 30, and index 1 of **Sd** is 50.

Implementation restriction: The initial implementation of C* version 6.0 restricts the possible shape declarations that are acceptable. It requires each dimension of a shape to be a power of two and the total number of positions of a shape to be a multiple of the number of processors in the machine on which the program is to be executed.

It is also possible to have arrays of shapes or a pointer to a shape, as follows:

```
shape [2][10]Sarray1[4];
shape Sarray2[40];
shape []Sarray3[20];
shape *Sptr;
```

Sarray1 is an array of four shapes; they are all of rank two with dimensions 2 by 10. Note that even though all shapes in the array have the same rank and dimensions, each of the shapes has its own context [see §6.2]. **Sarray2** is an array of forty shapes, all of which are fully unspecified. **Sarray3** is an array of twenty shapes, all of which are of rank one. **Sptr** is a pointer to a shape (which may be allocated by means of standard memory allocation routines [see §3.3]).

Left index is a unary prefix operator; it has lower precedence than the conventional right index operator. The left index operator is grouped with *unary operators* in the Standard C precedence levels, whereas the right index operator is a *postfix operator.*

As in Standard C, if used at file-scope or with **extern** or **static** at block scope, the expressions used as left or right subscripts in the shape statement must be constant expressions (the notation for axis alignment, in section §3.10, is allowed as well) . In all other cases, any expressions of integral type are permissible. The syntax of the shape declaration statement is as follows:

```
shape left-indexed-declarator-list;
shape declarator-list = initializer;
```

If the shape declarator is not fully specified, it may be initialized. If the shape declarator is an array, the elements may be initialized by a list of shape-valued expressions. As in Standard C, the initializer must be legal in its scope (file scope or block scope). At file scope, the intrinsic functions **allocate_shape** and **allocate_detailed_shape** [see §9] may be used as initializers, but their arguments must be constant expressions. This is the reason that **allocate_shape** and **allocate_detailed_shape** are intrinsic functions. At block scope, any shape-valued expression may be used to initialize the shape. If partially specified, the rank of the shape-valued expression used as an initializer must agree with the rank specified by the left indices.

Note that an initializer may be used *in a shape declaration statement* to declare or define an array of shapes in which each array element is of a different size or shape. It is also possible to have each element of an array of shapes differ in rank or dimensions by using assignment statements to define individual elements in the array of shapes.

A shape functions as a parallel template for the allocation of variables of that shape. When a variable is actually declared of a shape [see §3.4] or when a shape is selected [see §4], the shape must be fully specified: that is, it must then have a known rank, and each dimension must be defined. A shape itself may not be declared as a parallel variable (e.g., **shape:S [10]R**).

The shape must be declared appropriately in all compilation units that reference it. The shape must be tagged as **extern** without an initializer in all compilation units but one, and one compilation unit must either have an initializer for the shape or declare the shape without a storage class specifier (this is in keeping with the Standard C combination of the Strict Ref/Def model and the Initialization model [see Rationale for Standard C, §3.1.2.2]).

3.2 Predeclared Physical Shape

There is a predeclared shape identifier, **physical**, which is a one-to-one mapping to physical processors in the massively parallel computing system. **physical**, which is a new reserved word, is always of rank one, and its dimension is the same as the actual number of physical processors. The programmer may allocate parallel variables of **physical** shape and know that they will have a VP-ratio of one and will be accessed via *physical* instructions, if such instructions exist, whenever possible. However, the predominant use of **physical** probably will be to cast a variable from some other shape into the **physical** shape and then to act upon it as if it were of **physical** shape (i.e., explicitly writing VP loops on parallel data). Please refer to §10 for more information on casting.

3.3 Dynamic Shape Object Allocation

A shape object is, in essence, a descriptor for a shape — in Paris, a VP-Set-ID. In general, C* does not allow direct manipulation of the shape object. Instead, information in the shape object is used implicitly in the language or is accessed via intrinsics. However, certain features are available to allow dynamic allocation of shapes. The **sizeof** operator may be applied to a shape or to the **shape** type to return the number of bytes in a shape object. This capability is needed so that the programmer can use a storage allocation system call to allocate storage for shapes. For example:

```
shape *Sptr, [50][30]Sc;
Sptr = (shape *) malloc(sizeof(Sc));
```

and

```
shape *Sptr;
Sptr = (shape *) malloc(sizeof(shape));
```

each allocate a new shape object that can be referenced by indirecting **Sptr**.

The syntax for **sizeof** applied to the **shape** type is:

size_t sizeof(**shape**)   [*size_t* is defined in <stddef.h>]

3.4 Declaring Parallel Variables

Once a shape is fully specified, variables may be declared in that shape. Using these shapes:

```
shape Sa, [10]Sb, [50][30]Sc, [30][50]Sd, []Se, [][]Sf;
shape [2][10]Sarray1[4], []Sarray3[20], *Sptr;
```

the following are all legal declarations of parallel variables:

```
int:Sa ai1, ai2;     or equivalently:     int ai1:Sa, ai2:Sa;
int:Sb bi1, bi2;
float:Sb bf1, bf2;
double:Sb bd1, bd2;
int:Sc ci1, ci2;
int:Sd di1, di2;
int ei1:Se, ei2:Se, fi1:Sf, fi2:Sf;
int:(Sarray1[2]) pv1;
int:(*Sptr) pv2;
int:(*(Sarray3+4)) pv3;
int:(Sarray1[f(x)]) pv4;
```

Parallel variables **ai1** and **ai2** are of shape **Sa**; **bi1**, **bi2**, **bf1**, **bf2**, **bd1**, and **bd2** are of shape **Sb**; **ci1** and **ci2** are of shape **Sc**; **di1** and **di2** are of shape **Sd**; **ei1** and **ei2** are of shape **Se**; **fi1** and **fi2** are of shape **Sf**. Parallel variable **pv1** is of shape **Sarray1[2]**; **pv2** is of the shape to which **Sptr** points; **pv3** is of shape **Sarray3[4]**; and **pv4** is of the shape **Sarray1[f(x)]**. Note that each shape-valued expression that is used in a declaration is evaluated once per declaration. This is important when the expression may cause side effects — as in the declaration of **pv4** above.

Parallel variables **bi1**, **bi2**, **bf1**, **bf2**, **bd1**, and **bd2** (declared of shape **Sb**) will each consist of ten *elements*; they will exist on ten virtual processors. Parallel variables **ci1**, **ci2**, **di1**, and **di2** will each exist on 1500 virtual processors — with **ci1** and **ci2** organized 50 by 30 and **di1** and **di2** organized 30 by 50. Parallel variable **pv1** will exist on twenty processors, organized 2 by 10.

The declaration statements for parallel variables have the following syntax:

type-specifier:
- *signed-type-specifier*
- *floating-type-specifier*
- *unsigned-type-specifier*
- *character-type-specifier*
- *boolean-type-specifier*

signed-type-specifier:
- signed: *shape-qualifier*
- signed$_{opt}$ int: *shape-qualifier*

signed$_{opt}$ short int$_{opt}$: *shape-qualifier*

signed$_{opt}$ long int$_{opt}$: *shape-qualifier*

floating-type-specifier:

float: *shape-qualifier*

double: *shape-qualifier*

long double: *shape-qualifier*

unsigned-type-specifier:

unsigned short int$_{opt}$: *shape-qualifier*

unsigned int$_{opt}$: *shape-qualifier*

unsigned long int$_{opt}$: *shape-qualifier*

character-type-specifier:

char: *shape-qualifier*

signed char: *shape-qualifier*

unsigned char: *shape-qualifier*

boolean-type-specifier:

bool: *shape-qualifier*

declarator:

declarator: *shape-qualifier*

abstract-declarator:

abstract-declarator: *shape-qualifier*

If the shape qualifier is a simple shape name or the application of the intrinsic function **shapeof**, then parentheses are not needed around it. In all other circumstances, the shape qualifier should be enclosed within parentheses. This is required in general to allow unambiguous parsing of the expression.

If the shape qualifier (i.e., the specification of the shape in the declaration of a parallel variable) is part of the type-specifier, then the type is qualified to be a parallel type of the specified shape and, therefore, applies to all declarators specified in that declaration statement. If the shape qualifier is part of a declarator, then just that declarator is qualified to be of the specified shape. Only one shape qualifier may be applied to either a type-specifier or a declarator, and a shape qualifier may not be specified on both the type-specifier and the declarator in a declaration statement (so as not to be misleading).

A shape is the name of a shape, such as **Sb** or **physical**, or a shape-valued expression. A parallel type is a type-specifier that includes a shape, such as **int:Sb** or **int:physical**. A parallel variable is a variable declared to be of a parallel type, such as **bi2**.

If the declaration or definition appears at file scope or is **static** or **extern**, then the shape-valued expression must be constant. The definition of a constant expression is extended for shape-valued expressions. In particular, a constant expression may be a simple shape that is fully specified at compile time or that has storage class **extern**, an array of shapes that is fully specified at compile time and whose right index is a constant expression, or an indirection of the sum of an array of shapes that is fully specified at compile time and a constant expression. For example, given these shapes,

```
    shape [10]Sb, []Se, [2][10]Sarray1[4], Sarray2[40], *Sptr;
```

the following are valid constant shape-valued expressions:

```
    Sb
    Sarray1[4-3]
    *(Sarray1+(2*2)-2)
```

But the following expressions are not:

```
    Se
    Sarray2[4-3]
    *(Sptr+(2*2)-2)
```

The declarations of **pv2**, **pv3**, and **pv4** above could not appear in a context where a constant shape expression would be required. For **pv2** and **pv3**, **Sptr** and **Sarray3** are not fully-specified shapes. For **pv4**, **f(x)** is not a constant expression — it invokes a function whose result is not known until run time.

C* has borrowed the same syntax for shape qualifiers that is used for bit-fields. The shape specification is differentiated from the bit-field specification based on the type of the expression to the right of the colon. If the expression is of type shape, then the qualifier indicates a parallel variable in the specified shape; if the expression is of integral type (actually a non-negative constant integral expression in Standard C), then the qualifier indicates the bit-field width.

The ambiguity is also resolved by the current restrictions that a bit-field must be a non-negative constant integer expression, that bit-fields may appear only within a struct, that shape qualifiers may appear only outside a struct, and that a shape qualifier may appear on the type-specifier whereas bit-fields may not. However, these current restrictions may change (if, for example, bit-fields were allowed outside **structs**) and, therefore, are not the differentiating features.

The potential ambiguity is shown in the following declarations:

```
    struct struct1 {
       int x:y;
       } z;
    int ai3:Sa;
```

Is this an attempt to declare a parallel int **x** of shape **y** or to declare an int **x** with field width **y** (where **y** might have appeared in a prior **#define**)? Is **ai3** declared as a parallel int of shape **Sa** or as an int **ai3** with field width **Sa**? As previously stated, the rule above resolves the potential ambiguity based on the types of **y** and **Sa**.

The shape is part of the type specifier and must appear in the order shown above; however, it is possible to have storage class specifiers and type specifiers in either order. It is customary in C and C* programming to write the storage class specifiers first (if they are present). A shape may be either the name of a previously declared shape that is in the scope of the declaration and is visible, or it may be a shape-valued expression.

An external parallel variable must be declared in all compilation units but one with the **extern** keyword and without an initializer. In one compilation unit, the parallel variable must be declared either with an initializer, without a storage class specified, or both with an initializer and without a storage class specified. This is to be consistent with the Standard C linkage model [see Rationale for Standard C, §3.1.2.2].

The following is a more formal treatment of the use of shape-valued expressions. There is a hierarchy of such expressions. The most constrained shape-valued expression is a *constant shape-valued expression.* A constant shape-valued expression may be used as an initializer for shapes declared at file scope. In addition to syntactic entities allowed in a Standard C constant expression, such an expression may contain use of file scope shape names, the **shapeof** intrinsic function applied to compile-time fully specified constant shape-valued expressions, and dereferencing and indexing, but may not contain the use of any potentially side-effecting operators — such as assignment operators, increment or decrement operators, function calls, or comma operator — except if they are not evaluated (for example, as operands of the **sizeof** or **boolsizeof** operators).

The next, less constrained shape-valued expression is a *file scope shape qualifier.* A file scope shape qualifier is used as the shape qualifier in the declaration of parallel variables at file scope. Such an expression encompasses all constant shape-valued expressions and, in addition, allows the inclusion of the **void** shape name and the **physical** shape name.

At the next level is the *parameter scope shape qualifier.* In a function declaration, a parameter scope shape qualifier may be applied to a parameter of that function or to the return value of that function. This qualifier may be applied at file or block scope. It encompasses the attributes of file scope shape qualifiers and, in addition, allows the inclusion of visible shape names at parameter scope and use of the **current** shape name.

At the last level is the *block scope shape qualifier.* A block scope shape qualifier is used as the shape qualifier in the declaration of parallel variables at block scope. Such an expression encompasses all file scope shape qualifiers and, in addition, allows the inclusion of any shape-valued expression (this includes use of assignment operators, increment and decrement operators, functions calls, and the comma operator). If the expression is at block scope, but with the **static** storage-class specifier, or has external or internal linkage, the qualifier must be a file scope shape qualifier.

3.4.1 Parallel Enumerated Types

C* also supports the use of enumerated types in parallel variables. The enumeration is defined in the usual way; then a parallel **enum** may be specified as the type in declaring a parallel variable. For example:

```
    enum colors {green, yellow, red};
    enum colors:Sb trafficLight0;
    enum colors trafficLight1:Sb;
```

The grammar for a parallel **enum** follows:

enum-specifier:

enum-specifier: shape-qualifier

Like non-enumeration parallel variable declarations, if the shape qualifier is part of the *enum-specifier*, then the type is qualified to be a parallel **enum** type of the specified shape and, therefore, applies to all declarators specified in that declaration statement. If the shape qualifier is part of a declarator, then just that declarator is qualified to be of the specified shape. Only one shape qualifier may be applied to either a type-specifier or a declarator, and a shape qualifier may not be specified on both the type-specifier and the declarator in a declaration statement (so as not to be misleading).

3.4.2 Initializing Parallel Variables

When a parallel variable is defined, it may be initialized. Parallel variables with static storage duration at file or block scope (i.e., parallel variables declared at file scope or parallel variables declared at file or block scope with **extern** or **static** qualifiers) and of *any* shape may be initialized only to a scalar constant expression. Otherwise, initialized parallel variables with automatic storage duration must be of the current shape, and the initializer must be an expression that can be evaluated at its scope. This is consistent with the usual equivalence of initialization of block scope variables and writing that initialization as an assignment statement. That is, it would not be legal in C* to execute such an assignment statement; therefore it cannot be performed in an initializer either. This implies that any parallel code in the initializer must be able to be evaluated in the scope of the current shape when the initializer is reached. There must be a current shape when an initialized parallel variable's definition is reached, and that current shape must be the same as the shape of the variable. Each parallel variable may be initialized to an expression that evaluates to a scalar or a parallel value. If a scalar initializer is specified, all elements of the parallel variable are set to that single scalar value. By default (i.e., when no initializer is present), static variables (including all elements of static parallel variables) are initialized to zero.

Scalar variables may be initialized with an expression that contains parallel operations as long as the expression can be evaluated at its scope (and with the current shape) and evaluates to a scalar value.

For example, the following are legal C* initializing definitions:

```
int:Sa aizero1 = 0, aizero2 = 0;
int:Sb bi37 = 37, bi42 = 42;
float:Sb bfuninit, bfpi = 3.14159265;
double:Sb bdpi = 3.1415926535897932, bduninit;
int:Sc ci11 = 11, ci21 = 21;
int:Sd dizero1 = 0, dizero2 = 0;
int eizero1:Se = 0, eizero2:Se = 0, fizero1:Sf = 0,
     fizero2:Sf = 0;
```

File scope initializers may not contain any parallel operations (including reductions and left indexing). Block scope initializers may contain any appropriate operations. These are executed in the current shape.

3.5 Parallel Structs and Unions

Parallel **structs** and **unions** are supported by the C* programming language. The term *structure* will refer to both **structs** and **unions**. After a usual C structure is declared, parallel variables based on that structure may be declared. When a parallel structure is declared, each of the fields in the structure becomes parallel. Because parallel fields may not appear within structures, an instantiation of a structure is either wholly scalar or wholly parallel. However, the same structure declaration (when the structure itself is declared) may be used for both scalar and parallel structures. In addition, shapes may not be declared within a structure, but a pointer to a shape may exist as a structure field in a scalar structure. [Language designer's note: Shapes are not allowed within a structure because this would allow a compile-time fully specified shape to appear within a structure. This might imply that each new allocation of that structure would create a new shape or might imply that all allocations would share a single, interned shape. This would happen when a parallel version of the structure was defined. We did not want to allow this situation to occur. One possible future relaxation of this restriction would be to allow only partially specified and fully unspecified shapes within structures.] Of course, structures may be nested as in C. The programmer should also be aware of the potential for different structure sizes between parallel and scalar structures, as discussed in §3.7. As does Standard C, C* allows assignment of structures and performs the appropriate translation when assigning between parallel and scalar structures (even though the size and alignment of fields within such structures may differ).

