# Data Parallel C Extensions


**Numerical C Extensions Group of X3J11**
**DPCE Subcommittee**
**Draft Technical Report**
**X3J11/94-015**
**WG14/~~N329~~**
**April 28, 1994**

## Foreword

This technical report is the result of discussions that began within the Array Syntax subgroup of the Numerical C Extensions Group (NCEG) in May 1989. The parent group became officially affiliated with the ANSI C process and obtained the ANSI designation X3J11.1. The subgroup, to emphasize its primary focus, became known as the Data Parallel C Extensions (DPCE) subcommittee.

This report addresses only a subset of the issues that were discussed by the subcommittee. There were many areas of contention in defining these extensions. The subcommittee chose to consider only those areas where consensus could be reached. Hence, not every proposed extension or viewpoint is represented in this report: this should not be construed as denying their utility or merit.

For example, there was great interest in including some form of parallel control flow, but it was agreed that since no consensus could be reached after considerable debate on this topic, that the subcommittee would leave it for a future extension. There was also interest in including overloading, to simplify the extension of library functions to handle data parallel objects, but its inclusion at this time was rejected, since it was not essential to the overall goals of the language extensions.

Appendix A describes proposed extensions that were not included, along with a summary of why those extensions were not accepted.

The scope of this report is intentionally limited to detailing only the most fundamental concepts of a consistent data parallel model. The extensions described in this report have been designed to allow further extensions to be cleanly added in the future. The subcommittee believes that further experience with the data parallel paradigm will eventually allow consensus to be obtained for a more broadly defined C extension.

The subcommitee began by adopting a general model that included the basic concept of parallel data aggregates that have structure (rank and dimension), memory layout (possibly noncontiguous), and context (active or participating elements). The subcommittee then began deliberations on how parallel aggregates should be accessed and used in expressions and statements.

As these concepts were already embodied in the C* language developed at Thinking Machines Corporation, the subcommittee adopted the C* reference manual [4] as its base document. It proceeded with removing features of the C* language that were not considered essential to the model, and adding more extensions, notably in the area of elemental and nodal functions and parallel pointer handles.

The process has been a lengthy one, and the culmination of the subcommitte's work is described here in a thorough examination of the needed extensions to

each section of the C standard. This exercise itself has identified and addressed inconsistencies in the model, and has improved and focused the report.

The DPCE subcommittee wishes to thank the following persons for their valuable contributions to its deliberation process:

Analog Devices, Marc Hoffman
Analog Devices, Kevin Leary
Analog Devices, Alex Zatsman
Control Data Corporation, Azar Hashemi
Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelson
Cray Research, Incorporated, Tom MacDonald
Cray Research, Incorporated, Dave Becker
Digital Equipment Corporation, Randy Meyers
Digital Equipment Corporation, Jeffrey Zeeb
Farance Inc, Frank Farance (Chair)
Hewlett Packard, John Kwan
HyperParallel Technologies, Christian Fortunel
HyperParallel Technologies, Nicolas Paris
IBM, Pawel Molenda
IBM, Bill O'Farrell
IBM, Fred Tydeman
Keaton Consulting, David Keaton
Lawrence Livermore National Laboratory, Linda Stanberry
MasPar, David Alpern
Mimosa Systems, Incorporated, Hugh Redelmeier
Open Software Foundation, Mike Meissner
SunPro, Bob Jervis
SunPro, Marino Segnan
Supercomputing Research Center, Maya Gokhale
Supercomputing Research Center, Howard Gordon
Syracuse University, Pankaj Kumar
Syracuse University, Nancy McCracken
Thinking Machines Corporation, Gary Sabot
Thinking Machines Corporation, James L. Frankel (Vice Chair)
University of New Hampshire, Phil Hatcher
Unix System Laboratories, David Prosser

# Data Parallel C Extensions

Linda Stanberry, Technical Editor
Lawrence Livermore National Laboratory
PO Box 808, L-300
Livermore, CA 94551
linda@ocfmail.ocf.llnl.gov

## 1. INTRODUCTION [ISO §1, ANSI §1]

### 1.1 PURPOSE [ISO §1, ANSI §1.1]

This document describes a set of extensions to Standard C that support programming of data parallel applications. The intent is to provide a set of machine-independent extensions that permit an efficient mapping to high-performance architectures, especially massively parallel architectures.

### 1.2 SCOPE [ISO §1, ANSI §1.2]

This document descibes only the data parallel extensions to the C Standard. It presents those extensions in the context of the relevant sections of the Standard to be modified, and introduces new subsections of the Standard where appropriate. It does not provide a tutorial on data parallel programming, nor on Standard C.

### 1.3 REFERENCES [ISO §2, Annex A, ANSI §1.3]

1. *American National Standard for Information Systems–Programming Language C* (X3.159-1989). Note: this is now withdrawn and replaced by [2].

2. *International Standard Programming Languages–C* (ISO/IEC 9899:1990(E)).

3. "C* Programming Guide," Thinking Machines Corporation (X3J11.1/90-032).

4. "A Reference Description of the C* Language," James L. Frankel (X3J11.1/91-023).

5. "ASX Evaluation Method - Revision 2," Frank Farance (X3J11.1/92-004).

6. "Massively Parallel C: Architectures and Data Distribution," Tom MacDonald (X3J11.1/92-007).

7. "C* Language Model," James L. Frankel (X3J11.1/92-010).

8. "C* answers to evaluation criteria," James L. Frankel (X3J11.1/92-011).

9. "Expressing Communication Costs in an Array Syntax," Dave Becker (X3J11.1/92-025).

10. "Focusing the ASX Base Document," Bob Jervis (X3J11.1/92-026).

11. "Issues concerning the use of C* as a base document," Frank Farance (X3J11.1/92-028).





85

12. "Distributing Data Using the 'block' Qualifier in C (revision 2)," Dave Becker, Kent Zoya, Bill Homer (X3J11.1/92-033).

13. "Elemental Execution," Phil Hatcher (X3J11.1/92-041).

14. "Left Indexing versus Right Indexing," Frank Farance (X3J11.1/92-044).

15. "ASX Ten Commandments," Frank Farance (X3J11.1/92-045).

16. "A Critique of the Programming Language C*," Walter F. Tichy, Michael Phillipsen, and Phil Hatcher (X3J11.1/92-050).

17. "Commentary on 'A Critique of the Programming Language C*'," Phil Hatcher (X3J11.1/92-051).

18. "A Detailed Response to the C* Critique by Tichy, Phillipsen, and Hatcher," James Frankel (X3J11.1/92-053).

19. "The Pros and Cons of Current Shape in C*," James L. Frankel (X3J11.1/92-054).

20. "A Proposed Worklist of Extensions/Changes to C*," James L. Frankel (X3J11.1/92-055).

21. "Parallel Processing Model for High Level Programming Languages (3/92)," Cherri Pancake (X3J11.1/92-056).

22. "MasPar's C Directions and Reasons," David Alpern (X3J11.1/92-062).

23. "Parallel Control Flow Constructs," David Alpern (X3J11.1/92-073).

24. "Elemental Execution," Phil Hatcher (X3J11.1/92-076).

25. "HyperC, A C language for Data Parallelism," HyperParallel Technologies (X3J11.1/92-081).

26. "FORALL Proposal for Base Document," Gary Sabot (X3J11.1/93-008).

27. "A Parallel Extension to ANSI C," Rob E. H. Kurver (X3J11.1/93-009).

28. "Nodal Functions: A Strawman," Phil Hatcher (X3J11.1/93-011).

29. "Parallel Pointer Handles," James L. Frankel (X3J11.1/93-013).

30. "Using Iterators to Express Parallel Operations in C (revision 4)," Dave Becker, Kent Zoya, Bill Homer (X3J11.1/93-050).

## 1.4 ORGANIZATION OF THE DOCUMENT [ANSI §1.4]

This document is organized into sections that correspond to the relevant sections to be modified within the Standard C document. Included with each extension is a brief rationale or example for the extension.





86

If rationale for an extension is included, it is distinguished by indentation and a change of font such as this.

The major sections of the document are:

1. Introduction
2. Environment
3. Language
4. Library

Each subsection of these major sections follows the structure of ANSI C [1] and ISO C [2], and indicates which subsections are modified. Cross references are noted at the beginning of each subsection, enclosed in square brackets–e.g., [ISO §7.1, ANSI §4.1]. The numbering of all subsections directly corresponds to the numbering within the cited ANSI/ISO standard. Subsections for which there is no corresponding ANSI or ISO subsection are new [NEW]. Subsections of the standard which are not affected are skipped, so the numbering of subsections within this proposal will not necessarily be consecutive.

In each subsection of this document, the text is to be considered as amplifying the existing text of that subsetion of the Standard, not replacing or modifiying it Where parts of specific definitions in existing subsections of the standard are modified, the modification is introduced by an italicized and underlined heading to that effect, such as:

*Modify:*

These headings are used for clarity as needed, and omitted where it is obvious that the entire change is reflected in the text (e.g., in **Syntax** definitions).

<<Editor's notes are delimited by double angle brackets. This indicates where the editor has identified clarifications or revisions that need to be completed. "TBS" within an editorial note indicates "to be supplied.">>

Change bars (|) are affixed in the right margin on each paragraph that has changed since the previous version of this document.

## 1.5 BASE DOCUMENTS [ISO §2, ANSI §1.5]

This set of extensions represents the composition of multiple proposals from participating representatives, as reflected in the list of references in §1.3. Early in the deliberations, the committee elected to adopt the C* language reference manual [4] as its base document, and with this as its foundation, derived the current set of extensions by deleting some features of C* and adding new features.

## 1.6 DEFINITION OF TERMS [ISO §3, ANSI §1.6]

The following new terms are used throughout this document. Although it would be more natural to define each term in the subsection where it is first introduced, it is also convenient to have the new terms dealing specifically with the data parallel



87

extensions collected in one place. Hence, the most widely used new terms are defined here.

- active position – a position whose values participate in elemental execution.

- context – the active positions of a shape.

- element – the value or object at a position within a parallel operand, respectively; or a member of an array.

- elemental execution – execution of a function or operation on elements within corresponding active positions of parallel operands.

  An operation performed under a context is executed elementally. That is, it is executed on each value or object at the positions designated as active for a given context.

- layout – information specifying a distribution of a parallel object or parallel value onto memory.

  Memory refers to the total composite memory of a computing system.

- parallel indexing – selecting elements of a parallel operand; single elements or multiple elements may be selected.

- parallel object – a structured collection of one or more identically-sized objects where the structure is defined by a shape.

  A parallel object is distinct from an ordinary C object in that although it is composed of C objects, the collection itself is not guaranteed to be contiguously allocated.