Examples of parallel **struct** and **union** declarations follow:

```
struct Struct2 {
     int i1, i2;
     float f1, f2;
     };
struct Struct2:Sa struct2a;
struct Struct2 struct2b:Sb;
struct Struct3 {
     int i;
     }:Sc struct3c;
struct Struct3 {
     int i;
     } struct4c:Sc;
struct {
     int i;
     }:Sd structun1d;
struct {
     int i;
     } structun2d:Sd;
union Union1 {
     int i1;
     float f1;
     struct Struct2 str1;
     };
union Union1:Sb union1a;
```

Parallel variable **struct2a** is of shape **Sa**; **struct2b** is of shape **Sb**, **struct3c** and **struct4c** are of shape **Sc**, **structun1d** and **structun2d** are of shape **Sd**, **union1a** is of shape **Sb**. The grammar for a parallel **struct** is:

struct-or-union-specifier:

struct-or-union-specifier: shape-qualifier

Like non-structure parallel variable declarations, if the shape qualifier is part of the *struct-or-union-specifier*, then the type is qualified to be a parallel structure type of the specified shape and, therefore, applies to all declarators specified in that declaration statement. If the shape qualifier is part of a declarator, then just that declarator is qualified to be of the specified shape. Only one shape qualifier may be applied to either a type-specifier or a declarator, and a shape qualifier may not be specified on both the type-specifier and the declarator in a declaration statement (so as not to be misleading).

Of course, a structure declaration may contain pointer fields; however, it is a compile-time error to declare a parallel instance of such a structure. An array of non-empty size (i.e., not just empty brackets) may be declared within a parallel instance of a structure. The qualified name of the array will translate to a pointer to the first of the array elements.

3.5.1 Initializing Parallel Structs, Unions, and Arrays

When a parallel structure or array is defined, it may be initialized. The initializer for a **struct** or **union** object that has automatic storage duration must be either an initializer list or a single expression that has compatible structure or union type. All expressions in an initializer list for structures or arrays must be constant expressions. All instances of the field or all such array elements (i.e., all positions of the field or array element) will be set to that field's or element's single scalar initializer value. The initializers are subject to the usual Standard C constraints and semantics presented in §3.5.7 of the Standard. If the structure is a **union**, the initializer applies to the member that appears *first* in the declaration list of the **union** type.

For example, the following are legal C* initializing definitions:

```
struct Struct2:Sa struct2a = {3, 7, 3.14159, 2.7828};
struct Struct2 struct2b:Sb = {1, 2, 3.0, 4.0};
int:Sa arraya[6] = {4, 34, 2, -18, 0, 1};
union Union1:Sb union1a = {71};
```

3.6 Scalar Variables

Variables that are declared (and allocated) without a shape specification — that is, all traditional C variables — are referred to as *scalar* variables. The following are all declared as *scalar* variables:

```
int si1, si2;
float sf1, sf2;
double sd1, sd2;
unsigned char ucArray[15];
```

An individual element (element is defined in §3.7) of a parallel variable (e.g., a single **int** of a parallel **int**) is referred to as a *scalar* value.

3.7 Storage Size Differences

Just as the term *position* is used to refer to a slot within the framework established by a shape, the term *element* refers to the contents of one *position* of a parallel variable. An *element* of a parallel variable and its scalar counterpart do not necessarily occupy the same amount of storage. This may happen because of different basic datum widths (for example, a scalar **bool** [see §14 for boolean] may occupy one byte, but an element of a parallel **bool** might occupy one bit) or because of different data alignment constraints (for example, scalar data types might be aligned on word boundaries, but parallel data types might be aligned on bit boundaries). Therefore, it is necessary to be able to ascertain either storage size. The **sizeof** operator behaves as it always has when its argument is a scalar type, such as **float**. When invoked with a parallel type or a parallel variable, however, it returns the storage requirements of an element of that parallel type in bytes, rounded up to the nearest byte when necessary. For example:

```
    sizeof(float)
```

returns the size of a scalar **float** in bytes; whereas,

```
    sizeof(float:Sb)
```

and

```
    sizeof(bi1)
```

or

```
    sizeof bi1
```

return the size of a parallel **float** in bytes. An example of alignment having an effect on the size of storage allocated is seen when **sizeof** is applied to a **struct** type.

In addition to **sizeof**, a new operator, **boolsizeof**, is added. **boolsizeof** is a new reserved word in C*. Like **sizeof**, **boolsizeof** only requires parentheses enclosing its operand if the operand is a type. **boolsizeof** has the same precedence and associativity as the existing C **sizeof** operator. **boolsizeof** returns the size of its operand in units of the allocation of **bool**s. More explicitly, when **boolsizeof** is applied to a *parallel* type or variable, it returns its allocation in units of *parallel* **bool**s; when **boolsizeof** is applied to a *scalar* type or variable, it returns its allocation in units of *scalar* **bool**s.

```
    boolsizeof(char:Sb)        [See §14 for bool type]
```

would return the allocation of **char:Sb** in units of parallel **bool**s; whereas,

```
    boolsizeof(char)           [See §14 for bool type]
```

would return the allocation of **char** in units of scalar **bool**s.

For parallel types as operands when a parallel **bool** is implemented as a bit, **boolsizeof** returns the actual number of bits required for allocation of a single element of a variable of that parallel type. For parallel variables as operands when a parallel **bool** is implemented as a bit,

boolsizeof returns the actual number of bits that a single element of that variable occupies. Even though it may not be particularly useful, **boolsizeof** may even be invoked with a shape or the **shape** type as its operand. Some examples of using **boolsizeof** follow:

```
    boolsizeof(bool)            [See §14 for bool type]
```

and

```
    boolsizeof(bool:Sb)         [See §14 for bool type]
```

would each return 1; whereas,

```
    boolsizeof bi1
```

or

```
    boolsizeof(int:Sb)
```

might return 32, and

```
    boolsizeof(int)
```

might return 4 if scalar **bool**s are implemented as **char**s, and **int**s are four **char**s in size. The syntax for these operators follows:

size_t sizeof *unary-expression*	[*size_t* is defined in **<stddef.h>**]
size_t sizeof(*type-name*)	[*type-name* is extended to include parallel types as a
size_t boolsizeof *unary-expression*	result of extending the acceptable
size_t boolsizeof(*type-name*)	*type-specifiers*. See §3.4]
size_t sizeof(**shape**)	[**shape** is also allowed as a *type-name*]

Just like the **sizeof** operator, **boolsizeof** does not evaluate its operand.

3.8 Additional Intrinsics

The **positionsof** intrinsic function may be applied to a shape to return the total number of **positions** — or virtual processors to the Paris programmer — in the shape. **positionsof** returns the total number of positions in a shape, not just the number of *active* positions. Active positions of a shape are those positions of the shape that will participate in operations when that shape is selected [See §4 on shape selection]. Therefore, one important property of a shape is that each fully specified shape includes the allocation of the context for that shape. It is precisely this "context" that remembers the active and inactive positions. Two new intrinsic functions, **rankof** and **dimof**, may be applied to a shape to return the rank and dimensions of a shape. Of course, **positionsof**, **rankof**, and **dimof** may not be able to be evaluated at compile time — if applied to a shape that is not fully specified at compile time, they will return the appropriate value at run time. (As will be noted later, **positionsof**, **rankof**, and **dimof** may also be applied to parallel variables.) Thus, these functions are declared as follows:

int positionsof(shape *shape*)

int rankof(shape *shape*)

int dimof(shape *shape*, int *axis*)

rankof returns zero if the shape is fully unspecified; it still returns the rank of its argument even if the shape is just partially specified. **dimof** returns zero if the shape is not fully specified.

When a shape is specified, the left index axes are numbered from left to right starting with zero. The information required by **positionsof**, **rankof**, or **dimof** must be defined — either through compile-time information or through run-time calls — prior to execution of a request for that information. If it is known at compile time that an error will result from a call to one of these intrinsics, then a compile-time error is reported (this may occur, for example, when **dimof** is called to return the dimension of a non existent axis). If it is not known until run time that an error will result, then, with sufficient safety enabled, a run-time error is signalled. Given these declarations,

```
shape [10]Sb, [50][30]Sc, [30][50]Sd, []Se, [][]Sf;
```

the following expressions show uses of **positionsof**, **rankof**, and **dimof**, and all evaluate to true,

```
positionsof(Sb) == 10
(positionsof(Sc) == 1500) && (positionsof(Sd) == 1500)
(rankof(Sb) == 1) && (rankof(Se) == 1)
(rankof(Sc) == 2) && (rankof(Sd) == 2) && (rankof(Sf) == 2)
dimof(Sb, 0) == 10
(dimof(Sc, 0) == 30) && (dimof(Sd, 1) == 30)
(dimof(Sc, 1) == 50) && (dimof(Sd, 0) == 50)
```

3.9 Intrinsics Applied to Parallel Variables

A new intrinsic exists, **shapeof**, which returns the shape of a parallel variable (and, therefore, can be used as a shape-valued expression). It is a syntax error to apply **shapeof** to anything that is not a parallel variable. A use of **shapeof** is given here:

```
int:shapeof(bf1) bi3;
```

This is exactly equivalent to writing:

```
int:Sb bi3;
```

Equality of shapes is based on exact shape object matching (like **eq** in Common Lisp). Therefore, even if two shapes look identical (that is, they have the same rank and dimensions) they are not the same for the purposes of the C* type system. This attribute of shapes is required in C* because each shape has a layout associated with it. The layout is a mapping of the shape's positions onto the processors of a compute engine. Since elemental operations within a shape are guaranteed to be local (i.e., fast), shape equivalence must support that notion — that is, it may equate shapes only if they have the same rank, dimensions, and layout. C* accomplishes this goal through shape object equivalence.

Shapes may be compared by equality operators. Such a comparison (with **==**) will produce a true result if the two shapes are equal (i.e., denote the same shape object). Such a comparison with **!=** will produce a false result if the two shapes are not equal (i.e., denote different shape objects). This comparison is useful when one of the shapes being compared is **current** [see §4 Shape Selection]. For example, comparing the current shape against a known shape within a function could be used to check that the function was called with a particular shape selection. This technique could also be used within an assertion *type-check-block* [see §11.3] to perform the check at compile-time.

The implications of this shape-typing scheme are numerous. Because shapes may be assigned, passed to functions, returned from functions, their addresses taken, dynamically allocated, etc., there may in fact be two shape variables that both refer to exactly the same shape object. The compiler performs an "intermediate shape equivalence" test on parallel variable usage. Let's examine the following C* program:

```
shape [10]Sb;
int:Sb bi3;
shape newShape;

newShape = Sb;
with(newShape) {      /* See §4 for a discussion of the with stmt. */
  int:newShape newVar;

  newVar = bi3;       /* this line causes an error to be signalled */
}
```

The assignment of a parallel variable of shape **Sb** to a parallel variable of shape **newShape** is signalled as an error because they don't both have the same shape name. The compiler does not check all possible shape assignments to determine if, in fact, **Sb** and **newShape** must denote the same shape object. To correct the above program, insert a shape-to-shape cast [see §10] in the erroneous line as follows:

```
  newVar = (int:newShape) bi3;
```

This is then acceptable. The compiler allows operations with a parallel variable of **current** shape and a parallel variable of a named shape, which must be the current shape, as follows:

```
        int:current currentPVar;
        newVar = currentPVar;
```

positionsof, **rankof**, and **dimof** may also be applied to a parallel variable (rather than a shape). This is simply a shorthand for writing **positionsof**(**shapeof**(bi3)), **rankof**(**shapeof**(bi3)), and **dimof**(**shapeof**(bi3), axis). It is a syntax error to attempt to apply **positionsof**, **rankof**, or **dimof** to anything other than shape or a parallel variable. These functions are defined as follows:

int positionsof(*parallel-variable*)

int rankof(*parallel-variable*)

int dimof(*parallel-variable*, int *axis*)

Summary of valid arguments to intrinsics

Operator or Intrinsic	scalar type	scalar expr.	**shape** type	shape expr.	parallel type	par. expr.
sizeof	yes	yes	yes §3.3	yes §3.3	yes §3.7	yes §3.7
boolsizeof	yes §3.7	yes §3.7	yes §3.7	yes §3.7	yes §3.7	yes §3.7
positionsof	no	no	no	yes §3.8	no §3.9	yes §3.9
rankof	no	no	no	yes §3.8	no §3.9	yes §3.9
dimof	no	no	no	yes §3.8	no §3.9	yes §3.9
shapeof	no	no	no	no	no §3.9	yes §3.9

sizeof and **boolsizeof** do not evaluate their arguments; **positionsof**, **rankof**, **dimof**, and **shapeof** do evaluate their arguments. At file scope, **positionsof**, **rankof**, **dimof**, and **shapeof** may be used in a constant expression (in a declaration, for example) if their value is determinable at compile time.

3.10 Shape Axis Alignment

The initialization component of a shape declaration may refer to previous shapes in an axis-by-axis manner. In this way, a new shape may be declared that inherits the dimensions and alignment (bitmask) of a previous shape for any of its axes. This is accomplished by indexing into a shape that is in the scope of a new shape declaration. The index refers to the axis that is to be copied — indices are numbered from zero increasing by one from left to right. Since there can be arrays of shapes, indices of shapes are used first to select the appropriate shape and then to select an axis of a shape. Some examples of the use of shape axis alignment follow:

```
shape [256][512][128]Sq;
shape [Sq[0]][Sq[2]]Sr;
shape [Sq[0]][Sq[1]][Sq[2]][4]Ss;
shape [4][1024][64]St[3];
shape [St[0][2]][St[1][0]][St[2][1]]Su;
shape Sv[3];
allocate_shape(&Sv[0], 3, 64, 1024, 4); /* See §9 for a
allocate_shape(&Sv[1], 1, 65536);          discussion of
allocate_shape(&Sv[2], 2, 16, 512);        allocate_shape */
{
    shape [Sv[0][2]][Sv[1][0]][Sv[2][1]]Sw;
}
```

The declarations above are equivalent to those that follow when the specified alignments are honored:

```
shape [256][512][128]Sq;
shape [256][128]Sr;     0th axis aligned with Sq[0]; 1st axis
                        aligned with Sq[2]
shape [256][512][128][4]Ss;    0th axis aligned with Sq[0];
                        1st axis aligned with Sq[1]; 2nd axis
                        aligned with Sq[2]
shape [4][1024][64]St[3];
shape [1024][64][4]Su;  0th axis aligned with St[0][2]; 1st
                        axis aligned with St[1][0]; 2nd axis
                        aligned with St[2][1]
shape [512][65536][4]Sw;  0th axis aligned with Sv[0][2]; 1st
                        axis aligned with Sv[1][0]; 2nd axis
                        aligned with Sv[2][1]
```

4 Shape Selection

A new statement, the shape selection statement, has been added to C*; this statement selects a current shape. The statement has the following syntax:

with(*shape-expression*) *shape-body*

The *shape-body* is a statement that is executed with the specified *shape-expression* as its current shape. Of course, the *shape-body* may be a block containing declarations and statements. All statements executed within the context of *shape-body* must perform operations only on variables in the current shape or on scalar variables (with some exceptions listed later in this section), unless the operation is within another nested shape selection. Keep in mind that the **with** statement has effect on any code called from within *shape-body* as well. Thus, the current shape is determined by following the dynamic call chain of function invocations to the innermost shape selection statement .

When a C* program begins execution, all positions of all shapes are activated, but no shape is initially selected (i.e., the C* language does not guarantee that there is any default shape; however, an implementation may choose to provide one). A shape selection statement defines a *parallel context* for the dynamic duration of its *shape-body*. A shape selection statement must be used to select a current shape before any parallel code may be executed (with a few exceptions to be presented later in this section). The shape selection statement does not alter the set of active positions in the selected shape: it reestablishes whatever context was last associated with the selected shape. [See §6.2 for a discussion of context.]

A function need not contain any shape selection statement. If it does not, then it will be executed with the current shape of its caller. A predeclared shape identifier, called **current** (a new reserved word), is always equated to the current shape. Thus, it is possible to declare two integer parallel variables in the current shape as follows:

```
    int:current Ci1, Ci2;
```

The statements in the *shape-body* may reference parallel variables only of the current shape, with six exceptions. ¿ If a parallel variable's left indices are all scalar, then the result is treated as a scalar quantity and the parallel variable need not be of the current shape. ¡ A parallel variable in another shape may be left indexed by a parallel variable of the current shape, in order to produce an lvalue or an rvalue of the current shape. ¬ The **boolsizeof** operator and the intrinsic functions **dimof**, **rankof**, **positionsof**, and **shapeof** may be applied to parallel variables that are not of the current shape. √ The address-of operator, **&**, may be applied to a parallel variable that is not of the current shape. ƒ Declarations and definitions of parallel variables are not constrained to be of the current shape. However, if they are not of the current shape, they may not be initialized. ≈ The dot operator, ., may be applied to select a field of a parallel **struct** or **union** of other than the current shape — so long as that field is a non-aggregate type. A scalar pointer to a parallel type of any shape may be dereferenced independent of the current shape because the pointer itself is a scalar; however, the dereference expression is subject to the other constraints above. The compiler will tag errors when it is able to do so — it may not be able to find all errors at compile time. If a sufficiently high level of safety is enabled at run time, then those errors not found at compile time will be detected at run time.

In addition, several of the above operations are legal with no current shape (outside a parallel context): a parallel variable may be left indexed by all scalar indices; the **boolsizeof** operator and the intrinsic functions **dimof**, **rankof**, **positionsof**, and **shapeof** may be applied to parallel variables; the address-of operator, **&**, may be applied to a parallel variable; and parallel variables may be declared and defined.

5 Expression Syntax

The following changes to Standard C may affect the behavior of existing programs. There are several new reserved words in C*; they are listed in §2.3. Some of these reserved words are names of *intrinsic functions*. *Intrinsic function* is a term used for a function about which the compiler needs to have special knowledge. The intrinsic functions are **allocate_detailed_shape**, **allocate_shape**, **dimof**, **pcoord**, **positionsof**, **rankof**, and **shapeof**. No header file need be included to access the intrinsic functions.

The term *built-in function* is used to refer to functions about which the compiler may be aware of in order to have a more efficient implementation. A header file does need to be included when accessing built-in functions. The functions defined in Appendix A are built-in functions when their appropriate header files are included. This technique is similar to that used in Standard C to allow a C compiler to recognize the standard C functions when their header files are included and to produce more efficient (possibly in-line) code and to perform optimizations involving the internal structure of those functions.

5.1 New Minimum, Maximum, and Modulus Operators

C* supports all standard C operators. In addition, several new binary operators have been added. These include the minimum and maximum operators, **<?** and **>?**, and the modulus operator, **%%**. They may be used to provide the minimum, the maximum, or the modulus of their operands. Standard type compatibility and conversions, as described in the Standard C specification for binary relational operators, are performed for **<?** and **>?** [see the C Standard §3.3.8]. The precedence and associativity of the **<?** and **>?** operators is the same as for binary relational operators, as well. Standard type compatibility and conversions, as described in the Standard C specification for multiplicative operators, are performed for **%%** [see the C Standard §3.3.5]. The precedence and associativity of the **%%** operator is the same as for multiplicative operators, as well.

`a <? b` *is equivalent to* `#define min(x,y) ((x)<(y)) ? (x) : (y)`

```
                                    min(a, b)
  a >? b          is equivalent to  #define max(x,y) ((x)>(y) ? (x) : (y)
                                    max(a, b)
```

if the operands to **min** or **max** were evaluated only once.

C* also supports assignment operator versions of the **<?** and **>?** operators. These operators, **<?=** and **>?=**, are defined as follows:

```
  a <?= b;        is equivalent to    a = a <? b;
  a >?= b;        is equivalent to    a = a >? b;
```

except that the left-hand-side, **a**, is evaluated only once.

The modulus operator is added to C* because the **%** operator's result in Standard C is uniquely defined only when both of its operands are positive. The modulus operator evaluates the following formula to compute the result of **a %% b**:

```
a-(b*floor(a/b))
```

A consequence of this formula is that the result always has the same sign as that of the denominator. For example,

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

The modulus operator is used with the communication syntax to provide n-dimensional nearest neighbor communication [see §8.6]. Standard type compatibility and conversions, as described in the Standard C specification for compound assignment operators (like ***=**), are performed for **<?=** and **>?=** [see the C Standard §3.3.16 and §3.3.16.2]. The precedence and associativity of the **<?=** and **>?=** operators are the same as for compound assignment operators, as well.

5.2 Parallel Meanings for Standard C Operators

If two parallel variables are added together and assigned to a third parallel variable, and all variables are of the current shape, each position of the first parallel variable is added to the corresponding position of the second parallel variable and assigned to the third variable's corresponding position. For example,

```
bi1 = bi2+bi3;
```

is equivalent to

```
for(i = 0; i < 10; i++)
   [i]bi1 = [i]bi2+[i]bi3;
```

except that all of the operations are carried out in parallel. The left indices used in the above expression are used to select elements of the parallel variables [see §8]. In general, a parallel binary or ternary operator must have all of its operands of the current shape and will produce a result in the current shape with the operator applied elementally. That is, the operator performs its computation on operands in corresponding positions to produce a parallel result. [Detail: Just as the + operator has both integer and floating-point overloadings in C, it has scalar integer, scalar floating-point, parallel integer, and parallel floating-point overloadings in C*.]

For almost all C* operators, if one operand is parallel and one is scalar, the scalar operand is promoted to a parallel value of the other operand's shape by replicating the scalar value. This replication applies to assignment operators only when the left-hand-side of the operator, or LHS, is parallel and the right-hand-side of the operator, or RHS, is scalar [see §5.3 for those cases in which replication of the scalar does not apply]. The parallel overloadings of Standard C operators are affected by the same type compatibility constraints and conversion semantics that affect the same scalar Standard C operators.

The integral promotions are extended to include new parallel integral promotions. These state that a parallel **char**, a parallel **short int**, or a parallel **int** bit-field, or their signed or unsigned varieties, or a parallel enumeration type, may be used in an expression wherever a parallel **int** or parallel **unsigned int** may be used. If a parallel **int** can represent all values of the original type, the value is converted to a parallel **int**; otherwise, it is converted to a parallel **unsigned int**. All other parallel arithmetic types are unchanged by the parallel integral promotions. The parallel integral promotions preserve value, including sign. [As we will see in §14, both the integral and parallel integral promotions are also extended to include **bools**.]

Similarly, the usual arithmetic conversions are extended to include parallel types. After the scalar-to-parallel promotion detailed above has occurred, a set of new parallel arithmetic conversions is applied. These are identical to the usual arithmetic conversions, but all types are replaced by a parallel version of the same type and of the current shape. All other conversions are similarly extended for parallel types.

5.2.1 Binary Operators

A standard binary C operator may be applied to parallel operands when both operands are of the current shape. For the following list of binary operators, the operations are performed in parallel on corresponding elements.

Multiplicative operators:

*	multiplication	%	remainder
/	division	%%	modulus

Additive operators:

+	addition	−	subtraction

Shift operators:

<<	left shift	>>	right shift

Extremum operators:

<?	minimum	>?	maximum

Relational operators:

<	is less than	>	is greater than
<=	is less than or equal to	>=	is greater than or equal to

Equality operators:

==	is equal to	!=	is not equal to

Bitwise AND operator:

&	bitwise AND

Bitwise XOR operator:

^	bitwise XOR

Bitwise OR operator:

|	bitwise OR

Logical AND operator: *Causes contextualization of RHS just as scalar* && *causes*

&& conditional AND *conditionalization [see below and §6.2]*

Logical OR operator: *Causes contextualization of RHS just as scalar* || *causes*

|| conditional OR *conditionalization [see below and §6.2]*

Assignment expressions:

Simple assignment operators:

=

Compound assignment operators:

+=	<<=	<?=
-=	>>=	>?=
*=	&=	
/=	^=	
%=	|=	

Sequential expression:

, evaluate LHS then RHS expressions; the result is the value of the RHS

The short-circuit operators, **&&** and ||, in C cause their RHS to be evaluated only if the result of evaluating their LHS requires it. That is, for **&&**, the RHS is evaluated if and only if the LHS is non-zero. For ||, the RHS is evaluated if and only if the LHS is zero. When both the LHS and RHS are scalar, this normal Standard C behavior results.

This normal C behavior is extended to allow the LHS or the RHS or both to be parallel. Any parallel expressions must be of the current shape, and if one expression is scalar and the other parallel, the scalar is promoted to parallel of the current shape by replication. If either or both operands are parallel, a parallel overloaded version of the operator applies. In this case, the RHS is evaluated under the context imposed by the LHS [see §6.2 for a description of context and contextualization]. That is, the context for the RHS is narrowed to be active only where the LHS is true (non-zero) for parallel `&&` or false (zero) for parallel `||`.

5.2.2 Ternary Operator

The standard ternary C operator may be applied to parallel operands:

Conditional expression operator:

condition-expression ? *true-expression* : *false-expression*

Causes contextualization of true- and false-expressions

just as scalar ? : *causes conditionalization [see §6.2]*

When the *condition-expression* is scalar, the normal Standard C behavior results. That is, either the *true-expression* or the *false-expression* is evaluated depending on the value of the *condition-expression*. This usual C behavior is extended to allow the *true-* and *false-expressions* to be parallel (even when the *condition-expression* is scalar). Any parallel expressions must be of the current shape. If either the *true-expression* or the *false-expression* is parallel and the other is scalar, the scalar is promoted to parallel of the current shape by replication.

The conditional expression operator has an overloaded meaning when the *condition-expression* is parallel. In this case, it behaves like the **where** statement [See §6.2 Contextualization Statement] but *returns* a parallel result. Both the *true-* and *false-expressions* are promoted to be parallel of the current shape, and both the *true-expression* and *false-expression* are always evaluated (even if the *condition-expression* is either true or false in all positions). The *true-* and *false-expressions* are evaluated under the context imposed by the *condition-expression* [see §6.2 for a description of context and contextualization]. That is, the context for the *true-expression* is narrowed to be active only where the *condition-expression* is true (non-zero), and the context for the *false-expression* is narrowed to be active only where the *condition-expression* is false (zero).

5.2.3 Postfix Operators

A standard postfix C operator may be applied to parallel operands as well. For the postfix operators:

Subscripting: *[Described in detail in §8 on indexing]*

`[]` subscripting

Component selection:		*[Described in detail in §8 on indexing]*
.	component selection	
->	dereferencing and component selection	
Function calls:		*[Described in detail in §11 on functions]*
( )	call function	*There are no parallel functions; however, functions can take parallel arguments and return parallel results*
Postincrement operator:		
++	increment	
Postdecrement operator:		
--	decrement	

5.2.4 Unary Operators

A standard unary C operator may be applied to parallel operands as well. For the unary operators:

Sizeof operator:		*[Described in detail above in §3.7 on Storage Size Differences]*
`sizeof`	operator to return size	
Unary minus:		
-	integer or floating-point negative	
Unary plus:		
+	integer or floating-point identity	
Logical negation:		
!	non-zero becomes zero; zero becomes one	
Bitwise negation:		
~	bitwise ones-complement	
Address operator:		*[Described in detail in §12 on pointers]*
&	address of operand	
Indirection:		*[Described in detail in §12 on pointers]*
*	dereference operand	
Preincrement operator:		
++	increment	
Predecrement operator:		
--	decrement	

5.2.5 Cast Operator

The standard C cast operator may be applied to parallel operands as well:

Cast expression: *[Described in detail in §10 on casting]*

( )	type conversion

5.3 Parallel-to-Scalar Reduction Assignment Operators

When the assignment operators (except for the remainder and shift operators) are used with a scalar LHS and a parallel RHS, the operator performs a reduction. That is, it performs the specified operation on the parallel RHS to convert it to a scalar value, which is then combined with the scalar LHS. Only active positions of the RHS participate in the reduction [see §6.2]. If there are no active positions, the LHS is unchanged. The reduction operators are:

+=	sum	&=	bitwise AND	<?=	minimum
-=	negative of the sum	^=	bitwise XOR	>?=	maximum
*=	product	|=	bitwise OR		
/=	reciprocal of the product				

```
sil -= bil;   is equivalent to   sil -= += bil;     [See §5.4]
sil /= bil;   is equivalent to   sil /= *= bil;     [See §5.4]
```

The programmer should be aware that these reduction assignment operators take their LHS as one of the operands, just like the standard C assignment operators. Thus,

```
    sil += bil;
```

sums the ten elements of **bil** and the value of **sil** and stores the sum in **sil**.

The assignment operators =, %=, <<=, and >>= cannot be used with a scalar LHS and a parallel RHS. Any of these constructs causes a compile-time error.

An arbitrary representative of a parallel value may be chosen by casting a parallel value to be scalar. That is, when a parallel value is cast to a scalar type, one of the elements of the parallel value is selected and returned as the result. If no positions of the parallel value are active, the behavior is undefined. [See §10 for all uses of casting.]

5.4 Unary Use of the Reduction Assignment Operators

All of the reduction operators defined above are available as unary operators when applied to a parallel operand. When the reduction assignment operators are used in a unary sense with a parallel operand, they simply perform the reductions specified above and return the scalar reduced result. As always, only active positions of the RHS participate in the reduction [see §6.2]. Unary +=, −=, *=, and /= require their operand to be of arithmetic type. Unary **&=**, **^=**, and **|=** require their operand to be of integral type. Unary **<?=** and **>?=** require their operand to be of Standard-C-scalar type. All the unary assignment operators have the same precedence as the "unary operators" in Standard C. The integral promotions are performed on their operands, and their results have the promoted type. Use of these unary operators with scalar operands is a compile-time error. Some examples of unary use of reduction assignment operators follow:

```
    if((+= bi2) > 37) printf("Sum is greater than 37\n");
```

will print "Sum is greater than 37" if the sum of the **bi2**'s is greater than 37. And

```
    if(|= bi2) printf("At least one bit is on\n");
```

will print "At least one bit is on" if at least one bit in any of the **bi2**'s is set.

5.5 Parallel-to-Parallel Reduction Assignment Operators

All of the reduction operators defined above are available as binary reduction operators when used with a parallel LHS and a parallel RHS when the parallel LHS has collisions (more than one lvalue is the same). Collisions may occur when the LHS is parallel left indexed [§8.2 on parallel left indexing explains how collisions could occur]. When used in this way, the values on the RHS with the same destination are all combined into the LHS location by performing the specified operation. That is, these reduction operators perform the operation which, in Paris, is known as send-with-*type-operation* (e.g., send-with-f-add). An example of how to use assignment operators in this way is given in §8.4 on indexing.

When = is used with both a parallel LHS and RHS (which must be of the current shape), any possible collisions are allowed. Once again, collisions may occur when the LHS is parallel left indexed [see §8.2]. When collisions do occur with = as the assignment operator, one of the colliding data elements is arbitrarily chosen to be stored into the destination. It should be noted that although the choice is arbitrary, it must be reproducible on the same hardware in the same configuration. [In the Paris implementation of C*, storing into a parallel-left-indexed parallel variable uses send operations that allow overwriting].

6 Conditionalization and Contextualization Statements

6.1 Conditionalization Statement

There is no special conditional statement in C*. The standard C **if** statement functions as expected in C*. It may take only a scalar expression as its condition.

6.2 Contextualization Statement

where(*where-expression*) *then-body*

where(*where-expression*) *then-body* else *else-body*

The **where** statement is involved with setting the context, a process known as contextualization. The context is a parallel boolean mask (i.e., each element of it is true or false) that controls the execution of parallel operations position by position. A different context is associated with each shape object, and the context associated with the current shape is always applied to operators. If an element of the current context is true, parallel operations on elements in the corresponding position take place — these are active positions; if an element of the context is false, parallel operations on elements in the corresponding position do not take place — these are inactive positions. Scalar operations are not affected by context.

The contextualization statement must be invoked with a parallel *where-expression*, which must be of the current shape — it is a compile-time error to attempt to use **where** (a new reserved word) with a scalar expression. The **where** statement causes contextualization of the positions of that shape for the duration of the *then-body* and *else-body*. If AP_{prior} is the set of active positions in the current shape immediately prior to execution of the **where** statement, then parallel code in the *then-body* is executed only in those positions in the intersection of AP_{prior} and those positions in which the *where-expression* is true (or non-zero). That is, if C_{prior} is the context for the current shape immediately prior to execution of the **where** statement, then for the duration of the *then-body*, the context is narrowed to be the logical-AND of C_{prior} and (*where-expression* != 0). If there is an *else-body*, then parallel code in it is executed only in those positions in the intersection of AP_{prior} and those positions in which the *where-expression* is false (or zero). That is, if there is an *else-body*, then for the duration of the *else-body*, the context is narrowed to be the logical-AND of C_{prior} and (*where-expression* == 0). All scalar code in both the *then-body* and the *else-body* is always evaluated. After the **where** statement has completed, the set of active positions is restored to its state prior to executing the **where** statement. For example, with the following declarations,

```
shape [10]Sb;
int:Sb bi1, bi2, bi3, bi4;
int si1;
```

and if the data has the following values,

Shape Sb:	Position 0	Position 1	Position 2	...
bi1:	2	4	3	...
bi2:	0	0	0	...
bi3:	1	5	32	...
bi4:	0	0	0	...
Scalars:				
si1:	12			

and then the following program is executed with shape **Sb** current and all positions initially active,

```
        where(bi1 >= 3) {
                bi2 = bi3;
                si1++;
        }
        bi4 = bi3;
```

the data will have the following values after execution,

Shape Sb:	Position 0	Position 1	Position 2	...
bi1:	2	4	3	...
bi2:	0	5	32	...
bi3:	1	5	32	...
bi4:	1	5	32	...
Scalars:				
si1:	13			

Note that the assignment of **bi3** to **bi2** has occurred only in the active positions, that the scalar variable **si1** is incremented once, but that the assignment of **bi3** to **bi4** has occurred in all positions because the context after the **where** statement body has completed reverts back to its state prior to the **where** statement.