- parallel operand – parallel value or parallel object.

- parallel value – a structured collection of one or more identically-typed values where the structure is defined by a shape.

- physical – a predefined variable of type shape which is of rank 1 and dimension equal to the number of nodes in the execution environment.

- position – a point within the index space defined by the Cartesian product of the dimensions of a shape.

  A position of a shape denotes a point in all variables of a given shape.

- reduction – an operation that when applied to a parallel operand produces a single, nonparallel value, such as the sum of all the elements of a parallel object.

- shape – a type whose values consist of the following components: rank, dimensions, context, and layout. Shapes describe true multi-dimensional objects.

  Objects and values of type **shape** are descriptors or templates for parallel objects or parallel values. Variables may be declared to denote objects of type shape. See §3.1.2.5 and §3.5.2.

## 1.7 COMPLIANCE [ISO §4, ANSI §1.7]

In order to comply with this set of extensions, an implementation must provide for all the extensions detailed in this document.

## 1.8 FUTURE DIRECTIONS [ANSI §1.8]

The set of extensions here is intended as the minimal set of extensions needed to support data parallel programming. As this is a relatively new area of expertise, the DPCE subcommittee chose not to propose extensions in those directions where more experience is needed to evaluate alternate proposals. As such experience is gained, further data parallel extensions will be desirable to codify developing practice and promote portablity of data parallel applications.

# 2. ENVIRONMENT [ISO §5, ANSI §2]

## 2.1 CONCEPTUAL MODELS [ISO §5.1, ANSI §2.1]

These extensions are based on a data parallel model of programming. This model provides a single thread of control while allowing the manipulation of parallel objects. Parallel objects are manipulated by applying in parallel an operation across all the elements of the objects. Although the programmer's model presents the illusion of a single thread of control to simplify the program design, implementation, debugging, and maintenance tasks, the execution model may utilize many independent, unsynchronized threads or processes. In addition to the single thread model, DPCE also offers two mechanisms which allow the programmer to directly utilize a multithreaded model: modal function and elemental function invocations.

> The data parallel model supports a large class of parallel computations while being easy to learn and use. The ease of use is derived from its emphasis on a single thread of control, as used in serial programming. That is, this model is easier for users with respect to program design, debugging, and maintenance. The wide applicability is due in part to the demonstrated ability of compilers to translate data parallel programs for efficient execution on a variety of both serial and parallel hardware platforms.

**Example**

The following illustrates a comparison of programming style that one would use to perform the same operations on a parallel object using DPCE as one would use in C with arrays. Although the two code segments are not equivalent for the reasons noted, both demonstrate the same effect.

<<Add further clarification that you can't express in C what you can express in DPCE and why. Two column presentation might enhance the examples.>>

```
/* DPCE */
shape [100]Shape;
int:Shape x, y, z;

x = y + z;
x += 17;
```

```
/* ISO C */
typedef int Shape[100];
Shape x, y, z;

/* NOTE: The following is not truly equivalent to the DPCE
example above since the DPCE operations are not ordered as
the operations are ordered in the following loops.  Further,
in DPCE, operations are performed under 'context' and the
granularity is at the operation level rather than at the
statement level. */

for (i = 0; i < 100; i++)
      x[i] = y[i] + z[i];

for (i = 0; i < 100; i++)
      x[i] += 17;
```

### 2.1.1 Translation environments [ISO §5.1.1, ANSI §2.1.1]

<<TBS: discuss impact of data parallel model on translation environment, linking of DPCE compiled code with other C codes, new phases of translation for DPCE. >>

<<Bob Jervis volunteered to try to craft some words for this section.>>

### 2.1.2 Execution environments [ISO §5.1.2, ANSI §2.1.2]

At program startup, the DPCE execution environment shall define **physical** to denote a predefined variable of type shape which is of rank 1 and dimension equal to number of nodes <<need to clarify what is meant by "nodes">> in the execution environment. The layout of **physical** is implementation defined. <<Clarify whether this assumes gang scheduling??>>

<<TBS: discuss initialization of globals/statics, defaults, extern environment, execution, termination.>>

## 2.2 ENVIRONMENTAL CONSIDERATIONS [ISO §5.2, ANSI §2.2]

<<TBS: discuss any extensions for <limits.h>.>>

# 3. LANGUAGE [ISO §6, ANSI §3]

## 3.1 LEXICAL ELEMENTS [ISO §6.1, ANSI §3.1]

### 3.1.1 Keywords [ISO §6.1.1, ANSI §3.1.1]

The following keywords are added to the language only if the <dpce.h> header file is included. In addition to these keywords, <dpce.h> defines the **physical** shape identifier, and the functions described in §4 of this document.

*Add the following new keywords:*

**block**
**elemental**
**everywhere**
**nodal**
**scale**
**shape**
**where**

### 3.1.2.5 Types [ISO §6.1.2.5, ANSI §3.1.2.5]

*Add shape type:*

There is one *shape type*, designated as **shape**.

**shape** is an object type whose values consist of the following components: rank, dimensions, context, and layout. We refer to a value of type **shape** simply as "a shape."

A shape type of unknown size, whose rank and dimensions are not known, is *fully unspecified*. A shape type whose rank is known, but whose dimensions are not, is *partially specified*. A shape type whose rank and dimensions are known is *fully specified*. These three categories of shapes form three distinct subsets of the shape type. Note: in a declaration of a shape, either none or all of the dimensions must be specified.

A **void** shape designator can be used to specify a generic shape for parameter and return value types of a function prototype. See §3.5.

**Examples**

```
shape S;          /* fully unspecified shape */
shape []T;        /* partially specified shape */
shape [100]U;     /* fully specified shape */
shape [2][]V;     /* invalid */

int:void f(int:void arg); /* function that takes a parallel
                             int and returns a parallel int
                             of generic shape */
```

*Modify derived types to include parallel types:*

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows:

...

• A *parallel type* describes a nonempty, structured collection of objects or values with a particular member type, called the element type. The structure of the collection is defined by an associated shape. Parallel types are characterized by their element type and their shape. A parallel type is said to be derived from its element type and its shape, and if its element type is $T$ and its shape is $S$, the parallel type is sometimes called a "a parallel $T$ of shape $S$." The construction of a parallel type from an element type and a shape is called "parallel type derivation."





9)

*Modify recursive applicability of constructing derived types:*

These methods of constructing derived types can be applied recursively, except:

- the element type of a parallel type shall not be a parallel type;
- a struct or union type shall not contain a member that has parallel type or a member that is a shape;
- a parallel struct or union shall not contain a member that is a pointer.

Hence, you can have arrays of parallel types, functions returning parallel types or having parallel typed arguments, pointers to parallel types, parallel types whose elements are arrays or structs or unions or functions or pointers. But you can't have parallel types whose elements are parallel types or contain parallel types.

*Modify aggregate types to include parallel types:*

Array, structure, and parallel types are collectively called *aggregate types.*

*Modify pointer storage requirements*

A pointer to a parallel type need not have the same representation as a pointer to the corresponding nonparallel type. << Do we need to say something about whether or not a pointer to parallel can be cast to/from a void pointer? The current standard says that a void pointer shall have the same representation as a char*, and if we want to allow casts to/from void*, I wonder how this can be so. See §3.2.2.3.>>

**Forward references:** shape specifiers (§3.5.2.4)

### 3.1.2.6 Compatible and composite types [ISO §6.1.2.6, ANSI §3.1.2.6]

*Add clarifications for compatible shapes and compatible parallel types:*

Two shapes are compatible under the following conditions:

- A fully unspecified shape type is compatible with any other shape type
- A partially specified shape type is compatible with a fully specified shape type if the partially shecified shape type has the same rank as the fully specified shape type
- Two fully specified shape types are compatilbe only if they are the same type

Two shape types are the same type if they specify the same rank, dimension, and layout. See §3.2.3 and §3.3.16.1.

<< We need to determine if the following definition of compatible parallel types is useful. Currently, it is not used anywhere else in the document explicitly. We had originally intended this to be used in determining when parallel operands were valid for binary and ternary operations (see §3.2.3), but the connection between those two has been lost in version 1.4 of this document.>>

Two parallel types are compatible if they are derived from compatible element types unless they are derived from distinct shape variables.

Parallel types are determined to be compatible if they have compatible element types and their shapes are denoted by the same shape identifier, or if one or both are denoted by shape-valued expressions other than simple shape identifiers.

**Examples**

```
/* Compatible and incompatible shapes */
shape [10]S;
shape [10]T;        /* Compatible with S */
shape []U;          /* Compatible with S and T */
shape [][]V;        /* Incompatible with S, T, and U */
shape [2][5]W;      /* Incompatible with S, T, and U */
shape [][][]X;      /* Incompatible with S, T, U, V, and W */
shape Y;            /* Compatible with S, T, U, V, W, and X */

/* Compatible types */
int:S
signed int:S                /* Compatible with int:S */

long:physical
long int:physical           /* Compatible with long:physical */

/* Incompatible types */
int:S
float:S             /* Incompatible with int:S */
short:S             /* Incompatible with int:S */
int:T               /* Incompatible with int:S */
```

*Modify composite types to include composite shape types:*

A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisifies the following conditions:

- If one type is a fully unspecified shape and the other is fully specified, the composite type is identical to the fully specified shape.

- If one type is a fully unspecified shape and the other is a partially specified shape, the composite type is identical to the partially specified shape.

- If one type is a partially specified shape and the other is fully specifed, the composite type is identical to the fully specified shape.

<<I had a note to add layout to composite type. Only fully specified shapes can specify layout, and we said that two fully specified shapes were compatible only if they had the same layout. Do we want to amend that, and add a bullet for forming a composite from two fully specified types where only one specifies layout? We need some examples for this section.>>

To form a composite shape type, the shapes have to be compatible, and the composite will always be the more completely specified shape.

<<I think we should add wording to form composite types from elemental functions and non-elemental functions such that we can have <dpce.h> declare the functions to become elemental under DPCE for <math.h>, <stdlib.h>, and <string.h>. The wording should make the composite type be elemental regardless of the ordering of

header file inclusion. I think this will work better than trying to require modification of each of these other standard header files for implementations supporting DPCE.>>

### 3.1.5 Operators [ISO §6.1.5, ANSI §3.1.5]

*Add new operators:*

**<?  <?=  >?  >?=  $$**

## 3.2 CONVERSIONS [ISO §6.2, ANSI §3.2]

#### 3.2.1.1 Characters and integers [ISO §6.2.1.1, ANSI §3.2.1.1]

*Add to the integral promotions:*

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**; the promoted parallel value will be of the same shape as the original expression.

Integral promotions are applied elementally to parallel operands.