The parallel **where** statement contextualized the positions of the shape, making position 0 inactive. This then affects the body of the contextualized statement in a dynamically bound way. That is, the contextualization remains in effect for the duration of the statement and for all procedures that it may call. (Note: there is a mechanism, called **everywhere**, to turn on all positions of a shape in a nested context. [See §6.4]) It is mandatory that the parallel expression in the **where** statement follow all of the usual rules for an expression. For example, the following **where** statement is not acceptable because it tries to use a parallel variable in other than the current shape:

```
    where((bi1 >= 3) && (ci1 <= 1)) S;
```

[Note: The previous example would be illegal even if a selection statement were not in C* because it attempts to combine values in two different shapes with **&&**.]

The *then-body* and *else-body* that follow the *where-expression* may contain statements with parallel expressions only of the current shape. It is possible for them to have expressions in a different shape by nesting a shape selection statement within the *then-body* or *else-body* — expressions of another shape are not affected by the contextualization of the shape of the *where-expression*. Scalar code within the statements is also not conditionalized. Even if no positions of the shape are made active by the contextualization, the *then-body* or *else-body* is still executed. For example, with the following declarations,

```
shape [10]Sb, [50][30]Sc;
int:Sb bi1, bi2, bi3;
int:Sc ci1, ci2;
int si1;
```

and the following data:

Shape Sb:	Position 0	Position 1	Position 2	...
bi1:	2	4	3	0...
bi2:	0	0	0	0...
bi3:	1	5	32	54...
Shape Sc:	Position 0,0	Position 0,1	Position 0,2	...
ci1:	0	0	0	
ci2:	34	42	7	
Scalars:				
si1:	0			

then the following program is executed with shape **Sb** current and all positions (of both **Sb** and **Sc**) initially active:

```
where(bi1 > 4) {
     bi2 = bi3;
     si1 = 4;
     with(Sc)
          ci1 = ci2;
}
```

The data will be changed as follows after execution, even though no positions of shape **Sb** were left active by the *where-expression* (**bi1 > 4**):

Shape Sb:	Position 0	Position 1	Position 2	...
bi1:	2	4	3	0...
bi2:	0	0	0	0...
bi3:	1	5	32	54...
Shape Sc:	Position 0,0	Position 0,1	Position 0,2	...
ci1:	34	42	7	
ci2:	34	42	7	
Scalars:				
si1:	4			

Note that the assignment of **4** to **si1** has occurred even though no positions of **Sb** were active. An optimizer for C* programs might check to see if the *then-body* or *else-body* contained any scalar code. If it determined that there was no scalar code (including no calls to functions that might contain parallel code), it might not execute the particular *body* at all if no positions were active. This is an example of "as if" behavior — that is, an optimizer is free to change the actual behavior of code it produces so long as the effect of that code is *as if* it did exactly what the specification requires it to do. This is the basic license given to all code optimizers.

To keep scalar code and code in other shapes from executing if no positions of the current shape are left active, use a reduction operator to evaluate the condition to a scalar, and then use an **if** statement to conditionalize code, as follows:

```
if(|= (parallel_condition != 0))
    where(parallel_condition)
        then_statement;
if(|= !parallel_condition)
    where(!parallel_condition)
        else_statement;
```

except if the *parallel_condition* has side effects. The *parallel_condition* should be evaluated only once to avoid multiple side effects. For example:

```
{
    int:current parallel_temp;
    if(|= ((parallel_temp = parallel_condition) != 0))
        where(parallel_temp)
            then_statement;
    if(|= !parallel_temp)
        where(!parallel_temp)
            else_statement;
}
```

If the *parallel_condition* is known to be 0- or 1-valued, the not-equal comparison to 0 is not needed (the following example illustrates such a case). Alternatively, if the not-equal comparison to 0 is needed, the *parallel_temp* could be declared of **bool** type [see §14]. Our specific example from above can be reformed to execute its *then-body* only if some positions are active, as follows:

```
{
        int:Sb ptemp;
        if(|= ptemp = (bi1 > 4))
                where(ptemp) {
                        bi2 = bi3;
                        si1 = 4;
                        with(Sc)
                                ci1 = ci2;
                }
}
```

It is often desirable to allow iteration over parallel variables so that the iteration count varies from position to position. This is referred to as *per-position iteration*. The following idiom is useful in these cases:

```
while(|= (parallel_condition != 0))
        where(parallel_condition)
                statement;
```

It is expected that the execution of the *statement* will eventually decrease the positions in which the *parallel_condition* is true. Thus, the *statement* is repeatedly executed with a gradually diminishing set of active positions. When no more positions remain active, the **while** loop will terminate. As above, the *parallel_condition* should be evaluated only once to avoid multiple side effects, as follows:

```
{
        int:current parallel_temp;
        while(|= ((parallel_temp = parallel_condition) != 0))
                where(parallel_temp)
                        statement;
}
```

Also as above, if the *parallel_condition* is known to be 0- or 1-valued, the not-equal comparison to 0 is not needed. Note that this technique may be used with the other iteration statements in C — **do-while** and **for**) as well. Here is an example of this technique in a program fragment that computes 2^{count} in each position:

```
        shape [10]Sb;
        int:Sb count, prod;
        with(Sb) {
             /* Initialize each element of "count" here */
             prod = 1;
             while(|= (count>0))
                  where(count>0) {
                       count--;
                       prod *= 2;
                  }
        }
```

It is possible to cause more than one shape to be contextualized by nesting **where** statements, as follows:

```
        where(bi1 >= 3)
             with(Sc)
                  where(ci1 <= 1)
                       S;
```

Any code in *S* (or called from *S*) that is of shape **Sb** is contextualized by the **(bi1 >= 3)** expression, while any code in *S* that is of shape **Sc** is contextualized by the **(ci1 <= 1)** expression.

The *then-body* always appears to be executed before the *else-body*. The compiler is free to perform dependency analysis to prove that there are no possible dependencies between the two bodies and run them in any order or concurrently.

The **where** statement provides an efficient mechanism to guarantee that the compiler may generate code that simply contextualizes a shape without needing to perform a global reduction (global-logior) to determine if there are any active positions left. Of course, the compiler is free to actually generate a global-logior and branch around the code if the compiler determines that the body contains only parallel code in the current shape and no function calls. It would do this if there were a sufficiently large number of lines of such code.

6.2.1 Execution with No Active Positions

The **where** statement allows execution of statements when there are no active positions of the current shape. This implies that the following statement may be executed:

```
        where(bi1 > 4)
             si1 = (+= bi1);
```

This statement assigns to scalar variable **sil** the sum of the elements in active positions of **bil**. What happens if the **where** statement leaves no active positions of shape **Sb**? The assignment to **sil** should occur in any case because it is a scalar. C* deals with this situation by defining a set of values that are returned by the unary reduction operators when there are no active positions. These values are the identities for the operator (when one exists). Here is the table of reduction values when there are no active positions:

Unary Reduction Operator	Value
`+=`	0
`-=`	0
`*=`	1
`/=`	1
`&=`	~0 (all one bits)
`^=`	0
`|=`	0
`<?=`	maximum value representable
`>?=`	minimum value representable
(*scalar*) [parallel value cast to a scalar type]	undefined

When one of the reduction operators is used in a binary context, the LHS is left unchanged if there are no active positions of the expression's shape. (This is the natural consequence even if one considers that the operators return the values specified in the table above, which then operate with the LHS — except for the cast operator.) The **where** statement behavior implies that operators (i.e., code generated for all operators) in all parts of a program where the contextualization is not known at compile time (such as externally visible functions) must be able to deal with the cases where there are no active positions of shapes.

The **where** statement can only further constrain the set of active positions of a shape — it can never enlarge that set. It is precisely for the purpose of enlarging the active set of positions that the **everywhere** statement [see §6.4] exists.

6.3 The switch Statement

The **switch** statement may be used only with a scalar-valued expression, and it behaves in the same way that it does in C. Please note that the **switch** statement can cause a branch into a block.

C* does not define the behavior of branching into a nested **with**, **where**, or **everywhere** body or branching into a block containing a parallel variable declaration or a shape declaration.

6.4 The everywhere Statement

The grammar for the **everywhere** statement is:

everywhere *activated-statement*

The **everywhere** statement activates all positions of the current shape for the duration of the *activated-statement*. **everywhere** is a new reserved word. After the *activated-statement*, the set of active positions of the current shape is restored to its state immediately before the **everywhere** statement. Note that, as mentioned in §4, all positions of all shapes are activated when a C* program begins execution.

Because all parallel code in the *activated-statement* is executed in all positions (i.e., the context is made active everywhere), it is equivalent to say that it is executed without performing contextualization. An implementation is free to take advantage of this characteristic by producing code that doesn't check to see if a position is active or not. Such code will often be more efficient than code that may possibly be contextualized. In Paris, these instructions are known as **always** instructions and they execute about 30% faster than non-**always** instructions.

An optimizer for the C* language may also choose to generate instructions that are independent of context whenever it generates code whose result is stored into a compiler-generated temporary, so long as the operation could not produce an error condition in what would otherwise be inactive positions (e.g., division by zero).

7 Iteration Statements

All of the iteration statements, **while**, **do**, and **for**, behave exactly as they do in C. This implies that all expressions in these statements must be scalar — with the exception of the *initial-expr* and the *increment-expr* in the **for** statement, which may be parallel. The only interesting cases come up in the behavior of **break**, **continue**, **goto**, and **return**. If the **break**, **continue**, **goto**, or **return** statement is used to leave the nested context of a **with** statement to branch to an outer level, the shape selection at the destination is once again made current. If the **break**, **continue**, **goto**, or **return** statement is used to leave the nested context of a **where** or **everywhere** statement to branch to an outer level, the contextualization at the destination is once again made active. Note that this context may involve resetting the contexts of several shapes. C* does not define the behavior of branching into a nested **with**, **where**, or **everywhere** body or branching into a block containing a parallel variable declaration or a shape declaration. This behavior is determined by the implementation. Note that **goto** and **switch** can cause branches into blocks.

If a **return** statement is executed while in a contextualized shape (i.e., inside a **where** or **everywhere**), the return values are defined only in the active positions. The other return values are determined by the implementation (**everywhere** may be used to initialize all values). However, all values active in the caller's context will be visible after the return. This is simply a manifestation of seeing more positions active after being assigned in a constrained context.

8 Position Indexing of Parallel Expressions

Parallel expressions may have their positions indexed by a set of subscripts to the left of the parallel expression. We will refer to this type of an index as a *left index*. If a parallel expression is left indexed, it must be indexed with an appropriate number of subscripts so that they conform to the parallel expression's rank. The left indices must be integral expressions, and integral promotions are performed on them. As will be shown later in this section, the left indices may be either scalar or parallel. The precedence of left indexing is presented in §3.1.

If a shape is not fully specified but has its rank specified at compile time, as is the case for shapes **Se** and **Sf** [declared in §3.1], then a compile-time check can be made to insure that the appropriate number of subscripts are present. If the rank of a shape is not specified at compile time, as is the case for shape **Sa**, then a run-time check may be generated (depending on safety level) to insure that the rank is correct. This implies that the rank of a shape must be determined before expressions of that shape can be left indexed. In fact, the shape must be fully specified before parallel variables of that shape may be allocated.

Every left index must be in range for that dimension or undefined behavior may occur. Run-time safety may be set to check for left indices that are out of range.

8.1 Scalar Left Indices

A parallel expression may be left indexed by a list of scalar expressions to select an individual position of that parallel expression. So, for example, given the declarations:

```
    shape Sa, [10]Sb, [50][30]Sc;
    int ai1:Sa, ci1:Sc;
    float:Sb bf2;
    int si1, si2;
```

the following are scalar-valued expressions:

```
    [si1]ai1          if Sa is of rank one
    [si1][si2]ai1     if Sa is of rank two
```

```
[2]bf2
[10][si1]ci1
```

When a scalar-left indexed expression is used as an rvalue, the value of one element is fetched from the specified position. When a scalar-left indexed expression is used as an lvalue, it denotes a single position of that expression. If it is used as the left-hand-side of an assignment, a scalar value is stored into the specified position.

Because parentheses are used to convey information about the rank of an expression being left indexed, they may not appear within a contiguous list of scalar left indices (e.g., **[10]([si1]ci1)** is *not* valid). Parentheses surrounding a left indexed expression terminate the parsing of a complete left indexed expression.

8.2 Parallel Left Indices

It is also possible to specify parallel left indices for a parallel expression. If this is done, all parallel left indices must be of the current shape [see §4]. If some subscripts are scalar but at least one other is parallel, the scalar subscripts are promoted to be parallel of the current shape. The result of parallel left indexing a parallel expression of shape **S1** with subscripts of the same or another shape, **S2**, is a mapping from **S1** to **S2** — the result is the expression being left indexed mapped into the shape of the left indices. For example,

```
shape [10]Sb, [50][30]Sc, [30][50]Sd;
int:Sb bi1, bi2, bi3;
float:Sb bf2;
int:Sc ci1, ci2, ci3;
int:Sd di1, di2;
[ci1]bf2         is a parallel expression of shape Sc
[bi1]bf2         is still a parallel expression of shape
                 Sb, but its elements may be reordered in
                 any possible combination
[di1][di2]ci2    is a parallel expression of shape Sd
```

The indices indicate which elements of the source shape should be selected in the mapping to the destination shape. So, in the first example above, if it were used as an rvalue, it would be as if a temporary parallel floating-point variable in shape **Sc** were being created as follows:

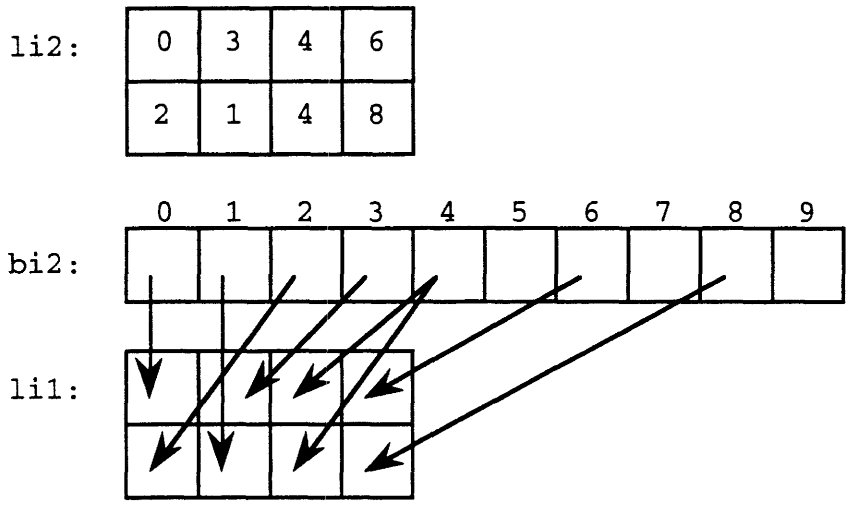
```
float:Sc temp;
for (0 <= index0 < 30) and (0 <= index1 < 50) do
   [index0][index1]temp = [[index0][index1]ci1]bf2;
```

If a parallel-left-indexed parallel expression appears as an rvalue, it would imply that a *get* would be needed to retrieve the value from another shape. If a parallel-left-indexed parallel expression appears as an lvalue, it would imply that a *send* would be needed to store the value in a variable of another shape.