#### 3.2.1.5 Usual arithmetic conversions [ISO §6.2.1.5, ANSI §3.2.1.5]

*Add for parallel operands:*

In general, arithmetic conversions performed on a parallel operand result in a parallel value of the same shape as the operand and whose value at each position is the result of performing the usual arithmetic conversions on the value at the corresponding position of the operand.

The usual arithmetic conversions are applied elementally to parallel operands, and the result is an homogenous parallel operand.

Exceptions are noted for specific operators in the following sections.

### 3.2.3 Parallel Operands and Contextualization [NEW]

In general, in binary or ternary operations involving operands of parallel types, the operands must all be of the same shape, and the behavior is undefined if the shapes are not the same.

Two fully specified shapes are the same if they are compatible and if they have the same context.

Two shapes are the same if they are structurally equivalent at run time. Since shapes may be dynamically specified and/or modified, it may not be possible to determine at compile time if two shapes will be the same at run time; the compiler must check, however, that the two shapes are compatible.

If one operand is parallel and one is nonparallel, the nonparallel operand is promoted to a parallel value of the other operand's shape by replicating the nonparallel operand's value.

If the operator is an assignment operator, this replication only applies when the left operand or destination is parallel and the right operand or source is nonparallel.

Other exceptions to these rules are noted for each operator in the sections that follow.

The context component of a shape is a specification of which positions of a parallel operand of the shape are active for a given operation. The context component of a shape is conceptually a parallel integral value of the shape <<maybe this should be described as a multi-dimensional bit mask instead since a parallel int of this shape also has context, which is what I'm trying to define!>>, where a position is indicated as active by a non-zero in the corresponding context element, and as inactive by a zero in the corresponding context element. The elements of the context of a given shape are all non-zero when the shape is initialized, indicating that all positions are active.

The context of the shape associated with a parallel operation, including function calls, determines which positions of the operands participate in the operation. Active positions participate in the operation, and inactive ones do not.

The context of a given shape is altered by context-modifying statements and expressions which assign elements of the context. The **where** and **everywhere** statements modify the context component of a shape (see §3.6.7). Expressions involving the **&&**, **||**, and **:?** operators also modify the context component of a shape.

## 3.3 EXPRESSIONS [ISO §6.3, ANSI §3.3]

### 3.3.2 Postfix Operators [ISO §6.3.2, ANSI §3.3.2]

#### 3.3.2.1 Array subscripting [ISO §6.3.2.1, ANSI §3.3.2.2]

*Add arrays of parallel objects indexed by nonparallel or by parallel ints:*

**Constraints**

One of the following shall hold:

- One of the expressions shall have type "pointer to object *type*," the other expression shall have integral type, and the result has type "*type*"

- One of the expressions shall have type "pointer to parallel object of type *type*," the other expression shall have integral type, and the result has type "*type*"

- One of the expressions shall have type "pointer to *T*," where *T* is of parallel type, the other shall have parallel integral type of the same shape as *T*, and the result has type parallel pointer to *T*.

**Semantics**





95

A postfix expression E1 followed by an expression E2 in square brackets [] is: (1) if E2 is an integral expression, a subscripted designation of a member of an array object; or (2) if E1 is an array of parallel type *T* and E2 is of parallel integral type, a parallel pointer.

<<Do we need to add more words about pointers to (arrays of) parallel objects here (or in §3.3.2.4) about how these operators get the next member of an array of parallel? I think we want to say that these operators work with arrays of parallel the same way they work with arrays of nonparallel, but I'm not sure where to put in those words. Suggestions?>>

**Examples**

```
shape [20] Shape;
int:Shape x;
int:Shape y[50];
int:Shape *z;
int i;

z = &y[39]; /* z is a subarray of y */
y[i];       /* Designates the i-th member of y, which is a
               parallel int of shape Shape */
z[i];       /* Designates the i-th member of z */
y[x];       /* Designates int:Shape ??? */
```

### 3.3.2.2 Function calls [ISO §6.3.2.2, ANSI §3.3.2.2]

**Constraints**

*Add for elemental functions:*

Each parallel argument to an elemental function shall have a type such that an element value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

*Add for nodal functions:*

Each parallel argument to a nodal function shall have a corresponding parameter such that one of the following holds:

- the parameter type is a parallel type of the void shape, such that an element value of the argument may be assigned to an object with the unqualified version of the type of an element of the parameter; or
- the parameter type is a pointer of nonparallel type, such that an element value of the argument may be assigned to an object with the unqualified version of the type of the object pointed to by the parameter; or
- the parameter type is a nonparallel type, such that an element value of the argument may be assigned to an object with the unqualified version of the type of the parameter.

A nonparallel argument to a nodal function shall have a type such that it can be promoted to the corresponding parallel type of shape physical.

An argument to a nodal function shall not be a pointer to a nonparallel type.

**Semantics**

*Add clarifications for parallel parameters in prototypes, parallel arguments, parallel return types:*

Both shapes and parallel operands may be passed to and returned from functions. Shape and parallel operand arguments are passed by value; creating the local copy of these arguments to a function can be inefficient. The usual rules for function calls with and without prototypes applies.

Parallel arguments and return values are evaluated under the context of the expression in which the function call occurs. Parallel operands passed as arguments behave as if assigned to the local copy, following the semantics of assignment under context (see §3.3.16). Only active positions are assigned.

To allow access to all positions of a parallel object, use an **everywhere** statement around the call, or pass a pointer to the object.

**Examples**

```
shape [100]Shape;
int:Shape a, b;

int active_positionsof(shape *x, int:(*x) mask)
{
      int:(*x) local;

      with (mask) {
            local = 1;
            return += local;
      }
}
void print_sum(int:Shape x)
{
      printf("Sum of parallel argument is %d\n", +=x);
}

int:Shape increment(int:Shape x)
{
      return x++;
}

/* examples of use */
print_sum(a);

b = increment(a);

printf("Number of positive elements in a is %d\n",
      active_positionsof(Shape,a>0));
```

*Add for elemental functions:*

A function whose return type is qualified with the **elemental** qualifier is called an elemental function. The return type of the function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.3, 3.6.6.4, and 3.7.1.

<<It really isn't the return type that is qualified, it is the function behavior; hence we need to consistently craft words for this for nodal and elemental functions in this section and in §3.5.3.>>

An elemental function can be executed either elementally or non-elementally. If none of the arguments are parallel operands, the function is executed non-elementally (i.e., as a normal C function). If one or more of the arguments are parallel operands, it is executed elementally. All the parallel arguments must be of the same shape; if not, the behavior is undefined. Nonparallel arguments to a function being executed elementally are promoted to parallel in the usual manner.

When an elemental function is executing elementally, a shape is established at run time for the function. This shape is the shape of the parallel arguments. An instance of the code contained in an elemental fucntion is executed for each active position of the established shape. An elemental function is indivisible with respect to synchronization. That is, an elemental function is treated as if it was a basic operator (like addition). All instances of the function execute as if in parallel, there are no assumptions about the synchronization of the intermediate steps, and there is a conceptual synchronization upon exit of the function.

When an elemental function is called from within an elemental function that is not executing elementally, the inner function call also executes as if it were a non-elemental function.

When an elemental function is called from within an elemental function that is executing elementally, the positions that execute the inner function call continue to execute the body of the called function elementally.

An elemental function executing elementally, which returns a non-void type, returns a parallel value of its established shape. When not executing elementally, an elemental function which returns a non-void type returns a non-parallel value.

**Examples**

```
shape [10] Shape;
int:Shape x, y;

elemental int f(int a, int b)
{
   return(a+b);
}

elemental int g(int a)
{
   return(f(a,a));
}

f(x,y);      /* Returns parallel int of shape Shape with sum
             of active positions of x and y. Inactive
             positons are unspecified. */
```

```
f(x,1);     /* Returns parallel int of shape Shape with sum
               of 1 and active positions of x; inactive
               positions are unspecified. */
f(1,2);     /* Returns int with sum of 1 and 2. */
g(x);       /* Returns parallel int of shape Shape with
               values equal to two times the values of x in
               the active positions. Inactive positions are
               unspecified. */
g(1);       /* Returns two times 1. */
```

*Add for nodal functions:*

A function whose return type is qualified with the **nodal** qualifier is called a nodal function. The return type of the function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.3, 3.6.6.4, and 3.7.1.
<<It really isn't the return type that is qualified, it is the function behavior; hence we need to consistently craft words for this for nodal and elemental functions in this section and in §3.5.3.>>

An invocation of a nodal function occurs as if the function is invoked once on each node of the execution environment in Single Program, Multiple Data (SPMD) style; that is, as if a separate thread is spawned on each node to execute the function body. Nodal functions therefore provide an escape to a multi-threaded programming model. These threads, one per node, are only required by the execution model to synchronize upon return from the nodal function.

On each node the body of the nodal function executes in a temporarily established single-node environment called the nodal execution environment. That is, during the execution of a nodal function, a call to **positionsof(physical)** will return 1. The execution environment of the caller is re-established upon return from the nodal function.

A nonparallel argument to a nodal function is first promoted to a parallel value of the physical shape by replication. Then the argument is processed as if it was originally a parallel argument.

For a parallel argument to a nodal function, a thread of the nodal function will receive exactly those positions of the argument that are stored on the node executing the thread.

For a parallel argument to a nodal function, if the corresponding parameter is of parallel type, then for each thread executing the nodal function, a shape object will be created whose rank, dimensions, context and layout are derived from the shape of the argument in the following way:

- The rank of the shape of the parameter is equal to the rank of the shape of the argument.

- The dimensions of the shape of the parameter are derived from the layout of the shape of the argument. For each dimension the number of positions is equal to the number of distinct parallel-index values in that dimension for the set of elements of the argument mapped to the node executing the thread.

- The context of the shape of the parameter will be initialized from the context of the shape of the argument: a position will be active in the parameter's shape if its corresponding position in the argument's shape is active; otherwise the position will be inactive.

- The layout of the shape of the parameter will reflect that the nodal function executes in a single-node environment. All positions will be mapped to the single node.

This shape will be returned when the **shapeof** function is applied to the parallel parameter during execution of the nodal function. If there is more than one argument of the same shape, all being passed to parameters of parallel type, then the same shape <<object??>> will be returned by application of **shapeof** to any of these parameters. The corresponding parameter will receive the values of its argument in those positions stored on the node executing the thread. The correspondence of positions in the shape of the argument and positions in the shape of the parameter is implementation defined, but will be the same for all arguments of the same shape.

Note that a parallel argument is evaluated under context and, if a reference is made to an inactive positon of the corresponding parameter, the behavior is undefined.