For the program fragment,

```
shape [10]Sb, [2][4]Sl;
int:Sb bi2;
int:Sl li1, li2;
with(Sl)
   li1 = [li2]bi2;
```

each element of parallel variable **li2** (of shape **Sl**) will serve as an left index to select an element of **bi2** (of shape **Sb**). The **bi2** retrieved will be stored into the corresponding position of **li1**. Thus, each **li2** (of shape **Sl**) is used as a position left index of **bi2** to get an element of **bi2** (from shape **Sb**) and store it into its **li1**. Note that it is possible for different **li1**'s to *get* from the same **bi2** element (this will happen if they both have the same value in **li2**). Here is a schematic representation of this operation with some specific data for **li2**:

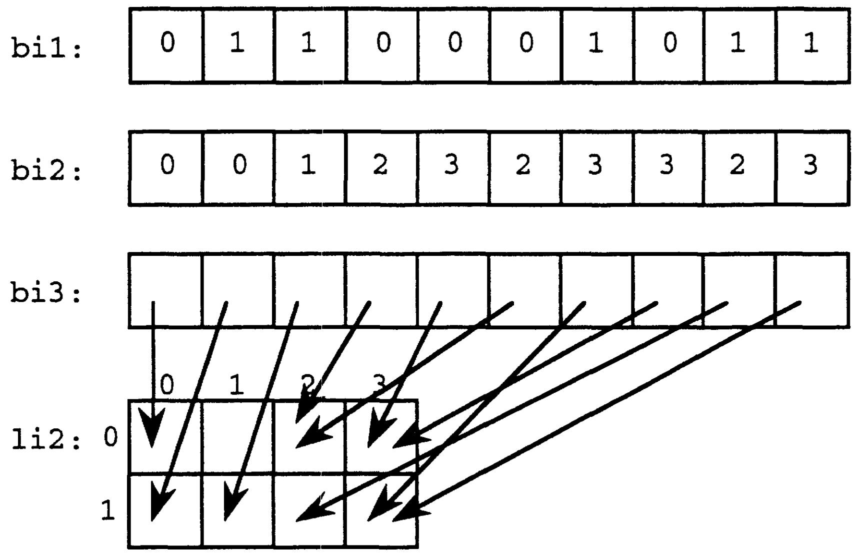


Note that in this example, element 4 of **bi2** was received by both elements [0][2] and [1][2] of **li1**, but that elements 5, 7, and 9 of **bi2** were not received by any elements of **li1**.

For the program fragment,

```
shape [10]Sb, [2][4]Sl;
int:Sb bi1, bi2, bi3;
int:Sl li2;
with(Sb)
   [bi1][bi2]li2 = bi3;
```

corresponding elements of parallel variables **bi1** and **bi2** (of shape **Sb**) will serve as a left index to select an element of **li2** (of shape **Sd**), where the corresponding element of variable **bi3** will be stored. Note that if the pair of **bi1** and **bi2** is the same for different **bi3**s, then a collision will occur — this is an attempt to store more than one value into the same location. When a collision occurs, one of the data elements is chosen arbitrarily. Although the choice is arbitrary, it must be reproducible on the same hardware in the same configuration. Here is a schematic representation of this operation with some specific data for **bi1** and **bi2**,



Note that in this example, there are three collisions that occur: elements 3 and 5 of **bi3** are both sent to element [0][2] of **li2**; elements 4 and 7 of **bi3** are both sent to element [0][3] of **li2**; and elements 6 and 9 of **bi3** are both sent to element [1][3] of **li2**. In each of these cases, one of the elements would be arbitrarily chosen. Also note that no datum was sent to element [0][1] of **li2**.

Because parentheses are used to convey information about the rank of an expression being left indexed, they must not be used to separate the contiguous indices used as a left index for a single expression. That is, for left indexing, no parentheses may separate the set of left indices whose number is equal to the rank of the expression being left indexed. In addition, if an expression is being left indexed more than once, parentheses must be used to separate each set of left indices. For example, in

```
[ci2][ci3]([ci1]bf2)
```

the parallel variable **bf2** (of shape **Sb**) first is indexed by **ci1** (of shape **Sc**), then is indexed again — this time by **ci2** and **ci3** (both of shape **Sc**).

A parallel-left-indexed parallel expression may not be the operand of the **&** operator. This restriction guarantees that a parallel pointer handle cannot be created as a result of the **&** operator. If the C* language is later extended to allow parallel pointer handles, this restriction may be lifted. A parallel-left-indexed parallel expression is a modifiable lvalue if and only if the expression being indexed is a modifiable lvalue. In addition, a parallel-left-indexed parallel expression is an lvalue if and only if the expression being indexed is an lvalue.

A scalar-left-indexed parallel expression also may not be the operand of the **&** operator. This restriction guarantees that a pointer to a single element cannot be created as a result of the **&** operator. If the C* language is later extended to allow pointers to single elements of parallel variables, this restriction may be lifted.

8.3 Left-indexed Expressions

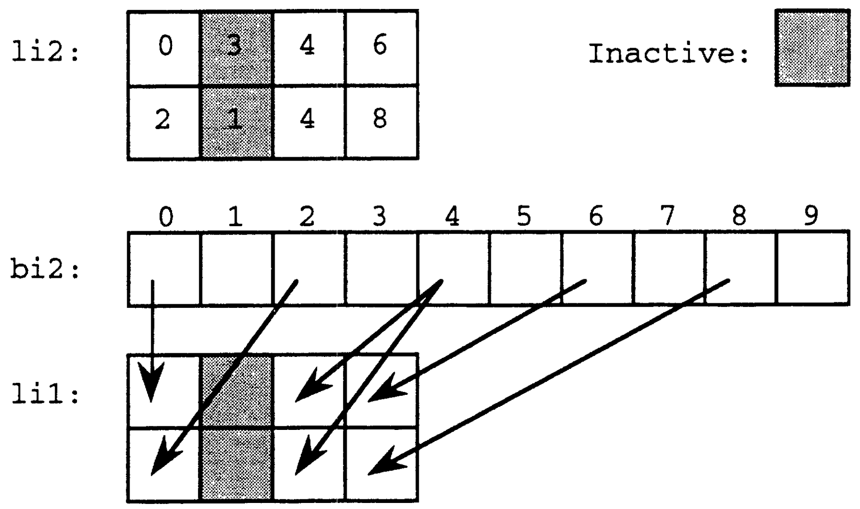
In addition to being able to left index parallel variables, certain parallel expressions may also be left indexed. Constraints on which expressions may be left indexed are implied because the language requires that almost all operators are evaluated on operands of the current shape [see §4]. So, when *no* shape change is performed by the left indexing (i.e., the indexed expression is of the current shape), any expression may be left indexed, subject to the constraint that if the left-indexed expression is used where a modifiable lvalue is required (such as the left-hand-side of an assignment), only a modifiable lvalue may be left indexed. If the left index operation is changing the shape of the indexed expression, then the only valid expressions are: parallel identifiers (including entire parallel **structs** and **unions**), a selected field of a parallel **struct** or **union**, a dereferenced scalar pointer to a parallel variable, and another left-indexed expression. This is the list of operations that may be performed on operands that are not of the current shape.

8.4 How Parallel Left Indexing Is Affected by Context

A parallel-left-indexed operation is performed only as directed by active positions. When parallel left indexing appears in an *rvalue*, context affects which elements are *received*. When parallel left indexing appears on the *LHS of an assignment*, context affects which elements are *sent*. Here are the examples from above with the inclusion of contextualization:

```
shape [10]Sb, [2][4]Sl;
int:Sb bi2;
int:Sl li1, li2;
with(Sl)
   where(!(li2 % 2))
      li1 = [li2]bi2;
```

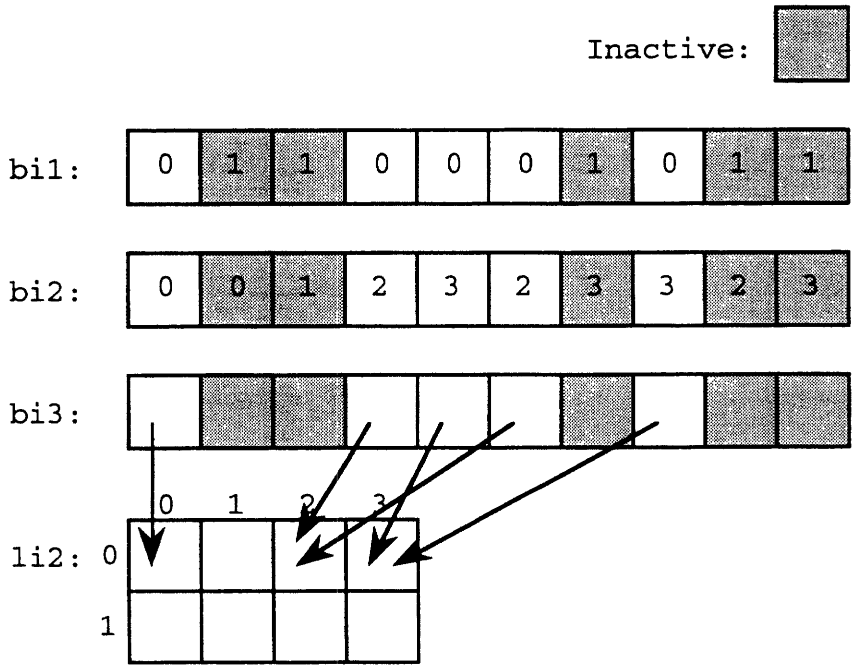
Here is a schematic representation of the operation of this program fragment with the same data for **li2**:



Notice that the context has affected which positions received data because the operation performed was a get.

```
shape [10]Sb, [2][4]Sl;
int:Sb bi1, bi2, bi3;
int:Sl li2;
with(Sb)
   where(!bi1)
      [bi1][bi2]li2 = bi3;
```

Here is a schematic representation of the operation of this program fragment with the same data for **bi1** and **bi2**:



Notice that the context has affected which positions sent data because the operation performed was a send.

§8.3 discusses how an expression rather than a simple identifier may be left indexed. The question arises as to the context under which the *left-indexed expression* is evaluated. (Remember that C* requires that, in almost all cases, parallel variables must be of the current shape. The exceptions are in §4. Following these rules, when left indexing an expression that is not simply an identifier or the selection of a **struct** or **union** member, the expression being left indexed must be of the current shape.) When the shape of the indexed expression does not have all positions active, should that expression be evaluated under the context of its shape, or should it be evaluated in those positions that are being requested by the values of the left index expression. In the former case, it is possible that an inactive position of the left-indexed expression's shape would be read into an active position of the current shape — this position would have an undefined value. In the latter case, the context of the current shape would be sent to the left-indexed expression and this expression would be evaluated in all positions whose values were requested — no requested positions could have undefined values because they are inactive.

The C* language uses the former semantics. This solution enables more efficient code to be generated because an additional communication operation (to send the context) is not required. The semantics of the latter case may still be implemented by explicitly sending the current context and recontextualizing.

8.5 Parallel-to-Parallel Reduction Assignment Operators Revisited

The reduction operators detailed in §5.3 may be used in conjunction with a left-indexed left-hand-side. In these cases, the reductions are performed with results accumulated in multiple elements of a parallel variable. For example:

```
shape [8]Sj, [10]Sk;
int:Sj ji1 = 42;
int:Sk ki1, ki2;
with(Sk)
    [ki2]ji1 += ki1;
```

Starting with the following values after initialization, but before execution:

Shape Sj positions:	0	1	2	3	4	5	6	7		
ji1:	42	42	42	42	42	42	42	42		
Shape Sk positions:	0	1	2	3	4	5	6	7	8	9
ki1:	34	1	4	7	3	2	1	1	2	5
ki2:	0	4	2	3	4	4	1	5	7	5

yields the following results after execution:

Shape Sj positions:	0	1	2	3	4	5	6	7		
ji1:	76	43	46	49	48	48	42	44		
Shape Sk positions:	0	1	2	3	4	5	6	7	8	9
ki1:	34	1	4	7	3	2	1	1	2	5
ki2:	0	4	2	3	4	4	1	5	7	5

As discussed earlier, if the simple assignment operator is used in conjunction with a left-indexed left-hand-side, collisions are resolved arbitrarily. For example:

```
shape [8]Sj, [10]Sk;
int:Sj ji1 = 42;
int:Sk ki1, ki2;
with(Sk)
     [ki2]ji1 = ki1;
```

Starting with the following values after initialization, but before execution:

Shape Sj positions:	0	1	2	3	4	5	6	7		
ji1:	42	42	42	42	42	42	42	42		
Shape Sk positions:	0	1	2	3	4	5	6	7	8	9
ki1:	34	1	4	7	3	2	1	1	2	5
ki2:	0	4	2	3	4	4	1	5	7	5

yields the following results after execution:

Shape Sj positions:	0	1	2	3	4	5	6	7		
ji1:	34	1	4	7	†	‡	42	2		
Shape Sk positions:	0	1	2	3	4	5	6	7	8	9
ki1:	34	1	4	7	3	2	1	1	2	5
ki2:	0	4	2	3	4	4	1	5	7	5

† Element 4 of **ji1** gets assigned either 1 (from element 1 of **ki1**), 3 (from element 4 of **ki1**), or 2 (from element 5 of **ki1**), selected arbitrarily.

‡ Element 5 of **ji1** gets assigned either 1 (from element 7 of **ki1**) or 5 (from element 9 of **ki1**), selected arbitrarily.

8.6 The pcoord Function and Grid Communication

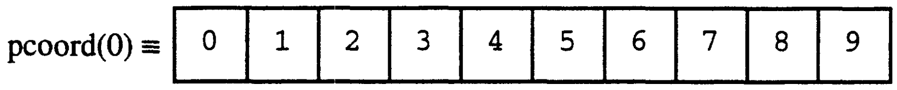
The **pcoord** intrinsic function is a parallel axis-coordinate value constructor. Its declaration is:

int:current pcoord(int *axis*)

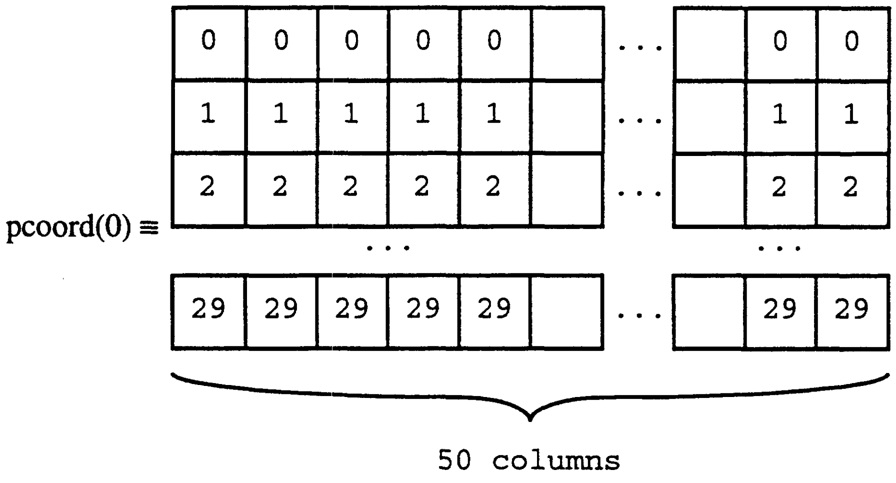
pcoord is called with an axis argument and returns a parallel value in the current shape in which each position is initialized to its coordinate along the specified axis. It is an error to specify an axis number that is greater than or equal to the rank of the shape.

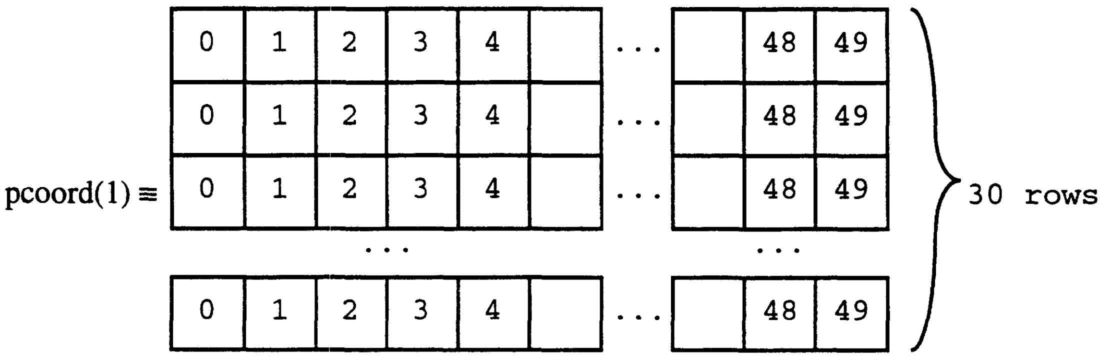
```
    shape Sa, [10]Sb, [30][50]Sd;
```

Here is an example of using **pcoord** with shape **Sb** current:



Here are examples of using **pcoord** with shape **Sd** current:





Besides being useful in its own right by providing a way to create an index array, **pcoord** is also used for n-dimensional NEWS , or grid, communication. When **pcoord** calls are used as left subscripts of a parallel value in a small number of well-defined ways (see below), NEWS communication primitives are generated. These operations are much faster than general communications but require that each datum transferred follow the same relative path from source to its destination. For example, with shape **Sb** current,

```
    bi1 = [pcoord(0)+1]bi2;
```

causes the generation of instructions that execute a get-from-right. As a shorthand for writing **pcoord**(*axis-where-this-appears*) when **pcoord** appears within a left index, one can write the period symbol instead. Therefore, the following statement is equivalent to the one above:

```
    bi1 = [.+1]bi2;
```

It is important that the programmer sufficiently contextualize the positions that are active before attempting to perform a NEWS operation, so that it will not attempt to access non-existent positions. That is, all subscripts must be within range — positions in which invalid subscripts would otherwise exist may be disabled through the use of the **where** statement. Therefore, for correct execution of the above statement it might be necessary to encapsulate it within a contextualization as follows:

```
    where (pcoord(0) < (dimof(Sb, 0)-1))
                                    /* turn off rightmost position */
        bi1 = [.+1]bi2;
```

Grid communication may appear on the left-hand-side as follows:

```
    where (pcoord(0) > 0)
                                    /* turn off leftmost position */
        [.-1]bi1 = bi2;
```

Note that the above send-to-left grid operation was appropriately contextualized before its execution.

It is also possible to perform NEWS operations in any number of dimensions of multi-dimensional data at once, as follows:

```
    di1 = [.+1][.-4]di2;
    di1 = [.-12][.]di2;
    di1 = [.][.+17]di2;
```

NEWS operations with wrapping are easily achieved by using the modulus operator as follows:

```
    bi1 = [(.+1) %% dimof(Sb, 0)]bi2;
```

Because the **dimof** intrinsic function returns a **signed int**, it also is possible to use wrapping with a negative NEWS offset as follows:

```
    bi1 = [(.-1) %% dimof(Sb, 0)]bi2;
```

The compiler may generate a NEWS instruction for the left indexing operator if all of the expressions being left indexed are of the current shape and if each of the left index expressions are of one of the following forms:

```
pcoord(this-dim)
pcoord(this-dim) +/- scalar-int-expression
(pcoord(this-dim) +/- scalar-int-expression) %% dimof(shape-
          of-this-parallel-var, this-dim)
(pcoord(this-dim) +/- scalar-int-expression) %% dimof(this-
          parallel-var, this-dim)
.
. +/- scalar-int-expression
(. +/- int-expression) %% dimof(shape-of-this-parallel-var,
          this-dim)
(. +/- int-expression) %% dimof(this-parallel-var, this-dim)
```

Note that simply referring to a parallel variable by name is equivalent to left indexing it with the appropriate number of **pcoord** expressions, each with the axis' self-index as its argument. For example, **bi2** is the same as **[pcoord(0)]bi2**, which is the same as **[.]bi2**. Keep in mind that just as for general, or router, communication, these left-indexed expressions may appear on either the left-hand-side or right-hand-side of assignment operators.

The most efficient communications operation is chosen by the compiler in these **pcoord**-offset left-index cases. For example, if the number of primitive grid communication operations is greater than approximately 15, router communications may be used. The C* compiler automatically generates primitive grid communication operations that can move data in both directions on any axis and that can move data a power-of-two distance in a single operation. Particularly with these instructions, it is almost never necessary to revert to router (general) communication when expressions are of the above forms.

If the *scalar-int-expression* added to any **pcoord** index is not a constant, the compiler cannot determine at compile time the distance to the source or destination. Therefore, in these cases, a run-time routine is called to determine the minimum number of positive/negative nearest-neighbor/power-of-two grid moves necessary to accomplish the operation. Once again, if the number of primitive grid communication operations exceeds some threshold, router communications may be used.

9 Dynamic Shapes and Parallel Variables

If a shape is not fully specified (as for **Sa**, **Se**, and **Sf** in §3.1), the programmer must call:

```
overload allocate_shape;

   /* Create a shape with specified rank and dimensions */

shape allocate_shape(
                shape *sp,
                int rank,
                int dimensions, ...);

        /* Create a shape of specified rank.  An array of the
           dimensions is passed as the second argument */

    shape allocate_shape(
                shape *sp,
                int rank,
                int dimension_array[]);
```

before allocating variables in that shape or selecting the shape through the use of a **with** statement. In the prototype declarations above, two different functions are declared with the same name [see Overloading in §13]. Either of these functions may be called, and the appropriate function is chosen based on the arguments supplied in the call. **allocate_shape** is an intrinsic function in C*. The call to **allocate_shape** modifies the shape object pointed to by its first argument and also returns the same fully specified shape. **allocate_shape** guarantees that the rank passed to it as the second argument is consistent with the rank specified when the shape was declared, if the shape was partially specified. This check (that assignments are performed so as to be consistent with a partially specified rank) will be performed at compile time when possible or will be conditionally emitted at run time, depending upon the safety level.

Several calls to **allocate_shape** are given here as examples:

```
    allocate_shape(&Sa, 1, 20);
    allocate_shape(&Se, 1, 40);
    allocate_shape(&Sf, 2, 5, 20);
    {
         shape Sg = allocate_shape(&Sg, 2, 5, 20);
    }
```

Note that the last use of **allocate_shape**, which might have appeared in a declaration at file scope level, uses the return value from the intrinsic — in fact, it requires that **allocate_shape** return the newly fully specified shape so that it can be assigned in the initialized declaration.

A shape may be assigned (copied) to another shape by using the assignment operator as well. As detailed above, this assignment may occur only if the LHS of the assignment operator is a non-fully-specified shape of appropriate rank (if partially specified). When one shape is assigned to another, the storage duration of the shape specified on the left-hand-side should be the same as the storage duration of the shape specified on the right-hand-side (i.e., they will both share the same shape). This issue does not become a problem unless one examines the possibilities of assigning a fully specified shape with a shorter storage duration to a non-fully-specified shape with a longer storage duration. For example,

```
    shape S;                    /* Unspecified shape S */

    void f(void) {
      shape [1024][512]T;       /* Fully-specified shape T */

      S = T;                    /* S will share T's shape */
      }

    void main() {
      f();
        {
        int:S i;                /* This allocation will fail
                                because S's shape was deallocated
                                when procedure f exited */
        }
      }
```

This illustrates how shape assignment is similar to pointer assignment. It is the user's responsibility to ensure that such behavior is correct.

Shapes may be deallocated through the use of the **deallocate_shape** function — defined in the **<stdlib.h>** header file. It has the following form:

```
        /* Deallocate the shape pointed to by sp and make it be
           either fully unspecified or partially specified, so as
           to be consistent with the pointer's declaration.*/

    void deallocate_shape(shape *sp);
```

A shape thus deallocated should not be used, nor should any copies made of it through shape assignment. In addition, there should be no remaining parallel variables of that shape still allocated (i.e., the behavior is undefined if parallel variables of a deallocated shape still exist). That is, before calling **deallocate_shape** all parallel variables of that shape that were allocated with **palloc** should have been deallocated by calling **pfree**, and all automatic parallel variables of that shape should have been deallocated by leaving the blocks in which they were declared. **deallocate_shape** modifys the shape object to indicate that a shape is no longer associated with that shape object and to maintain a rank constraint if the shape was partially-specified when declared (i.e., only allow shape allocations that are consistent with the declaration).

Shapes that are explicitly allocated by the programmer by calling **allocate_shape** or **allocate_detailed_shape** will not be implicitly deallocated by the compiler. It is the programmer's responsibility to explicitly call **deallocate_shape**. This statement is true even if

allocate_shape or **allocate_detailed_shape** is called in the initializer of a shape. **allocate_shape** and **allocate_detailed_shape** may be called in the initializer of a file-scope shape. These functions are defined to be intrinsics precisely to allow them to be called in this context.

Parallel variables may not be allocated in a non-fully-specified shape. They may be declared as automatics in a nested scope, which is executed after the appropriate call to **allocate_shape** has been made, as follows:

```
shape []S;
main() {
   allocate_shape(&S, 1, 4096);
   /*  Open a block so that auto parallel variables of
       shape S can be declared now that S is fully-
       specified */
   {
      int:S t0, t1;
      with(S) {
         t0 = 23;
         t1 = 76;
         t0 += t1;
      }
   }
   deallocate_shape(&S);
}
```

If a pointer to a parallel variable [see §12] is declared, then a heap-managed parallel variable in a dynamic shape may be allocated by explicitly allocating storage for that variable:

```
int:Sd *p1, *p2;
int:Sa *q1, *q2;
p1 = palloc(Sd, boolsizeof(int:Sd));
q1 = palloc(Sa, boolsizeof(int:Sa));
```

Of course, the shape must be fully-specified before calling **palloc**. Notice that **palloc** may be called for non-dynamic shapes as well. In the example above, shape **Sd** is fully-specified at compile-time. If the call to **palloc** does not succeed, a value equal to **CMC_no_field** — declared in <cscomm.h> — is returned.

The detailed function prototype of **palloc** is as follows:

```
void:void *palloc(shape shape, int size_in_bools);
```

The corresponding function **pfree** is also available to free the storage allocated by palloc. Its argument must be a pointer to a parallel variable previously returned by palloc. The function prototype of pfree follows:

```
     void pfree(void:void *pvar);
```

Both **palloc** and **pfree** are prototyped in the <stdlib.h> header file. Because **palloc/pfree** storage is heap managed, it may be allocated and freed in any order — not necessarily in a last allocated, first freed stack protocol. For the example above, the storage pointed to by **p1** could be freed before the storage pointed to by **q1**.

Once again, the extent of run-time checking is determined by the safety level specified at compile time. At high safety, a check will be made to ensure that a shape has been fully-specified before allocation of a variable of that shape is allowed. At no safety, no such check will be emitted by the compiler.

In addition to **allocate_shape**, another intrinsic function is provided to dynamically allocate shapes. It allows several other characteristics of the shape to be defined in addition to rank and dimensions. The **allocate_shape** intrinsic function may be specific to a particular implementation — the following is the CM-2 version:

```
     overload allocate_detailed_shape;

        /* Create a detailed shape with specified rank and axes */

     shape allocate_detailed_shape(
             shape *sp,
             int rank,
             CM_axis_descriptor_t axes[]);

        /* Create a detailed shape of specified rank.  An array of
           axis descriptors is allocated locally and initialized
           with the variable list of arguments */

     shape allocate_detailed_shape(
             shape *sp,
             int rank,
             unsigned long length,
             unsigned long weight,
             CM_news_order_t ordering,
             unsigned long on_chip_bits,
             unsigned long off_chip_bits, ...);

        /* Fill in the axis descriptor with the list of
           arguments.  This function is used to prepare the
           "axes" argument passed to the first overloading of
           allocate_detailed_shape above. */

     void fill_axis_descriptor(
             CM_axis_descriptor_t axis,
             usigned long length,
                usigned long weight,
                CM_news_order_t ordering,
                unsigned long on_chip_bits,
                unsigned long off_chip_bits);
```

allocate_detailed_shape should be used in all cases where the programmer needs to have finer control over the exact allocation for a shape. For more detail on the CM-2 version, please see the Paris reference manual.

deallocate_shape, **palloc**, and **pfree** are functions provided by the run-time system. The compiler has no special knowledge about these functions.

10 Casts Involving Parallel Types and Values

The cast operator may be used to cast an expression to be in a particular shape. For example, it is possible to cast a scalar expression to a parallel expression in a named shape. This cast is accomplished by replication of the scalar value:

```
        /* Store number of active positions in shape Sc in si1. */
     si1 = += (int:Sc) 1;
        /* Is any position of shape Sc active? */
     si1 = |= (int:Sc) 1;
```

The reader should recognize the programming idioms expressed in these examples. The constant one is cast to a parallel value by replication. Then, in the first example, a sum reduction is performed on that parallel value. The sum reduction takes place in all positions that are active in shape **Sc**. Therefore, the value assigned is the number of active positions in shape **Sc**. In the second example, an inclusive OR reduction is performed on the promoted parallel value. The result is zero if there are no active positions and one otherwise. Therefore, the value assigned is a boolean reflecting whether there are any active positions.

The full type-specifier must be given in the cast type (i.e., **(: Sc)** is *not* a legal cast operation).

Another case where a cast may be used is where there are two expressions that are of the identical shape, but the type system in the compiler cannot ascertain that. In these situations, the programmer may explicitly cast one of the expressions to be of the shape of the other so that they may interact together. There is no movement of data implied in a parallel-to-parallel cast. [See §3.9 for an example of such a parallel-to-parallel cast.]

Another parallel-to-parallel cast is one that does not alter the shape but changes the base type. This kind of cast performs the same conversions that such a scalar-to-scalar cast would cause in Standard C. An example of this kind of cast follows:

```
     overload fcn;
     float:current fcn(float:current);
```

```
int:current fcn(int:current);
shape [10]Sb;
int:Sb bi1;
float:Sb bf1;

with(Sb)
   bf1 = fcn((float:Sb) bi1);
```

In the above program, the cast was used to force the selection of the parallel **float** version of the overloaded function **fcn** rather than the parallel **int** version.

A cast may also be used to arbitrarily select an element of a parallel expression. In this case a parallel expression is cast to a scalar type. [Such a cast is referred to in §5.3.] If no positions of the parallel value are active, the behavior is undefined. Please remember that a parallel expression being cast to a scalar type must be of the current shape. Such a parallel to scalar cast has cost comparable to any other reduction operation. An example of parallel to scalar cast follows:

```
shape [10]Sb;
float:Sb bf1;
float sf1;

with(Sb)
   sf1 = (float) bf1;
```

Casts may also be used for scalar pointers to parallel data — once again, no movement of data is implied. For example, the program fragment in the previous section may optionally contain casts, as follows:

```
int:Sd *p1, *p2;
int:Sa *q1, *q2;
p1 = (int:Sd *) palloc(Sd, boolsizeof(int:Sd));
q1 = (int:Sa *) palloc(Sa, boolsizeof(int:Sa));
```

If the shape of the target of a pointer is changed to actually cause data to later be accessed as if it were of a different shape, then the behavior of this operation is implementation defined.

It is also possible to use a cast to the **physical** shape to view any other shape as it is actually allocated on the machine. Such a physical view of a shape is dependent upon the shape's layout and is implementation defined. Any other use of cast, for instance casting a parallel value from one shape to another of different rank, size, or layout, has implementation defined behavior. Any implementation is free to allow these to function without signalling an error. They should simply allow a parallel value of one shape to be viewed as if it were of another shape. This kind of shape altering parallel-to-parallel cast will make layouts visible to the programmer. Shape altering parallel-to-parallel casts do not work in the early Paris implementations of the C* compiler (the compiler will allow the parallel value to be treated as if it were of the cast shape, but Paris still believes that the value is of the original shape and, if Paris safety is turned on, will signal a run-time error).

10.1 Index Mapping Function

Casting a parallel variable to a new shape does not guarantee row-major order of positions (row-major is the normal ordering of C arrays). Therefore, C* includes a built-in function to map left indices of parallel variables of one shape to left indices of that same parallel variable when cast to the physical shape. This function, **physical_index**, is defined as follows:

```
overload physical_index;

int physical_index(shape shape, int indices, ...);

int:current physical_index(shape shape, int:current indices,
          ...);
```

11 Functions

Both shapes and parallel variables may be passed to and returned from functions. It is acceptable to use a non-fully-specified shape to declare or define a function's arguments, return value, and local variables so long as the shape is fully specified when the function is invoked. Parallel arguments to functions and parallel return values must be of the current shape (the keyword **current** need not be used, but the specified shape does need to be the current shape when the function is called). In the same way that parameters in C are passed by value, parallel variables are also passed by value. This implies that there may be a hidden local variable for each parameter passed to a function. For parallel parameters, the storage required for such a local copy may not be negligible.

As occurs with assignment, a parallel expression passed by-value is only passed in the active positions. To allow all positions (i.e., including the inactive positions) of a parallel variable to be accessible from within a function, pass a pointer to the variable (or insure that all positions are active by using **everywhere** around the call). In practice, the contextualization of arguments and return value is only visible if the function accesses inactive positions via an **everywhere** statement in the function or if it performs communication into or from inactive positions.

As in C, arguments to a prototyped function are implicitly converted, as if by assignment, to the types of the corresponding parameters. Arguments to an unprototyped function undergo default argument promotions extended by parallel default argument promotions. Parallel default argument promotions include the parallel integral promotions [see §5.2] and the conversion that parallel **float** arguments are promoted to parallel **double**.

Here is a function that takes a parallel argument.

```
void print_sum(int:Sb x)
   {
      printf("Sum of parallel argument is %d\n", += x);
   }
```

This function would be called as follows:

```
print_sum(bi1);
print_sum(bi2);
```

Functions may also return parallel values:

```
float:current increment(float:current x)
   {
      return x+1.0f;
   }
```

This function might be called as follows:

```
bf1 = increment(bf2);
```

Shapes may also be passed as arguments and returned. Here is a function that takes a shape as an argument and allocates a local variable of that shape.

```
int number_of_active_processors(shape x)
   {
      int:x local;

      with(x) {
         local = 1;
         return += local;
      }
   }
```

11.1 Passing Arguments of Non-Current Shape

C* requires that all parameters and return values be of the current shape. This restriction is a natural consequence of parameter passing being similar to assignment. The assignment operators require that their operands are of the current shape, and the same is true of parameters and return values. However, C* does allow parallel variables of any shape to be passed into and returned from a function by reference (i.e., use a pointer to a parallel variable). [Please see §12 on Pointers.]

At high safety, the compiler will emit code to check the shape of such arguments at run-time to guarantee the type safety of the program. At low safety, such checks may be omitted.