For a parallel argument to a nodal function, if the corresponding parameter is of pointer type, then, for each thread executing the nodal function, an array object will be created that contains the values of the argument in those positions stored on the node executing the thread. Memory for the array will be allocated at the time of the call to the nodal function and will be freed upon return from the nodal function. The corresponding parameter will receive the created array object. The correspondence of positions in the shape of the argument and elements in the passed array is implementation defined, but will be the same for all arguments of the same shape.

For a parallel argument to a nodal function, if the corresponding parameter is of nonparallel type and is not of pointer type, then the argument must be of the **physical** shape and each thread of the nodal function will receive as the parameter the single value of the argument stored on the node executing the thread. If the argument is not of the **physical** shape, then the behavior is undefined.

When the element type of a parallel argument to a nodal function is a pointer type, the pointer value at each position is converted so that the corresponding elements of each thread's parameter array receives a pointer to the object stored at the corresponding position of the parallel object pointed to by the argument's pointer value. <<Reworded because of other changes for pointer vs array parameters; recheck for accuracy.>>

If no positions of a parallel argument to a nodal function are stored on the node executing a thread of the nodal function, then the behavior is undefined.

**Examples**

```
nodal void f1(int:void, int [], int *, int);
shape [100]S;
int:S px;
int:physical py;
int i;
```

```
int *ptr;
int:physical * :physical par_ptr_to_par;
int:S * :S par_ptr_to_par2;

nodal void f2(int:void * :void, int * [], int **);

f1(px, px, px, py);
            /* valid <<but need to add annotation of what
            parameters receive for the given arguments>> */

f1(py, py, py, py);
            /* valid <<complete annotation>> */

f1(i, i, i, i);
            /* valid - nonparallel promoted to physical
            shape first <<complete annotation>> */

f1(px, px, ptr, py);
            /* invalid? can't promote pointer to nonparallel
            to parallel pointer to scalar? If this promotion
            is allowed, then we will need to add a
            constraint to disallow passing pointers to
            scalar here, because pointers to nonparallel do
            not make sense within a nodal function. */

f1(px, px, par_ptr_to_par, py);
            /* I would report this as invalid because the
            base type of the parameter pointer is "int", and
            the element type of the argument is "int *", or
            perhaps more specifically "int * :S". */

f2(par_ptr_to_par2, par_ptr_to_par2, par_ptr_to_par2);
            /* I want this to be valid. <<complete
            annotation>> */

<<Add example with, say,

   int *:S par_ptr_to_int;
   f1(px,ps,par_ptr_to_int,py);
??>>
```

### 3.3.2.3 Structure and union members [ISO §6.3.2.3, ANSI §3.3.2.3]

**Semantics**

*Add:*

If the first operand of the . operator is of parallel type, the result is a parallel value of the same type as the member designated by the second operand; the value at each position of the result is the designated member at the corresponding position of the first operand.

**Examples**

```
shape [10]Shape;
struct Struct { int i; float f; };
struct Struct:Shape s;
```





101

```
struct Struct:Shape *p;

s.i             /* Denotes a parallel int value whose elements
                   are the corresponding int members of the
                   parallel struct s */

p->f            /* Denotes a parallel float value whose elements
                   are the corresponding float members of the
                   parallel struct pointed to by p */
```

### 3.3.2.4 Postfix increment and decrement operators [ISO §6.3.2.4, ANSI §3.3.2.4]

**Constraints**

*Revise as indicated:*

**The operand of the postfix increment or decrement operator shall have qualified or unqualified, parallel or nonparallel, scalar type and shall be a modifiable lvalue.**

**Semantics**

*Add:*

**If the operand of the postfix increment or decrement operator is of parallel type, each position of the parallel operand is incremented or decremented, respectively.**

**<<Do we need to add more words about pointers to parallel objects here (or back in §3.3.2.1) about how these operators get the next member of an array of parallel? I see I've included an example, but no words here to explain them. I think we want to say that these operators work with arrays of parallel the same way they work with arrays of nonparallel, but I'm not sure where to put in those words. Suggestions?>>**

**Examples**

```
shape [10]Shape;
int:Shape x;
int:Shape y[20];
int:Shape *p = y; /* p points to first element of y, y[0] */

x++;            /* Increments each element of x */
p++;            /* Increments p to point to the next array
                   element of y (y[1]), which happens to be a
                   parallel int. */
```

## 3.3.3 Unary Operators [ISO §6.3.3, ANSI §3.3.3]

*Revise as indicated:*

**Syntax**

*unary-expression:*
*postfix-expression*
*++ unary-expression*
*-- unary-expression*
*unary-operator cast-expression*

**sizeof** *unary-expression*
**sizeof** ( *type-name* )
*parallel-index postfix-expression*
*reduction-operator postfix-expression*

*parallel-index:*
[ *expression* ] *parallel-index*
[ *expression* ]

*reduction-operator:* one of

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

### 3.3.3.1 Prefix increment and decrement operators [ISO §6.3.3.1, ANSI §3.3.3.1]

**Constraints**

*Revise as indicated:*

The operand of the prefix increment or decrement operator shall have qualified or unqualified, parallel or nonparallel, scalar type and shall be a modifiable lvalue.

**Semantics**

*Add:*

If the operand of the prefix increment or decrement operator is of parallel type, each position of the parallel operand is incremented or decremented, respectively.

**Examples**

```
shape [10]Shape;
int:Shape x;
int:Shape y[20];          /* An array of parallel ints */
int:Shape *p = &x;        /* Pointer to the parallel int x */

--x;          /* Every element of x is decremented */
--y[10];      /* Every element of y[10] is decremented */
--(*p);       /* Every element of x is decremented */
```

### 3.3.3.2 Address and indirection operators [ISO §6.3.3.2, ANSI §3.3.3.2]

**Constraints**

*Revise as indicated:*

The operand of the unary **&** operator shall be either a function designator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier, or an lvalue that designates a parallel object that is not a bit-field and is not declared with the **register** storage-class specifier.

*Add:*

If the operand of the unary & operator is a parallel-indexed expression, at least one of the index expressions shall be parallel. If the operand of the unary & operator is a parallel-indexed expression, the shape of the parallel index expression(s) shall be the same as the shape of the expression being indexed. <<How will this be enforceable if the shapes cannot be determined until runtime? Should we make this undefined behavior instead of a constraint violation?>> (See also §3.3.3.5.)

Note: you cannot take the address of an element of a parallel operand.

**Semantics**

*Add:*

The application of & to a parallel-indexed lvalue produces a parallel pointer whose shape is the shape of the parallel index to that parallel lvalue. If the operand has type "parallel T of shape S parallel-indexed by expressions of shape S," the result has type "parallel pointer of shape S to parallel T of shape S."

The application of & to a parallel lvalue produces a pointer to that lvalue. If the operand has type "parallel T of shape S" the result has type "pointer to parallel T of shape S."

Note that the application of & to a shape object produces a pointer to that shape.

**Examples**

```
float a, b;
shape [10]Shape, f();          /* shape Shape, function f
                                  which returns a shape */
int:(f()) w;                   /* parallel int whose shape is
                                  the return value of f() */
int:Shape x;                   /* parallel int */
double:Shape y, z;             /* parallel double */
shape *sp;                     /* pointer to shape */
int:Shape *p;                  /* pointer to parallel int */
float *pp2f:Shape;             /* parallel pointer to
                                  float */
double:Shape *pp2pd:Shape;     /* parallel pointer to
                                  parallel double */
int:Shape A[20];               /* array of parallel int */
double:Shape B[30];            /* array of parallel double */

sp = &Shape;          /* Assigns sp to point to shape Shape */
p = &x;               /* Assigns p to point to parallel int x */
p = A;                /* Since A is an array, it is coerced to a
                         pointer to parallel int.  p is assigned
                         that pointer, which is the address of
                         the first element of A */
pp2f = &a;            /* Assigns each element of pp2f to point
                         to float a */
[3]pp2f = &b;         /* Assigns 3rd element of pp2f to point to
                         float b */
pp2dp = &y;           /* Assigns each element of pp2pd to point
                         to parallel double y */
[2]pp2dp = &z;        /* Assigns 2nd element of pp2pd to point
                         to parallel double z */
```

```
pp2pd = &B[x];      /* Assigns each element of pp2pd to point
                       to parallel double which is the x-th
                       element of B */
pp2pd = &[x]y;      /* Assigns elements of pp2pd to be the
                       address of parallel double y; which
                       elements of pp2pd are assigned depends
                       on the parallel index x (see §3.3.3.5).
                    */ <<??>>
&[3]x;              /* Constraint violation, can't take & of
                       an element of a parallel operand */
&[w]x;              /* Possible constraint violation,
                       depending on the shape of w */ <<??>>
```

**The result of the * operator applied to a parallel pointer of shape S to parallel type T also of shape S is a parallel lvalue of parallel type T and shape S.**

**The result of the * operator applied to a pointer to parallel type T of shape S is a parallel lvalue of parallel type T and of shape S.**

**<< I think we need to say somewhere what a parallel lvalue is!!>>**

The result of the * operator applied to a pointer to shape produces the shape.

**Examples**

```
float a,b;
shape [10]Shape;
shape *sptr = &Shape;           /* initialized to point to
                                   Shape */
int:Shape x = 3;                /* initializes all elements
                                   to 3 */
int:Shape *ptr = &x;            /* initialized to point to
                                   parallel int x;

double:Shape B[30];
float *pp2f:Shape = &a;         /* initializes all elements
                                   to point to float a */

double:Shape *pp2pd:Shape;

[7]pp2f = &b;       /* Assigns the 7th element of pp2f to
                       point to float b */
[5]x = 2;           /* Assigns 2 to the 5th element of x */
pp2pd = &B[x];      /* Assigns corresponding elements of pp2pd
                       to point to parallel double elements
                       of B as directed by parallel subscript
                       x */

*sptr;          /* Denotes the shape pointed to by sptr; in this
                   case Shape */
*ptr;           /* Denotes the parallel int x */
*pp2f;          /* Denotes a parallel float of shape Shape; in
                   this case equal to a in all positions except
                   equal to b in position 7 */
*pp2pd;         /* Denotes a parallel double of shape Shape; in
                   this case equal to the corresponding element of
                   B[3] in all positions except equal to the
                   corresponding element of B[2] in position 5. */
```

### 3.3.3.3 Unary arithmetic operators [ISO §6.3.3.3, ANSI §3.3.3.3]

**Semantics**

*Add:*

If the operand of a unary arithmetic operator is of parallel type, the result is a value of the parallel type, where the value at each position of the result is determined by applying the operator to the operand's value at the corresponding position.

**Examples**

```
float:Shape f;

-f          /* Denotes a parallel float value whose value at
               each position is the negation of the value at
               the corresponding position of f */
```

### 3.3.3.4 The sizeof operator [ISO §6.3.3.4, ANSI §3.3.3.4]

**Semantics**

*Add:*

The result of the **sizeof** operator applied to a parallel type or a value of parallel type is the size of the element type; it reflects the storage requirements for an element, including possible alignment constraints.

Hence, it may not be the same as the result of **sizeof** applied to its nonparallel counterpart.

**Example**

```
shape [10]Shape;
int:Shape *ap;

/* Allocate an array of 10 parallel ints of shape Shape */
/* Is an implicit cast allowed?? */
ap = (int:Shape *)malloc(sizeof(int:Shape) * 10);

/* Assign elements of the array, which are parallel ints */
ap[0] = 0;  /* All elements of parallel int are set to 0*/
ap[1] = 1;
```

The result of the **sizeof** operator applied to an array of parallel typed elements is the product of the array length and the **sizeof** operator applied to the element type.

**Example**

```
shape [100]Shape;
int:Shape x[10];

sizeof(x)   /* 10 times the sizeof(int:Shape) */
```

The result of the **sizeof** operator applied to a shape or object of parallel type is the number of bytes in an object of that shape.

**Example**

```
shape *Sptr;
shape [10]Shape;
int:Shape x;

<<Is implicit cast allowed??>>
Sptr = (shape *) malloc(sizeof(shape));

sizeof(Shape)      /* Denotes the size of the shape object
                      Shape */
sizeof(x)          /* Denotes the size of the parallel object
                      x */
```

**Forward References:** the **salloc** function (§4.14.1.9)

### 3.3.3.5 Parallel indexing [NEW]

**Constraints**

The parallel index shall consist of *n* index expressions where *n* is the rank of the parallel expression being indexed. Each index expression shall be of integral or parallel integral type.

All index expressions of parallel type shall be of the same shape.

Index expressions of parallel integral type need not be the same shape as the parallel object being indexed.

> This is an exception to the general rule that parallel operands to binary operations must have the same shape.

The postfix expression shall be of parallel type.

If a parallel-indexed expression is used where a modifiable lvalue is required (such as the left-hand-side of an assignment), only a modifiable lvalue shall be parallel-indexed.

If a parallel expression is parallel-indexed more than once, each set of *n* indices, where *n* is the rank of the parallel expression being indexed, and the indexed expression shall be separately parenthesized. <<Need to add to examples for this case.>>

> This allows the compiler to determine the rank of each of the parallel index expressions.

**Semantics**

A postfix expression, E2, preceded by a parallel index (one or more index expressions in square brackets []), E1, is a parallel-indexed expression. The shape of E2 is herein denoted by S2, and the shape of each of the parallel index expressions in E1, if any exist, is herein denoted by S1.

Note that such an expression is called a parallel-indexed expression even if all of the prefix indices are non-parallel.

If all the index expressions of E1 are of integral type, the parallel-indexed expression designates an individual position of the parallel object being indexed. The type of the parallel-indexed expression is the element type of E2. The result is an lvalue designating the selected element of E2.

**Examples**

```
shape [10]S, [5][5]T;
int:S a;
float:T b;
int i,j;

[8]a;          /* Selects the int at the 8-th position of a */
[i][j]b;       /* Selects the float at the i,j-th position of b
                  */
[5]b;          /* Invalid - improper indexing for rank 2
                  object */
```

If one or more of the index expressions in E1 are of parallel type, any integral index expressions are first promoted to parallel ints of the same shape as the index expressions of parallel type by replicating the integral expression value. The parallel-indexed expression designates a mapping from the shape S2 of the postfix expression E2 to the shape S1 of the parallel index E1. The type of the result is a parallel object of shape S1 whose element type is the element type of E2. The result is a parallel lvalue designating elements of E2 whose selection and order is as directed by the index expressions E1.

<<Need to add text to describe the mapping designated by parallel index expressions on either side of assignment.>>

**Examples**

```
shape [4]S, [4][8]T;
int:S index1, index2;
float:T a;
float:S b;
int:U jtemp;
int i,j;

for (i=0; i<4; i++) {    /* set up index map */
   [i]index1 = 3 - i;
}

index2 = 2;              /* Assigns 2 to all elements of
                            index2 */

[index2]index1;          /* denotes a parallel int of shape
                            S whose values at each position
                            are the value of [2]index1 */

b = [index1][index2]a;   /* assigns some elements of a to b
                            as directed by the parallel ints
                            index1 and index2; equivalent to
```

```
                                        the loop assigning b below */

        for (i=0; i<4; i++)
          [i]b = [[i]index1][[i]index2]a;

        [index1][index2]a = b;   /* assigns the elements of b into a
                                    as directed by the parallel ints
                                    index1 and index2; equivalent to
                                    the loop assigning to a below */


        for (i=0; i<4; i++)
          [[i]index1][[i]index2]a = [i]b;
```

The context of S also affects which positions are selected in a parallel-indexed expression. There is an implicit contextualization of the resulting parallel int denoted by the parallel index E1.

If a value in the resulting parallel int denoted by the parallel index E1 selects a non-existent position of E2, the behavior is undefined.

It is possible that the parallel int denoted by E1 maps more than one position of E2 into the same position of the parallel result expression; in this case, the result expression will arbitrarily, but reproducibly, receive the value of one of the selected positions of E2.

**Examples**

```
<<TBS:  examples of mapping with collisions, mapping with
                narrowed
  contextualization>>
```

### 3.3.3.6 Unary reduction operators [NEW]

**Constraints**

Unary +=, -=, *=, and /= require their operands to be of parallel arithmetic type. Unary &=, ^=, and |= require their operand to be of parallel integral type. Unary <?= and >?= require their operand to be of parallel scalar type.

**Semantics**

These operators perform a reduction: the specified operation is performed on the operand values, resulting in a nonparallel value.

The result of the += operator is the sum of all the elements of the operand.

The result of the -= operator is the negation of the sum of all the elements of the operand.

The result of the *= operator is the product of all the elements of the operand.

The result of the /= operator is the reciprocal of the product of all the elements of the operand.

The result of the &=, ^=, and |= operators are the bitwise AND, XOR, and OR, respectively, of all the elements of the operand.

The result of the <?= and >?= operators is the minimum and maximum, respectifvely, of all the elements of the operand.

**Examples**

```
shape [1000]Shape;
int:Shape x;
int result;

result = +=x;      /* Assign result to be the sum of all the
                      elements of x */
result = <?=x;     /* Assign result to be the minimum valued
                      element of x */
```

### 3.3.4 Cast Operators [ISO §6.3.4, ANSI §3.3.4]

**Semantics**

*Add:*

If the type name specifies a parallel type and the operand is also of parallel type, the result is a parallel value where the value at each position is the result of the conversion (if any) represented by a cast to the nonparallel type counterpart of the type name. The shape of the parallel type and the operand must be the same; the result is undefined if they are not the same shape.

If the type name specifies a parallel type and the operand is of nonparallel type, the result is a parallel value of the same shape as the parallel type that has the operand value replicated and converted as indicated at each position.

If the type name specifies a nonparallel type and the operand is of parallel type, the result is to arbitrarily select an element of the operand and apply the indicated conversion to the selected element. The method of selecting the operand element is implementation defined.

Note that "arbitrarily" means, "not guaranteed to be deterministic or reproducible or random."

**Examples**

```
<<Need to add examples from §10 of C* ref manual.>>
```

### 3.3.5 Multiplicative Operators [ISO §6.3.5, ANSI §3.3.5]

**Syntax**

*Revise as indicated:*

*multiplicative-expression:*
*cast-expression*
*multiplicative-expression* * *cast-expression*

*multiplicative-expression* / *cast-expression*
*multiplicative-expression* % *cast-expression*
*multiplicative-expression* %% *cast-expression*

**Semantics**

*Add:*

The result of the %% (modulus) operator is the remainder on division of the first operand by the second, but unlike the % operator, the result has the same sign as the first operand. The modulus operator evaluates the following formula to compute "a %% b":

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

**Examples**

```
shape [10][20]Shape;
float:Shape f;
int:Shape i;

f * 2        /* Denotes a parallel float value whose value at
                each position is 2 times the value at the
                corresponding position of f . Note that the
                nonparallel operand 2 will be promoted to a
                parallel int value of shape Shape whose value
                at each position will be 2. The usual
                arithmetic conversions will be applied to
                this parallel int value to convert it to a
                parallel float value. */

i %% 2       /* Denotes a parallel int value of shape Shape
                whose value at each position is the true
                modulus resulting from dividing the values at
                the corresponding positions of i by 2. */
```

## 3.3.6 Additive Operators [ISO §6.3.6, ANSI §3.3.6]

**Constraints**

*Revise to include arrays of (pointers to) parallel types:*

For addition, either both operands shall have arithmetic type; one operand shall be a pointer to an object type and the other shall have integral type; or one operand shall be a pointer to an object type or a parallel object type, and the other shall have integral or parallel integral type.

For subtraction, one of the following shall hold:
- both operands have arithmetic type;
- both operands are pointers to qualified or unqualified versions of compatible object types; or
- the left operand is a pointer to an object type or a parallel object type, and the right operand has integral type. (Decrementing is equivalent to subtracting 1.)