11.2 Assertion Grammar for Functions

Any function may contain a set of assertions that can be evaluated at either compile- or run-time. These assertions may be used to indicate that the rank of an argument must be equal to a particular value, that the second dimension of one argument need be the same as the first dimension of another argument, that all arguments are of the same rank, and so forth. The assertions, if present, are written at the end of the argument list enclosed within braces (the code within the braces is known as a *type-check-block*) as follows:

```
float transpose(shape St, float:current array {
                    assert rankof(current) == 2;
                    assert rankof(St) == 2;
                    assert dimof(current, 1) == dimof(current, 0);
                    } );
```

These assertions should be included in both the header file (i.e., in the compilation unit that references the function) and the file that defines the function. In that way, both the caller and the callee know the set of assertions made by the programmer and can use them in code generation. The compiler will attempt to interpret the assertions at compile-time. If the assertions cannot be guaranteed at compile-time and if a sufficiently high safety level is specified, code will be generated to check the assertions at run-time.

The only statements that may appear within the *type-check-block* are assertion statements. The following operands may appear within the block: formal parameters that are parallel variables or shapes, constants, and the predeclared shape identifiers **physical** and **current**. Parallel variables may appear only when they are the operand of **sizeof** or **boolsizeof** or the argument of a call to **positionsof**, **rankof**, **dimof**, or **shapeof**. All binary operators, with the exception of the assignment and sequential expression operators, may be used. The ternary operator may be used. All unary operators, with the exception of pre-increment and pre-decrement, may be used. The operators **sizeof** and **boolsizeof** and the intrinsics **positionsof**, **rankof**, **dimof**, and **shapeof** may also appear in this context. [See §3.9 for a discussion of comparing shapes and shape equivalence.]

12 Pointers and Arrays

This section discusses the kinds of pointers that are available in C*. C* extends the set of possible pointer targets to include shapes, parallel types, functions which take and return extended C* types, and other pointer types composed of these. C* also allows arrays of parallel types to be created. The following are some examples of the pointer and array types (including those already available in C):

• Scalar pointer to a scalar **int**:

```
int *ptrtoint1;                ptrtoint1 = &si1;
```

• Scalar pointer to a scalar pointer to a scalar **int**:

```
int **ptrtoint2;               ptrtoint2 = &&si1;
```

• Scalar pointer to an unprototyped function returning a **double**:

```
double (*ptrtofcn1)();         ptrtofcn1 = fcn1;
```

• Scalar pointer to a prototyped function taking no arguments and returning a **char**:

```
char (*ptrtofcn2)(void);       ptrtofcn2 = fcn2;
```

• Scalar pointer to a prototyped function taking a pointer to an **int** as its only parameter and returning a **float**:

```
float (*ptrtofcn3)(int *p);    ptrtofcn3 = fcn3;
```

• Scalar pointer to parallel **int** of shape **Sb**:

```
int:Sb *ptrtoparint1;          ptrtoparint1 = &bi1;
```

• Scalar pointer to scalar pointer to a parallel **int** of shape **Sb**:

```
int:Sb **ptrtoparint2;         ptrtoparint2 = &&bi1;
```

• Scalar pointer to a parallel **int** of the current shape:

```
int:current *ptrtoparint3;     ptrtoparint3 = &bi1;
```

• Scalar pointer to a scalar **shape**:

```
shape *ptrtoshape1;            ptrtoshape1 = &Sb;
```

• Scalar pointer to a scalar pointer to a scalar **shape**:

```
shape **ptrtoshape2;           ptrtoshape2 = &&Sb;
```

• Array of 30 parallel **ints** of shape Sc:

```
int:Sc Aci[30];                ci2 = Aci[4];
                               ci2 = Aci[ci1];
```

• Scalar pointer to a prototyped function taking three parameters and returning a pointer to a parallel **float** of shape C. The parameters are: (1) a parallel **char** of the current shape, (2) a pointer to a parallel **double** of any shape, and (3) a pointer to a parallel **float** of shape **T**:

```
float:C *(*ptrtofcn4)(char:current a, double:void *b, float:T
*c);                           ptrtofcn4 = fcn4;
```

In the C tradition, it is a goal of C* that utilizing pointers will continue to be a fast operation. C*'s pointers are guaranteed to have an efficient implementation. In this regard, dereferencing pointers never involves communications among positions.

The application of the address-of operator, **&**, to a parallel lvalue produces a scalar pointer to such an lvalue. The operand of **&** need not be of the current shape, and no shape need be current to apply **&** to a parallel lvalue. The application of **&** to a shape produces a pointer to shape. Left-indexed expression may not be the operand of **&** [see §8.2].

The application of the dereference operator, *****, to a pointer to a parallel type produces the parallel type. The shape of the target of the pointer must be current for the result of the dereference to be used in a program. The application of the dereference operator, *****, to a pointer to a shape produces the shape. C* allows the dereference operator to be applied to the result of adding a scalar integral expression to either a pointer to parallel data or an array of parallel data (since an array of parallel data is coerced into a pointer to parallel data when used in an expression). The result has the type of the parallel data. As in C, when an integral expression is added to a pointer, the addition is performed so that the integral expression is scaled by units of the target of the pointer. For example, adding two to a pointer to a parallel **int** produces a pointer to a parallel **int** which is two parallel **ints** past the pointer's original target. C* also allows the dereference operator to be applied to the result of adding a parallel integral expression to an *array of parallel data*. The parallel integral expression must be of the same shape as the data in the array (i.e., the target of the pointer into which the array is coerced), which must be the current shape. The result of this operation also has the type of the parallel data but requires the parallel integral expression to be added in each position. This addition and subsequent dereference is an indirect addressing operation.

The result of adding a parallel integral expression to a pointer to parallel data may appear in an expression only if it is immediately dereferenced. Keep in mind that right indexing is exactly equivalent to the sequence of addition followed by dereference (i.e., a[b] ≡ *(a+b)). The addition of a parallel index to a pointer to parallel data produces a parallel pointer handle! This type is explicitly hidden from the programmer and no variables may be declared of this type. By making this construct illegal in all cases but this one, the language reserves the right to extend this construct in the future. It was a goal of the C* design to simplify the pointer types available. If it becomes useful to include parallel pointer handles in a future version of C*, this issue may be revisited.

Note that dereferencing **ptrtoint1** results in a scalar **int**; dereferencing **ptrtoint2** results in a pointer to a scalar **int**; dereferencing **ptrtoparint1** results in a parallel **int** of shape **Sb**; dereferencing **ptrtoparint2** results in a pointer to a parallel **int** of shape **Sb**; dereferencing **ptrtoparint3** results in a parallel **int** of the current shape; dereferencing **ptrtoshape1** results in a scalar shape; dereferencing **ptrtoshape2** results in a pointer to a scalar shape.

Dereferencing or indexing into **Aci** with a scalar integral expression (i.e., a *scalar* right index) results in a parallel **int** of shape **Sc**; indexing into **Aci** with a parallel integral expression of shape **Sc** (i.e., a *parallel* right index) results in a parallel expression of shape **Sc** and also requires indirection (indirect addressing) within each position. This is the only case in which a parallel right index is allowed in C*.

C* supports the concept of **void** shape in a manner not unlike that used for pointers to **void**. That is, the user may declare a pointer to a parallel variable of some currently unknown shape. Pointers to parallel variables of **void** shape maintain knowledge of the shape of the variable to which they actually point. This shape may be fetched through the use of the **shapeof** intrinsic function. When a pointer to a parallel variable is dereferenced, the variable may be manipulated as a parallel variable in the current shape. This pointer facility allows a pointer to a parallel variable of any shape to be passed into a function. If the variable is not of the current shape, the current shape may be set to be appropriate by using **with(shapeof(...))**. Note that unlike C, it is not necessary to cast a pointer to a parallel variable of **void** shape into a pointer to a parallel variable of some specific shape before dereferencing it. However, it is still necessary to cast a pointer if the base type is **void** (this is the use of **void** in pointer targets in Standard C).

13 Overloading

C* allows the programmer to overload functions, but not overload operators. Overloading is performed on the basis of argument types and number. That is, several declarations of a function with the same name may exist if the compiler is told that overloading is being utilized with an **overload** statement, which should precede the second declaration (and possibly precede all declarations) of the named function. Additionally, any two overloaded function declarations with the same name must differ in either the argument type of some argument or in the number of arguments accepted by the function (except if those declarations may form a composite type). Only prototyped functions may be overloaded.

The overload statement is simply the word **overload** followed by a comma-separated list of the names of functions which may be overloaded:

```
overload fcn1, fcn2, fcn3;
```

The **overload** keyword may also be used as a type-qualifier directly in the declaration of a function.

Most C* compilers will probably choose to utilize some form of name mangling to implement overloading. This is a scheme in which a new name that embodies all necessary overloading information is created for each overloaded function. However, because C* compilers need to remain compatible with other languages and compilers, they still need to be able to create unmangled names. C* accomplishes this goal by stating that any function declaration which precedes an **overload** designation of that name will not be mangled. Any function declaration which follows or includes the **overload** designation will be mangled. This capability imposes a constraint on the programmer: if some particular overloading precedes the **overload** designation in any one compilation module, it must precede the **overload** designation in all compilation modules.

For purposes of overloading, the types of arguments are considered in their entirety. That is, a pointer-to-**int** and a pointer-to-**float** are of different types. In addition, **struct** equivalence for overloading is consistent with C's equivalence of structs based on struct names and field types. Therefore, for two **struct**'s to be equivalent, they must have the same **struct** name and the same field types in the same order. In practice, these names and types are always consistent because common header files are used across compilation units.

In addition, overloading may be triggered on the basis of the *shape* of a parallel variable. That is, there may be one overloading for an **int** of shape **Sa** and another overloading for an **int** of shape **Sb**. [The initial implementations of C* do not, in fact, allow overloading based on shape. Because of this restriction, they allow only **current** and **void** shapes in the declarations of *overloaded* function formal parameters.]

When an overloaded function is called, the most appropriate overloading must be found. This search, known as the overloading algorithm, is detailed here. Initially, a search is started for an exact match based on the number and types of the arguments. If one overloading is found, the search has succeeded; if more than one overloading is found, an error is signalled. Next, the search proceeds by applying conversions to the arguments. The search is conducted by considering each parameter over all the overloadings. The first parameter's type in each overloading is compared to the first argument's type in the call. If the call could be made by applying some conversion, then the cost of that conversion is remembered. This is repeated for each overloading of that called function. When all overloadings have been considered, if there is a single minimum cost conversion, it is selected. If several overloadings have the same minimum cost, the search continues by considering the second parameter and argument. If the search tries all parameters and arguments and still has multiple overloadings to consider, an error is signalled. If no appropriate overloading is found (because no conversion could be applied to an argument to make it assignment compatible to the parameter's type), an error is also signalled. If a single function overloading is selected, a final check is performed to insure that all remaining arguments may be successfully converted to the types of the parameters. If any arguments cannot be converted, an error is signalled.

The conversions are listed in the following table in order of increasing cost:

Conversion	Examples *<actual argument, formal parameter>*
Exact match	<int, int> <float:S, float:S>
Conversion of base type to larger base type	<char, int> <bool:S, double:S>
Promotion of scalar to parallel	<char, char:S> <float, float:S>
Conversion of scalar base type to parallel larger base type	<char, int:S> <bool, double:S>
Conversion of base type to smaller base type	<int, char> <double:S,bool:S>
Conversion of scalar base type to parallel smaller base type	<int, char:S> <double,bool:S>

If the programmer wants to guarantee the invocation of one specific overloaded function, the arguments in the call must be cast to be of the appropriate types for that function. Although not present in the early C* implementations, a technique to take the address of one of the overloading of a function should be available. Following the lead of C++, it should be possible to assign the address of a function to a pointer or to cast the address of a function to a specific overloaded type and thereby cause that overloading to be selected. Although somewhat foreign to a C programmer (because the cast operator affects the overloading selection to which it is applied — that is, because understanding the expression requires information to flow up the parse tree), this approach was chosen to be consistent with C++. Here is an example,

```
overload float:current increment(float:current x);
overload double:current increment(double:current x);
double:current (*ptrtofcn1)(double:current);
float:current (*ptrtofcn2)(float:current);
   /*   Select the parallel double overloading becase of the
        declaration of the pointer: */
ptrtofcn1 = increment;
   /*   Select the parallel float overloading becase of the
        cast type: */
ptrtofcn2 = (float:current (*)(float:current)) increment;
```

14 Boolean Type

C* extends the basic set of types with a boolean integral data type. This type is named **bool**, a new reserved word. A **bool** undergoes the standard integral promotions specified in the C Standard and the extended parallel integral promotions. That is, when used as an rvalue, it is promoted to an **int** (when a parallel **bool** of the current shape is used an an rvalue, it is promoted to a parallel **int** of the current shape). A **bool** is unsigned — this is important when booleans are compared or interact with other signed or unsigned data. The modifier **signed** may not be applied to a **bool** because creating **signed bools** would cause boolean comparisons to return different results depending on the number of bits in a particular implementation. The size of a **bool** is at least one bit. Both scalar and parallel **bools** may exist. Their actual size and alignment are implementation dependent — on a CM-2 System, a parallel **bool** occupies one bit of memory and is aligned on a bit boundary. In this same implementation, a scalar **bool** is currently implemented as a **char**. It is possible for scalar and parallel **bools** to have different size and alignment, but it is not possible for them to have different sizes within one implementation. In particular, a scalar **bool** may not have a size of one when in a struct and a size of eight when declared outside a struct. The size of an object in units of **bools** may be determined through the use of the **boolsizeof** operator [See §3.7]. The address of either a scalar or a parallel **bool** may be taken. Unlike bit-fields, **bools** are not limited to appear only within structs, and arrays of **bools** may be formed.

When a non-**bool** is assigned to a **bool** or when a non-**bool** is cast to a **bool**, a logical conversion test occurs. That is, the value assigned or the result of the cast, respectively, is 0 if the operand was 0 and 1 if the operand was non-zero. When a **bool** is cast or assigned to a different integral type, the value 0 or 1 is the result.

14.1 Pointers to Booleans

Pointers to booleans are potentially different from pointers to other types. Specifically, if a pointer to a **bool** is a bit address and all other pointers are byte addresses (or, in general, if a pointer to a **bool** has a finer granularity address than other pointer types), translation would be required when a pointer is cast from a bit-based pointer to a byte-based pointer and vice versa. When a pointer to a **bool** is cast to a byte-based pointer type, the pointer will point to the byte of which the **bool** is a part. When a pointer to a byte-based pointer type is cast to **bool**, it is implementation dependent which bit it will point to.

14.2 Boolean String Handling Functions

The set of string handling functions is extended to include parallel overloadings for the Standard C **memcpy**, **memmove**, **memcmp**, and **memset** functions and to add equivalent scalar and parallel boolean-sized string handling calls. Prototypes for all of these functions are included in the **<string.h>** header file. [Please see Appendix A.4 for a complete description.]

15 Run-time Specification of Array Size

The array size of auto variables may be a run-time defined expression. This is acceptable for parallel arrays as well.

16 Calling Paris

The C* language does not define how parallel variables, shapes, or pointers to these are represented in memory, passed to functions, or returned from functions. The representation of, passing technique for, and returning technique for parallel variables, shapes, and pointers to these are implementation dependent.

The Paris implementation of C* for the Connection Machine System has chosen to document some of the above interface for that implementation. Because it is often desirable to be able to pass field-id's to Paris, the Paris implementation of C* simplifies this task by making the representation of a pointer to a parallel variable be a field-id. Therefore, Paris functions that expect field-id's may be called directly from C* by passing parallel variables by address.

When a Paris function expects a VP-set-id, call the function passing a C* shape as the appropriate argument.

17 Scans and Spreads

Scans and spreads are provided through functional interfaces. [Please see Appendix A.1 for a full description.]

Appendix A Library Functions

This appendix includes prototyped declarations of parallel versions of library functions (including standard library functions). [In practice (i.e., with a non-Standard-C back-end C compiler), the following rules apply: (1) only those scalar functions supported by the underlying C system will be supported by C* and (2) parallel versions of functions will exist only for those functions with scalar implementations.]

These function descriptions are prototyped function declarations. *type* is a placeholder for all C* base types — namely: **bool**, **signed char**, **unsigned char**, **signed short int**, **unsigned short int**, **signed int**, **unsigned int**, **signed long int**, **unsigned long int**, **float**, **double**, and **long double**. *ftype* is a placeholder for all C* floating-point types — namely: **float**, **double**, and **long double**.