**<<Do we want to define subtraction of parallel pointers?? Will that be part of the parallel pointer edits??>>**

**Semantics**

*Revise to include arrays of (pointers to) parallel types:*

**When an expression that has integral type is added to or subtracted from a pointer, the integral value is converted to the appropriate scaled units to produce a pointer to the appropriate object: the integral value is first multiplied by the size of the object pointed to. The result has the type of the pointer operand. ...**

**Examples**

```
shape [10][20]Shape;
float:Shape f;
double:Shape g;
int:Shape h[100];
shape sarray[10];
shape *sptr = &sarray[5];

f - g       /* Denotes a parallel double value whose value
               at each position is the difference of the
               value at the corresponding position of f  and
               the value at the corresponding position of g.
               Note that the value of f  will be promoted to
               a parallel double value of shape Shape by the
               usual arithmetic conversions prior to
               computing the difference between f and g.*/

*(h+10)     /* Denotes the 10th element of the array h;
               equivalent to h[10]. */

*sarray     /* Denotes the first element of sarray, which
               is a shape. */

*(sarray+5) /* Denotes the 5th element of sarray */

*(sptr-2)   /* Denotes the 3rd element of sarray */
```



### 3.3.7 Bitwise Shift Operators [ISO §6.3.7, ANSI §3.3.7]

*Add:*

**Example**

```
shape [100]Shape;
int:Shape  bits;

bits >> 2   /* Denotes a parallel int value of shape Shape,
               whose value at each position is the value at
               the corresponding position of bits shifted
               right by2. */
```

### 3.3.8 Relational Operators [ISO §6.3.8, ANSI §3.3.8]

**Syntax**

*Revise as indicated:*

*relational-expression:*
*shift-expression*
*relational-expression* < *shift-expression*
*relational-expression* > *shiftexpression*
*relational-expression* <= *shiftexpression*
*relational-expression* >= *shift-expression*
*relational-expression* <? *shift-expression*
*relational-expression* >? *shift-expression*

**Semantics**

*Add:*

**Examples**

```
shape [100]Shape;
int:Shape x, y;

x > 0          /* Denotes a parallel int value of shape Shape
                  whose value at each position is the result (0
                  or 1) of the > operator applied to the value
                  at the corresponding position of x and 0. */

x <? y         /* Denotes a parallel int value of shape Shape
                  whose value at each position is the minimum
                  of the values at the corresponding positions
                  of x and y. */
```



### 3.3.9 Equality Operators [ISO §6.3.9, ANSI §3.3.9]

**Semantics**

*Add:*

**Example**

```
shape [100][100]Shape;
float:Shape x, y;

x == y         /* Denotes a parallel int value of shape Shape
                  whose value at each position is the result (0
                  or 1) of the == operator applied to the
                  values at the corresponding positions of x
                  and y. */
```

**<<Add examples/words for comparing shapes, pointers to shapes, and pointers to parallel objects.>>**

### 3.3.10 Bitwise AND Operator [ISO §6.3.10, ANSI §3.3.10]

**Semantics**

*Add:*

**Example**

```
shape [1000]Shape;
int:Shape x;

x & 0x01     /* Denotes a parallel int value of shape Shape
                whose value at each position is the result of
                the & operator applied to the values at the
                corresponding positions of x  and 0x01.
                Hence, the value at each position is 1 if the
                least significant bit of x at that position
                is 1, and 0 otherwise. */
```

### 3.3.11 Bitwise Exclusive OR Operator [ISO §6.3.11, ANSI §3.3.11]

**Semantics**

*Add:*

**Example**

```
shape [20][20]Shape;
int:Shape x, y;

x ^ y        /* Denotes a parallel int value of shape Shape
                whose value at each position is the result of
                the ^ operator applied to the values at the
                corresponding positions of x and y. */
```

### 3.3.12 Bitwise Inclusive OR Operator [ISO §6.3.12, ANSI §3.3.12]

**Semantics**

*Add:*

**Example**

```
shape [20][20]Shape;
int:Shape x, y;

x | y        /* Denotes a parallel int value of shape Shape
                whose value at each position is the result of
                the | operator applied to the values at the
                corresponding positions of x and y. */
```

### 3.3.13 Logical AND Operator [ISO §6.3.13, ANSI §3.3.13]

**Semantics**

*Add:*

If one of the operands is parallel, after the normal promotions have been performed (see §3.2.3), the first operand (LHS) is evaluated to determine a context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. The result is a parallel value which is 1 in those positions for which the values in the corresponding positions of both the LHS and RHS are non-zero, and 0 in all other positions.

**Example**

```
shape [10][10][100]Shape;
int:Shape x, y;

x && y      /* Denotes a parallel int value of shape Shape
               whose value at each position is the result (0
               or 1) of applying the && operator to the
               values at the corresponding positions of x
               and y.  Note that the && operator does not
               require evaluation of its second operand if
               its first operand has value 0. */
```

### 3.3.14 Logical OR Operator [ISO §6.3.14, ANSI §3.3.14]

**Semantics**

*Add:*

If one of the operands is parallel, after the normal promotions have been performed (see §3.2.3), the first operand (LHS) is evaluated to determine a context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. The result is a parallel value which is 0 in those positions for which the values in the corresponding positions of both the LHS and RHS are 0, and 1 in all other positions.

**Example**

```
shape [10][10][100]Shape;
int:Shape x, y;

x || y      /* Denotes a parallel int value of shape Shape
               whose value at each position is the result (0
               or 1) of applying the || operator to the
               values at the corresponding positions of x
               and y.  Note that the || operator does not
               require evaluation of its second operand if
               its first operand has value 1. */
```

### 3.3.15 Conditional Operator [ISO §6.3.15, ANSI §3.3.15]

**Semantics**

*Add:*

**Example**

```
shape [100]Shape;
```



115

```
int:Shape x;
int y, z;

x < 0 ? -x : x
                /* Denotes a parallel int value whose value at
                   each position is the absolute of the
                   corresponding value at each postion of x.
                   Note that both the true and false parallel
                   operands will be evaluated at each position.
                */

x < y  ? x : z
                /* Denotes a parallel int value whose value at
                   each position is the value at the
                   corresponding value of x, if that value is
                   less than y, and otherwise z. */
```

**<<Need to explain contextualization of second and third subexpressions by the first subexpression.>>**

### 3.3.16 Assignment Operators [ISO §6.3.1, ANSI §3.3.16]

**Syntax**

*Revise as indicated:*

*assignment-expression:*
*conditional-expression*
*unary-expression assignment-operator assignment-expression*

*assignment-operator:* one of

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



**Constraints**

*Add:*

**The assignment operators =, %=, %%=, <<=, and >>= shall not be used with a left operand that is nonparallel and a right operand that is parallel.**

#### 3.3.16.1 Simple assignment [ISO §6.3.16.1, ANSI §3.3.16.1]

**Constraints**

*Revise to allow assignment to shapes and parallel objects:*

**One of the following shall hold:**

**• the left operand is of parallel type T1 and the right operand is of nonparallel type T2, such that an operand having the element type of T1 can be the left operand of simple assignment where the right operand is of type T2;**

**• the left operand is the identifier of a shape variable which is fully unspecified when declared, and the right operand denotes a partially specified or fully specified shape;**

• the left operand is the identifier of a shape variable which is partially specified when declared, and the right operand denotes a fully specified shape having the same rank as the left operand;

...

**Semantics**

*Add:*

If the left operand is of parallel type and the right operand is of nonparallel type, the value of the right operand is replicated to form a parallel value before assignment, including any necessary type conversions, is performed.

If the left operand is the identifier of a shape variable, and the right operand denotes a compatible shape object, the left operand is replaced by the value denoted by the right operand. The type of the left operand becomes the composite type of the two operands.

Note that assignment of shapes involves copying of the shape value.

**Examples**

```
shape [10]S, []T, [][]U, V;
int:S x;

x = 10;      /* 10 is first converted to a parallel int */
V = S;       /* composite type is type of S */
V = T;       /* composite type is type of T */
V = U;       /* composite type is type of U */
T = S;       /* composite type is type of S */
U = S;       /* invalid, incompatible shapes */
T = U;       /* invalid, incompatible shapes */
S = T;       /* invalid, can't assign a shape variable
                that is fully specified when declared */
```



**3.3.16.2 Compound assignment** [ISO §6.3.16.2, ANSI §3.3.16.2]

**Semantics**

*Add:*

**Examples**

```
shape [10][20]S;
int:S x, y;
int result;

result += x;       /* Equivalent to result += +=x */
result -= x;       /* Equivalent to result += -=x */
result <?= x;      /* Equivalent to result = <?= x */
y *= x;            /* Equivalent to y = y * x */
[x]y += x;         /* Equivalent to y = [x]y + x; this
                      may involve collisions (see §3.3.3.5)
                   */
```

### 3.3.17 Comma Operator [ISO §6.3.17, ANSI §3.3.17]

**Semantics**

*Add:*

**Example**

```
shape [10][10]Shape;
int:Shape x;

x++, x          /* Denotes the parallel int value represented by
                   x after it has been postincremented. */
```

## 3.4 CONSTANT EXPRESSIONS [ISO §6.4, ANSI §3.4]

**<< TBS: needs to be modified for parallel constant expressions.>>**

## 3.5 DECLARATIONS [ISO §6.5, ANSI §3.5]

**Syntax**

*Revise as indicated:*

*declaration:*
*declaration-specifiers init-declarator-list*$_{opt}$ ;

*declaration-specifiers:*
*storage-class-specifier shape-specifier*$_{opt}$ *declaration-specifiers*$_{opt}$
*type-specifier shape-specifier*$_{opt}$ *declaration-specifers*$_{opt}$
*type-qualifier shape-specifier*$_{opt}$ *declaration-specifiers*$_{opt}$

**<< I don't know what some of these mean (like "typedef :S a"). We need to make sure they all make sense, and/or add constraints as needed.>>**

*shape-specifier:*
: *shape-expression*

*shape-expression:*
( *expression* )
*identifier*
**physical**
**void**

**<<I deleted shapeof from these since it is no longer a keyword, and we have made it into an ordinary library function. Hence, it would have to be enclosed in parentheses.>>**

**Constraints**

*Add:*

In shape expressions of the form ( *expression* ) used in file scope declarations, the *expression* shall be a *constant-expression*.

Shape expressions of the form **void** shall be used only in function prototypes.

Only one shape specification can be given per declaration (see also §3.5.4.4).

A **void** shape designator can be used to specify a generic shape for parameter and return value types of a function prototype. A **void:void** * type specifier can be used to specify a generic pointer to parallel type in a function prototype.

**Semantics**

*Add:*

A shape specification does not apply to the declaration specifier to which it is attached, rather it applies to the whole encompassing declaration.

<<Does this mean that
typedef:S int a
is the same as
typedef int:S a
??? If so, I think we need to revise the constraints to require a type specifier when a shape specifier is included in a declaration, and I need to add some examples for these. We should explain semantics of void and where it can be used, how unspecified shape gets bound.>>

**Examples**

```
shape [10]S;

typedef:S a;                /* syntax error */
typedef int:S it;           /* type it is a parallel int */
extern int:S b;             /* b is an extern parallel int */
static int:physical c;      /* c is a static parallel int of
                               physical shape */
auto float:S d;             /* d is an auto parallel float */
register int:S e;           /* e is a parallel int */
volatile:S f;               /* syntax error */
int:(shapeof(e)) g;         /* g is a parallel int */
short:S int:S h;            /* invalid, multiple shape
                               specifiers */
void:void generic;          /* generic is a parallel object
                               of unspecified element type and
                               unspecified shape */
int:void x;                 /* x is a parallel int of
                               unspecified shape */
```

## 3.5.2 Type Specifiers [ISO §6.5.2, ANSI §3.5.2]

**Syntax**

*Add:*

*type-specifier:*
...
*shape-type-specifier*

**Constraints**

*Add to the type specifier sets:*

- shape-type-specifier

### 3.5.2.4 Shape type specifiers [NEW]

**Syntax**

*shape-type-specifier:*
**shape** *dimension-and-layout-list*$_{opt}$

*dimension-and-layout-list:*
*dimension-and-layout-specifier dimension-and-layout-list*$_{opt}$

*dimension-and-layout-specifier:*
[ *expression*$_{opt}$ ]
[ *expression* **block** ( *constant-expression* ) ]
[ *expression* **scale** ( *constant-expression* ) ]

**Constraints**

The expression used to specify the dimensions and layout of a shape shall be an integral expression having a value greater than zero; if used at file scope or in an extern or static declaration, the expression shall be a constant integral expression. If any *dimension-and-layout-specifier* in the list specifies a dimension, all the specifiers in the list must specify a dimension.

**Semantics**

As discussed in §3.1.2.5, a shape is a type whose values consist of the following components: rank, dimensions, context, and layout. A shape may be fully unspecified, partially specified, or fully specified, depending on the rank and dimensions specified.

The rank of a shape is determined from the number of dimension-and-layout-specifiers given. The dimension is optionally specified as the expression in a dimension-and-layout-specifier.

The **block** and **scale** specifiers determine the layout or data distribution for parallel operands associated with a shape. The **block** specifier indicates data is to be distributed in blocks of the indicated size. The **scale** specifier indicates that data is to be evenly distributed across memory partitions by computing an optimally-sized block for each partition; each partition receives the indicated relative number of blocks, if the shape is multi-dimensional, or a single block, if the shape is one-dimensional. A **block** specifier equal to the dimension in a given rank indicates no distribution across that axis of an object. If a **block** specifier is omitted or equal to

**zero, the compiler determines an appropriate distribution for the object on the target architecture.**

**Examples**

```
shape [100 block (10)]S;
            /* Indicates objects of shape S are to be
               distributed in blocks of 10 elements per
               partition; each partition will have blocks
               of 10 contiguous elements. */

shape [100 block (100)]T;
            /* Indicates objects of shape T are not to be
               distributed; all elements are in the same
               partition */

shape [100 block (0)]U;
shape [100] V;
            /* Indicates objects of shape U or V are to be
               distributed as determined by the compiler */

shape [100 scale (1)]W;
            /* Indicates objects of shape W are to be
               distributed with 1 block per partition where
               the compiler determines the block size */

shape [100 block (10)][100 block (20)]X;
            /* Indicates objects of shape X are to be
               distributed in blocks of 10x20 elements */

shape [100 scale (1)][100 scale (2)]Y;
            /* Indicates objects of shpae Y are to be
               distributed in equally sized blocks with
               kx2k elements where 2k^2 elements are in
               each block; k will be approximately
               (*sqrt((100x100)/2n) where n is the number
               of partitions */
```



## 3.5.3 Type Qualifiers [ISO §6.5.3, ANSI §3.5.3]

**Syntax**

*Revise as indicated:*

*type-qualifier*:
**const**
**elemental**
**nodal**
**volatile**

**Constraints**

*Add for elemental:*

**The elemental qualifier shall only be used to qualify function return types. The elemental qualifier shall not be used to qualify parallel function return types. The elemental qualifier shall not be used to qualify the return value of a function with parallel parameter types.**

**<< Clarify that elemental qualifies the function type, not really the function return type.>>**

*Add for nodal:*

**The nodal qualifier shall only be used to qualify function return types.**

**Semantics**

*Revise as indicated:*

**The properties associated with const- and volatile-qualified types are meaningful only for expressions that are lvalues. The properties associated with elemental- and nodal-qualified types are meaningful only for expressions that are function calls.**

...

**If the specification of an array type includes a const or volatile type qualifier, the element type is so-qualified, not the array type.**

*Add for elemental functions:*

**A function whose return type is qualified with the elemental qualifier is called an elemental function. See also §3.3.2.2, 3.6.6.4, and 3.7.1.**

**Examples**

```
elemental int f1(int);          /* elemental function
                                   returning int */
int * elemental f2(int);        /* elemental function
                                   returning pointer to int */
elemental int i;                /* invalid, must qualifiy a
                                   function return type */
elemental int * f3(int);        /* invalid, does not qualify
                                   the return type */
```

*Add for nodal functions:*

**A function whose return type is qualified with the nodal qualifier is called a nodal function. See also §3.3.2.2, 3.6.6.4, and 3.7.1.**

**Examples**

```
shape S;

nodal int f1(int:S);            /* nodal function returning
                                   int:physical if called
                                   from non-nodal, and an
                                   int if called from nodal */
nodal int:S f2(int:S);          /* nodal function returning
                                   an int:S */
```

```
nodal int i;                    /* invalid, must qualify a
                                   function return type */
nodal int * f3(int);            /* invalid, does not qualify
                                   the return type */
```

### 3.5.4 Declarators [ISO §6.5.4, ANSI §3.5.4]

**Syntax**

*Revise as indicated:*

*declarator:*
*shape-dimension*$_{opt}$ *pointer*$_{opt}$ *direct-declarator shape-specifier*$_{opt}$

...

*pointer:*
* *type-qualifier-list*$_{opt}$ *shape-specifier*$_{opt}$
* *type-qualifier-list*$_{opt}$ *shape-specifier*$_{opt}$ *pointer*

...

*shape-dimension:*
*shape-dimension*$_{opt}$ *dimension-specifier*

*dimension-specifier*
[ *expression*$_{opt}$ ]

**<<Do we want to allow these anonymous shapes???>>**

**3.5.4.3 Function declarators (including prototypes)**

**Constraints**

*Add for nodal functions:*

**A function declarator for a nodal function shall contain a parameter type list.**

A nodal function must provide a prototype specification of its parameters.

**3.5.4.1 Pointer declarators** [NEW]

**Constraints**

*Add for parallel pointers:*

**T shall not contain a shape type specifier.**

**If, in the declaration "T D1," D1 has the form**

* *type-qualifier-list*$_{opt}$ *shape-specifier* D

and D contains a function declarator, then the corresponding function type must be qualified as elemental.

**Semantics**

*Add for parallel pointers:*

If, in the declaration "T D1," D1 has the form

> * *type-qualifier-list*$_{opt}$ *shape-specifier* D

and T or D contains pointer declarators with shape specifiers, then all those shape specifiers shall denote the same shape, or the behavior is undefined.

<<Needs annotated examples of valid and invalid pointer declarators.>>

### 3.5.4.4 Parallel object declarators [NEW]

Parallel object declarators are declarators that denote parallel objects--i.e., variables of parallel type. There are three syntactic methods for specifying the shape of a parallel object declarator; two are optional parts of the declarator syntax (§3.5.4) and the other one is part of the declaration-specifier syntax (§3.5).

**Constraints**

The element type of a parallel object declarator shall not be, directly or indirectly, another parallel type.

The element type of a parallel object declarator shall not be, directly or indirectly, a struct or union type that has a member that has parallel type or a member that has shape type.

The element type of a parallel object declarator shall not be, directly or indirectly, a struct or union type that has a member that has pointer type.

The element type of a parallel object declarator shall not be, directly or indirectly, a shape type.

Only one shape specification can be given per declaration (see also §3.5).

**Semantics**

A shape specifier following a declarator applies only to that declarator.

Shape specifiers can be part of the type specifier or of the declarator of a parallel object declaration. A shape specifier that is part of the type specifier applies to all declarators in that declaration, but a shape specifier that is part of a declarator only applies to that declarator. Note: only one shape specifier per declaration is allowed (see §3.5).

**Examples**

```
shape [100][4][25]S;
typedef int:S T;
```

```
int:S x, y, z;      /* All of x, y, and z are parallel ints */
int a, b:S, c;      /* Only b is a parallel int */
T:S d;              /* Invalid; element type is parallel T */
struct { T e; }:S w;
                    /* Invalid; element type is struct with
                       parallel member */
struct { int *ip; }:S v;
                    /* Invalid; element type is struct with
                       pointer member */
int [10] q;         /* Declares a parallel int of anonymous
                       shape of rank 1 having 10 positions */
<<Do we want to allow these anonymous shapes???>>
```

### 3.5.5 Type names [ISO §6.5.5, ANSI §3.5.5]

**Syntax**

*Revise as indicated:*

*abstract-declarator:*
*pointer*
*shape-dimension*$_{opt}$ *pointer*$_{opt}$ *direct-abstract-declarator shape-specifier*$_{opt}$

### 3.5.7 Initialization [ISO §6.5.7, ANSI §3.5.7]

**Constraints**

**<<Do we need to add anything here?>>**

**Semantics**

*Add for parallel object declarators:*

**An initializer for a parallel object declarator specifies the initial value of every element of the parallel object. The initializer value is replicated and assigned to the element object values at each position.**

**An initializer for an array whose element type is a parallel type initializes each element of the parallel object with the initializer value specified for the corresponding array element; that is, each array element is initialized by replicating the corresponding (nonparallel) value in the initializer list.**

**Examples**

```
shape [200]S;

int:S x = 10;       /* Initializes all int elements to 10 */
int:S a[5] = { 9, 4, 6, 2, 5 };
                    /* Initializes a[0] to be a parallel int
                       whose values are all 9, a[1] to be
                       parallel int whose values are all 4,
                       etc. */
struct { int i; float f; }:S ss = { 10, 1.5};
                    /* Initializes all elements to have member
                       i initialized to 10 and member f
```





125

```
                                initialized to 1.5 */
        union { int i; float f; }:S uu = { 1.0 };
                              /* Initializes member i of all elements
                                 to be 1 */
```

## 3.6 STATEMENTS [ISO §6.6, ANSI §3.6]

**Syntax**

*Revise as indicated:*

*statement:*
...
*contextualization-statement*

**<<Need to revise constraints and semantics section for all statements.>>**

### 3.6.6.4 The return statement [ISO §6.6.6.4, ANSI §3.6.6.4]

**Semantics**

*Add for elemental functions:*

When an elemental function that is executed non-elementally executes a **return** statement, the semantics are the same as a return for a non-elemental function. When an elemental function that is executed elementally executes a return to an elemental caller, the semantics are the same as a return for a non-elemental function.

An elemental function that is executed elementally executes as if the function was called once per active position for the established shape of its parallel argument(s). When a **return** statement with an expression is executed at a position, and the caller is an non-elemental function, the value of the expression is assigned to that position in a parallel return value. If the element type of the parallel return value is different from the type of the return expression, then the expression value is converted as if by assignement. Control is not returned to the caller until all active positions have executed a **return** statement.

*Add for nodal functions:*

When a nodal function that is declared to return a nonparallel value is returning to a non-nodal execution environment, then a parallel value of the **physical** shape is returned. Each thread executing the nodal function contributes one element of the return value.

When a nodal function that is declared to return a nonparallel value is returning to a nodal execution environment, then a nonparallel value is returned in the usual manner.