A.1 Communication functions from <cscomm.h>

Please see the *C* Programming Guide* for more information on these functions.

```
#define CMC_no_field 0

extern int CM_error_on_failed_get;
enum CMC_collision_mode {CMC_no_collisions, CMC_few_collisions,
      CMC_many_collisions, CMC_collisions};
typedef enum CMC_collision_mode CMC_collision_mode_t;
typedef void (*CMC_combiner_t)();
extern CMC_combiner_t CMC_combiner_add;
extern CMC_combiner_t CMC_combiner_max;
extern CMC_combiner_t CMC_combiner_min;
extern CMC_combiner_t CMC_combiner_multiply;
extern CMC_combiner_t CMC_combiner_logand;
extern CMC_combiner_t CMC_combiner_logior;
extern CMC_combiner_t CMC_combiner_logxor;
extern CMC_combiner_t CMC_combiner_copy;
extern CMC_combiner_t CMC_combiner_overwrite;

type:current get(
   CMC_sendaddr_t:current send_address,
   type:void *sourcep,
   CMC_collision_mode_t collision_mode);
void get(
   void:current *destp,
   CMC_sendaddr_t:current *send_addressp,
   void:void *sourcep,
   CMC_collision_mode_t collision_mode,
   int length);

type:current send(
   type:void *destp,
   CMC_sendaddr_t:current send_address,
   type:current source,
   CMC_combiner_t combiner,
   bool:void *notifyp);
void:current *send(
   void:void *destp,
   CMC_sendaddr_t:current *send_addressp,
   void:current *sourcep,
   int length,
   bool:void *notifyp);

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);

type global(
   type:current source,
   CMC_combiner_t combiner);

type:current spread(
   type:current source,
   int axis,
   CMC_combiner_t combiner);

type:current copy_spread(
   type:current *sourcep,
   int axis,
   int coordinate);

type:current multispread(
   type:current source,
   unsigned int axis_mask,
   CMC_combiner_t combiner);

type:current copy_multispread(
   type:current *sourcep,
   unsigned int axis_mask,
   CMC_multicoord_t multi_coord);

void reduce(
   type:current *destp,
   type:current source,
   int axis,
   CMC_combiner_t combiner,
   int to_coord);

void copy_reduce(
   type:current *destp,
   type:current source,
   int axis,
   int to_coord,
   int from_coord);

unsigned int:current enumerate(
   int axis,
   CMC_communication_direction_t direction,
   CMC_scan_inclusion_t inclusion,
   CMC_segment_mode_t smode,
   bool:current *sbitp);

unsigned int:current rank(
   type:current source,
   int axis,
   CMC_communication_direction_t direction,
   CMC_segment_mode_t smode,
   bool:current *sbitp);

type read_from_position(
   CMC_sendaddr_t send_address,
   type:void *sourcep);

type write_to_position(
   CMC_sendaddr_t send_address,
   type:void *destp, type source);

CMC_multicoord_t make_multi_coord(
   shape s,
   unsigned int axis_mask,
   CMC_sendaddr_t send_address);
CMC_multicoord_t make_multi_coord(
   shape s,
   unsigned int axis_mask,
   int axes[]);
CMC_multicoord_t make_multi_coord(
   shape s,
   unsigned int axis_mask,
   int axis,
    ...);

CMC_sendaddr_t:current make_send_address(
   shape s,
   int:current axis,
    ...);
CMC_sendaddr_t:current make_send_address(
   shape s,
   int:current axes[]);
CMC_sendaddr_t make_send_address(
   shape s,
   int axis, ...);
CMC_sendaddr_t make_send_address(
   shape s,
   int axes[]);

type:current from_grid(
   type:current *sourcep,
   type:current value,
   int distance,
    ...);
void from_grid(
   void:current *destp,
   void:current *sourcep,
   void:current *valuep,
   int length,
   int distance,
    ...);

type:current from_grid_dim(
   type:current *sourcep,
   type:current value,
   int axis,
   int distance);
void from_grid_dim(
   void:current *destp,
   void:current *sourcep,
   void:current *valuep,
   int length,
   int axis,
   int distance);

void to_grid(
   type:current *destp,
   type:current source,
   type:current *valuep,
   int distance,
    ...);
void to_grid(
   void:current *destp,
   void:current *sourcep,
   void:current *valuep,
   int length,
   int distance,
    ...);

void to_grid_dim(
   type:current *destp,
   type:current source,
   type:current *valuep,
   int axis,
   int distance);
void to_grid_dim(
   void:current *destp,
   void:current *sourcep,
   void:current *valuep,
   int length,
   int axis,
   int distance);

type:current from_torus(
   type:current *sourcep,
   int distance,
    ...);
void from_torus(
   void:current *destp,
   void:current *sourcep,
   int length,
   int distance,
    ...);

type:current from_torus_dim(
   type:current *sourcep,
   int axis,
   int distance);
void from_torus_dim(
   void:current *destp,
   void:current *sourcep,
   int length,
   int axis,
   int distance);

void to_torus(
   type:current *destp,
   type:current source,
   int distance,
    ...);
void to_torus(
   void:current *destp,
   void:current *sourcep,
   int length,
   int distance,
    ...);

void to_torus_dim(
   type:current *destp,
   type:current source,
   int axis,
   int distance);
void to_torus_dim(
   void:current *destp,
   void:current *sourcep,
   int length,
   int axis,
   int distance);

void read_from_pvar(
   type *destp,
   type:current source);

type:current write_to_pvar(
   type *sourcep);
```

A.2 Math functions from <math.h>

```
/* Parallel overloadings */
overload ftype:current sqrt(ftype:current x);
overload ftype:current fabs(ftype:current x);
overload ftype:current exp(ftype:current x);
overload ftype:current log(ftype:current x);
overload ftype:current log10(ftype:current x);
overload ftype:current cos(ftype:current x);
overload ftype:current sin(ftype:current x);
overload ftype:current tan(ftype:current x);
overload ftype:current acos(ftype:current x);
overload ftype:current asin(ftype:current x);
overload ftype:current atan(ftype:current x);
overload ftype:current cosh(ftype:current x);
overload ftype:current sinh(ftype:current x);
overload ftype:current tanh(ftype:current x);
overload ftype:current acosh(ftype:current x);
overload ftype:current asinh(ftype:current x);
overload ftype:current atanh(ftype:current x);
overload ftype:current atan2(ftype:current x, ftype:current x2);
overload ftype:current pow(ftype:current, ftype:current);
overload ftype:current ceil(ftype:current);
overload ftype:current floor(ftype:current);
overload ftype:current truncate(ftype:current);
overload ftype:current frexp(ftype:current value, int:current *exp);
overload ftype:current ldexp(ftype:current x, int:current exp);
overload ftype:current modf(ftype:current value, ftype:current *iptr);
overload ftype:current fmod(ftype:current x, ftype:current y);
```

A.3 Utility functions from <stdlib.h>

```
/* Parallel overloadings */
overload int:current abs(int:current i);
overload int:current atoi(const char:current *);
overload long int:current atol(const char:current *);
overload void qsort(void:current *, size_t:current, size_t:current, int
      (*)(const void:current *, const void:current *));

/* Parallel functions */
void psrand(unsigned seed);
int:current prand(void);
void deallocate_shape(shape *s);
void:void *palloc(shape s, int bsize);
void pfree(void:void *pvar);
```

A.4 String Handling and Boolean String Handling functions from <string.h>

```
/* Parallel overloadings of memcpy, memmove, memcmp, and memset */
void:current *memcpy(void:current *s1, const void:current *s2, size_t
      n);
void:current *memmove(void:current *s1, const void:current *s2, size_t
      n);
int:current memcmp(const void:current *s1, const void:current *s2,
      size_t n);
void:current *memset(void:current *s, int:current c, size_t n);
```

New scalar and parallel boolean-sized memory manipulation calls follow:

```
bool *boolcpy(bool *s1, const bool *s2, size_t n);
bool:current *boolcpy(bool:current *s1, const bool:current *s2, size_t n);
```

Description:

The **boolcpy** function copies **n** booleans from the object pointed to by **s2** into the object pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.
Returns:

The **boolcpy** function returns the value of **s1**.

```
bool *boolmove(bool *s1, const bool *s2, size_t n);
```

```
bool:current *boolmove(bool:current *s1, const bool:current *s2, size_t n);
```

Description:

The **boolmove** function copies **n** booleans from the object pointed to by **s2** into the object pointed to by **s1**. Copying takes place as if the **n** booleans from the object pointed to by **s2** are first copied into a temporary array of **n** booleans that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by s1.

Returns:

The **boolmove** function returns the value of **s1**.

```
int boolcmp(const bool *s1, const bool *s2, size_t n);
int:current boolcmp(const bool:current *s1, const bool:current *s2, size_t n);
```

Description:

The **boolcmp** function compares the first **n** booleans of the object pointed to by **s1** to the first **n** booleans of the object pointed to by **s2**.

Returns:

The **boolcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

```
bool *boolset(bool *s, bool b, size_t n);
bool:current *boolset(bool:current *s, bool:current b, size_t n);
```

Description:

The **boolset** function copies the value of **b** into each of the first **n** booleans of the object pointed to by **s**.

Returns:

The **boolset** function returns the value of **s**.

Appendix B Sample Programs

B.1 Program to Compute Cuberoots

```
#include <math.h>

#define Epsilon 0.001
#define Limit 8192

shape [Limit]cubes;

double oneThird = 1.0/3.0;

double:cubes result;

double:cubes cuberoot(double:cubes a) {
  double:cubes x, nextX;
  int:cubes active;
  nextX = 1.0;
  active = 1;
  do
    where(active) {
      x = nextX;
      nextX = oneThird * ((x+x) + a/(x*x));
      active = (fabs(nextX-x)>=Epsilon);
    }
  while(|=active);
  return nextX;
}

main() {
  int i;

  with(cubes)
    result = cuberoot(pcoord(0)+1);

  for(i=1; i<=Limit; i++)
    printf("The cube root of %3d is %f\n", i, [i-1]result);
}
```

B.2 Program to Find Prime Numbers

```
#define MAXIMUM_PRIME 16384

#define FALSE 0
#define TRUE 1
#define FIRST_PRIME 2

/*                                                          */
/*    Function to find prime numbers                        */
/*                                                          */
/* Parameters:                                                     */
/*                                                                 */
/*    A pointer, "is_prime_p," to a one-dimensional parallel       */
/*    char which will have non-zero elements in all positions      */
/*    where the index is a prime number                            */
/*                                                                 */
/* Side effects:                                                   */
/*                                                                 */
/*    find_primes alters the one-dimensional parallel char         */
/*    which is pointed to by "is_prime_p."                         */
/*                                                                 */
/* Calling constraints:                                            */
/*                                                                 */
/*    The shape of the parallel char pointed to by                 */
/*    "is_prime_p" must be the current shape and all               */
/*    positions must be active                                     */
/*                                                                 */
/* Algorithm:                                                      */
/*                                                                 */
/*    This function will use the Sieve of Eratosthenes to          */
/*    find the prime numbers.  That is, it will iterate            */
/*    through all numbers which are indices to the one-            */
/*    dimensional parallel char                                    */
/*                                                                 */
void find_primes(bool:current *is_prime_p) {
  bool:current is_candidate;
  int minimum_prime;

  *is_prime_p = FALSE;

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

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

main() {
  shape [MAXIMUM_PRIME]s;

  bool:s is_prime;
  int i;

  printf("Finding primes...\n");

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

B.3 Program to Play Conway's Game of Life

```
#include <sys/types.h>
#include <stdio.h>
#include <cm/cmsr.h>
#include <cm/display.h>
#include <stdlib.h>

#define LIMIT 127
#define N 512

void initialize_display(void);

shape [N][N]S;

main() {
  time_t timeofday = time((time_t *)0);
  int gen_no = 0;
  bool:S bool_gen, save_gen;
  char:S neighbors;
  int:S generation;

  psrand(timeofday);
  with(S) everywhere {
    initialize_display();

    /* initialize the first generation to a random pattern at the
       lowest value (zero or one) -- also, generate the bool version
       of the generation.  "generation" has the display value for the
       frame buffer (values from zero to "LIMIT", inclusively); "bool_gen"
       has a zero or one value */
    bool_gen = generation = prand() & 1;

    do {
      CMSR_write_to_display(&generation);

      /* to check for stability and every other generation oscillation,
         save every other generation */
      if(++gen_no & 1) save_gen = bool_gen;

      /* count number of live neighbors */
      neighbors = [(.-1)%%dimof(S,0)][(.-1)%%dimof(S,1)]bool_gen +
        [.][(.-1)%%dimof(S,1)]bool_gen +
          [(.+1)%%dimof(S,0)][(.-1)%%dimof(S,1)]bool_gen +
            [(.-1)%%dimof(S,0)][(.+1)%%dimof(S,1)]bool_gen +
              [.][(.+1)%%dimof(S,1)]bool_gen +
                [(.+1)%%dimof(S,0)][(.+1)%%dimof(S,1)]bool_gen +
                  [(.-1)%%dimof(S,0)][.]bool_gen +
                    [(.+1)%%dimof(S,0)][.]bool_gen;

      /* a cell continues to live if it has 2 or 3 neighbors and a
         new cell is born if it has exactly 3 neighbors -- in all
         other cases, the cell dies or no birth occurs */
      where((neighbors == 3) | (bool_gen & (neighbors == 2))) {
        /* increment "generation" to change its color, but don't
           increment beyond "LIMIT" */
        where(generation < LIMIT)
        generation++;
      }
      else
        generation = 0;

      /* create the bool version of the generation, as well */
      bool_gen = generation;

      /* print out the generation number every 100 generations */
      if((gen_no % 100) == 0) printf("Generation %d\n", gen_no);

      /* loop until stability or every other generation oscillation has
         occurred */
    } while(|=(bool_gen != save_gen));

    /* print out the generation number when stability occurred */
    printf("Last generation %d\n", gen_no);
  }
}

void initialize_display(void) {
  int zoom;

  CMSR_select_display_menu(8, N, N);
  if(CMSR_display_type() == CMSR_cmfb_display) {
    zoom = (1024/N) - 1;
    CMFB_set_zoom(CMSR_cmfb_display_display_id(), zoom, zoom, 0);
    CMSR_set_display_offset(128/(zoom+1), 0);
  }
}
```

B.4 Matrix Multiply of Square Matrices

```
/*******************************************************************************

               Matrix multiply of two square matrices


     This is what we have been calling Cannon's matrix multiply.
       Below is the explanation of the algorithm:

        1) First both matrices being multiplied, A and B, get skewed
             along a dimension.  A will be skewed along dimension 1 and
             B along dimension 0.  The skewing vector is the same for
             both matrices, except it is applied to a different axis.

        2) Then a loop gets executed for as many rows as there are in a
             matrix.  Inside the loop:
               R = R + A*B
               A = A shifted by 1 along dimension 1
               B = B shifted by 1 along dimension 0

     Example of 3x3 matrix multiply:

     Vector of skewed values: [0 1 2]
     Matrix A before and after the skewing, it is skewed along dimension 1:

     +-            -+        +-            -+
     | a00 a01 a02 |         | a00 a01 a02 |
     | a10 a11 a12 |    ==>  | a11 a12 a10 |
     | a20 a21 a22 |         | a22 a20 a21 |
     +-            -+        +-            -+

     Matrix B before and after the skewing, it is skewed along dimension 0:

     +-            -+        +-            -+
     | b00 b01 b02 |         | b00 b11 b22 |
     | b10 b11 b12 |    ==>  | b10 b21 b02 |
     | b20 b21 b22 |         | b20 b01 b12 |
     +-            -+        +-            -+

     Now we enter the loop, which goes on for the number of rows in a matrix
       and we do a dot product.

     +-                      -+
     | a00b00 a01b11 a02b22 |
     | a11b10 a12b21 a10b02 |
     | a22b20 a20b01 a21b12 |
     +-                      -+

     Next we shift each matrix by one, along a dimension.  Matrix A would
       get shifted along dimension 1 and matrix B would get shifted along
       dimension 0.

     Then the loop is repeated.

     This alorithm is optimal, since no processor is ever idle.

*****************************************************************************/

#define size 512

shape [size][size]s;

float:s a,b,c;

main() {
  int i;

  with(s) {

/* Initialize the arrays */

    b = /*1.0f*/pcoord(0)*size+pcoord(1);
    a = /*pcoord(0)+1*/pcoord(0)*size+pcoord(1);

/* Skew both matrices first (using general communications) */

    [.][(.-pcoord(0)) %% dimof(s,1)]a = a;
    [(.-pcoord(1)) %% dimof(s,0)][.]b = b;

/* Perform matrix multiplication */
```

```
    c = 0.0f;
    for(i=0; i<size; i++) {
      c = c + a*b;
      [.][(.-1) %% dimof(s,1)]a=a;
      [(.-1) %% dimof(s,0)][.]b=b;
    }
  }
}
```

B.5 Program to Perform the Shuffle Exchange

```
#include <stdio.h>
#define DECK_SIZE 52

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

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

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

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

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

      /* generate the perfect shuffle transformation: positions in the
         first half of the deck are to be sent to consecutive even positions
         in the shuffled deck; whereas positions in the second half of the
         deck are to be sent to consecutive odd positions in the shuffled deck*/
      shuffling_order = (2*deck < DECK_SIZE) ? (2*deck) : (2*(deck-offset)+1);

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

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

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

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

```
        }
}
```

Index