For a nodal function that is declared to return a parallel value, the shape of the return expression shall be a shape that was created when passing an argument to the nodal function (see §3.3.2.2); otherwise the behavior is undefined. <<I think this needs more explanation!>> If S is the shape of the return expression and T is the

shape of the argument that caused S to be created, then the shape of the parallel value returned to the caller will be T. << I think this is only if returning to a non-nodal environment; what happens if returning to a nodal environment?>> The correspondence of positions of the shape of the return expression and positions in the shape of the value returned to the caller is implementation defined, but the mapping used will be the inverse of the mapping used when passing arguments of the latter shape to a nodal function.

In all cases, when a nodal function executes a return statement with an expression, the expression value is converted as if by assignment if the type of the return expression is different than the declared return type. <<Does this imply implicit shape casting??>>

In addition, if one or more of the threads executes a return statement without an expression, and the value of the function call is used by the caller, the behavior is undefined.

This is the multi-threaded version of the single-threaded semantics for C.

The threads executing a nodal function synchronize upon return to a non-nodal execution environment: control is not returned to the caller until all threads have executed a return.

<<Need examples here for both elemental and nodal functions!>>

### 3.6.7 Contextualization [NEW]

**Syntax**

*contextualization-statement:*
**where** ( *mask-expression* ) *statement*
**where** ( *mask-expression* ) *statement* **else** *statement*
**everywhere** ( *shape-expression* ) *statement*

**Semantics**

A contextualization statement modifies the context of a shape for the duration of the substatement(s) of the statement. The context component of the shape is restored to its previous value at the end of the substatement. Note that context-modifying statements may be nested, and the effects on the contextualization are recursively defined for the nested statements.

<<Need to specify the effect of context-modifying constructs on function calls when a parallel operand is passed as an argument. Is its shape with the modified context implicitly passed along with the operand, or must it be explicitly passed? Need to specify for both elemental and nonelemental functions. The following was hidden text, made public again for discussion.>>

If a nonelemental function is called with a parallel operand as an argument from within the substatement of a context-modifying statement or a subexpression of a context-modifying expression, and the parallel argument is the same shape as that being modified, the shape must also be passed as an argument to the function in order to extend the effect of the context-modification to the function body.

### 3.6.7.1 The where statement [NEW]

**Constraints**

The mask expression shall evaluate to a parallel scalar value of shape *S*.

The mask expression must be a parallel type whose element type is a scalar, which is either an arithmetic type or a pointer type.

**Semantics**

The context of shape *S* will be constrained by the value of the mask expression for the duration of the first substatement, and if there is an **else** and a second substatement, the context of shape *S* will be constrained by the value of the logical complement of the value of the mask expression for the duration of the second substatement. The mask expression (or its complement) constrains the context by further limiting what positions are active: it can only make active positions inactive for the duration of the substatement(s); it cannot make inactive positions active.

Each parallel operand of shape *S* accessed within the substatement(s) of the **where** statement will have the context narrowed by the mask expression. Parallel operands that are not of shape *S* within the substatement(s) are not affected by the narrowed context.

**Examples**

```
shape [100]S, [10][10]T;
int:S x, y, mask;
int:T z;

where (mask > 0) {
   x++;              /* Affected by the narrowed context */
   y++;              /* Affected by the narrowed context */
   z++;              /* Not affected by the narrowed context */
}
else {
   x++;              /* Affected by the narrowed context */
   y++;              /* Affected by the narrowed context */
   z--;              /* Not affected by the narrowed context */
}
```

### 3.6.7.2 The everywhere statement [NEW]

**Constraints**

The shape expression shall evaluate to a pointer to a shape *S*.

**Semantics**

The context of the designated shape *S* will be as if assigned a parallel value of 1 for the duration of the substatement. This will widen the context so that all positions are active.

Each parallel operand of shape *S* within the substatement of the everywhere statement will have a widened context in which all positions are active. Parallel operands that are not of shape *S* are not affected by the widened context.

**Examples**

```
shape [100]S, [10][10]T;
int:S x, y, mask;
int:T z;
shape *p = &S;

everwhere (p) {
   x++;              /* Affected by the widened context */
   y++;              /* Affected by the widened context */
   z--;              /* Not affected by the widened context */
}
```

## 3.7 EXTERNAL DEFINITIONS [ISO §6.7, ANSI §3.7]

### 3.7.1 Function Definitions [ISO §6.7.1, ANSI §3.7.1]

**Constraints**

*Add for elemental functions:*

An elemental function shall not contain any parameter or variable declarations of parallel or shape types, or types derived from parallel or shape types.

An elemental function shall not be declared to return a parallel value.

An elemental function shall not contain a **where** or **everywhere** statement.

An elemental function shall not contain reduction or parallel-index operations.

An elemental function shall only call elemental functions.

An elemental function shall not reference a parallel variable with file scope.

A variable declared within an elemental function shall not have static storage class.

*Add for nodal functions:*

A nodal function may only call nodal functions and elemental functions.

A nodal function may not reference any identifier with file scope.

**Semantics**

*Add for elemental functions:*

An elemental function that is executed non-elementally has the same semantics as a non-elemental function.

An elemental function that is executed elementally executes as if the function was called once per active position of the shape of its parallel argument(s). Assignment of arguments to parameters also occurs on a per active position basis, with conversion, if necessary, between the element type of a parallel argument and the type of the corresponding parameter.

If an elemental function writes to a nonparallel object with static storage duration, the behavior is undefined.

*Add for nodal functions:*

The body of a nodal function executes in a single-node environment (see §3.3.2.2).

If a pointer to a nonparallel object created in a non-nodal execution environment is dereferenced <<in a nodal environment?>>, then the behavior is undefined.

<<Examples for both elemental and nodal functions would be useful.>>

<<Need any extra words about parallel parameters to non-elemental functions?>>

# 4. LIBRARY [ISO §7, ANSI §4]

## 4.1 INTRODUCTION [ISO §7.1, ANSI §4.1]

### 4.1.2 Standard Headers [ISO §7.1.2, ANSI §4.1.2]

*Add to standard headers:*

<dpce.h>

This header file defines the predefined shape identifier **physical**, and declares new functions that are available for implementations supporting DPCE. In addition, it declares specific functions declared in <math.h>, <stdlib.h>, and <string.h> to be elemental functions; if these header files are also included, the correct composite type will be formed for these functions for DPCE implementations (see §3.1.2.6). This header file must be included by all source files that use DPCE.

### 4.1.6 Use of Library Functions [ISO §7.1.6, ANSI §4.1.6]

<<Add any special considerations for elemental and nodal functions??>>

## 4.14 DATA PARALLEL UTILITIES <dpce.h> [NEW]

The header <dpce.h> defines the predefined shape identifier **physical** and declares the functions specified below.

The exact definition of **physical** is implementation specific. The following definition is a placeholder for that specific definition, which must be fully specified.

```
shape []physical;
```

### 4.14.1 General purpose DPCE utilities [NEW]

There are four functions from <stdlib.h> that are redeclared here as elemental functions. The intent is that the following functions will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.

```
elemental int abs(int j);
elemental int atoi(const char *nptr);
elemental long int atol(const char *nptr);
void qsort(void *base, size_t nmemb, size_t size,
           elemental int (*compar)(const void *, const void *));
```

There are eleven new general purpose utility functions declared as follows.

### 4.14.1.1 The dimof() function [NEW]

**Synopsis**

```
#include <dpce.h>
int dimof(shape s, int axis);
```

**Description**

The **dimof** function extracts the dimension of the shape s in the specified axis.

**Returns**

The **dimof** function returns the extracted dimension.

**Examples**

```
shape [10][20][30]S;
int dim0, dim1, dim2;

dim0 = dimof(S,0);      /* Assigns dim0 to be 10 */
dim1 = dimof(S,1);      /* Assigns dim1 to be 20 */
dim2 = dimof(S,2);      /* Assigns dim2 to be 30 */
```

### 4.14.1.2 The palloc() function [NEW]

**Synopsis**

```
#include <dpce.h>
void:void *palloc(shape *s, int bsize);
```

**Description**

The **palloc** function allocates space for a parallel object of shape s and having element size of bsize bytes.

The shape s must be fully specified when **palloc** is called; if it is not fully specified, the behavior is undefined.

**Returns**

The **palloc** function returns a null pointer or a pointer to the allocated space.

**Examples**

```
shape [20][20]S, [N][M]T;
shape U;
int:S *x;

x = palloc(S, sizeof(*x));    /* Allocates space for a 20x20
                                 parallel int */
x = palloc(T, sizeof(*x));    /* Allocates space for an NxM
                                 parallel int */
x = palloc(U, sizeof(*x));    /* Undefined */
```

### 4.14.1.3 The pcoord() function [NEW]

**Synopsis**

```
#include <dpce.h>
int:void pcoord(int axis, shape s);
```

**Description**

The **pcoord** function is a parallel axis-coordinate value constructor. For a given axis specifier (0, 1, etc.) and a given shape s, this function returns a parallel int of shape s whose value at each position is intialized to the coordinate at the position in the given axis. It is an error to specify an axis number that is outside the range 0..(*n*-1) for a shape of rank *n*.

**Returns**

The **pcoord** function returns a parallel int of the same shape as s.

**Examples**

```
shape [10]S, [2][2]T;
int:S x;
int:T y;

x = pcoord(0,S);  /* parallel int of shape S having value 0
                     in position [0], 1 in position [1],
                     ..., and 9 in position [9]. */
y = pcoord(1,T);  /* parallel int of shape T having value 0
                     in positions [0,0] and [1,0], and
                     value 1 in positions [0,1] and [1,1].
                  */
pcoord(3,T);      /* invalid, axis 3 out of range */
```

### 4.14.1.4 The pfree() function [NEW]

**Synopsis**

```
#include <dpce.h>
void pfree(void:void *pptr)
```

**Description**

The **pfree** function causes the space pointed to by pptr to be deallocated. If pptr is a null pointer, no action occurs. If pptr does not match a pointer returned earlier by the **palloc** function, or if the space has already been deallocated by an earlier call to **pfree**, the behavior is undefined.

**Returns**

The **pfree** function does not return a value.

**Examples**

```
shape [20][20]S;
int:S *x, y;

x = palloc(&S, sizeof(*x));    /* Allocates space */
pfree(x);                      /* Deallocates space */
pfree(&y);                     /* Undefined */
pfree(x);                      /* Undefined */
```

### 4.14.1.5 The positionsof() function [NEW]

**Synopsis**

```
#include <dpce.h>
int positionsof(shape s);
```

**Description**

The **positionsof** function computes the number of positions in any parallel object of shape s, which is the product of all the dimensions of the shape.

**Returns**

The **positionsof** function returns the computed number of positions.

**Examples**

```
shape [10][30]S;
int n;

n = positionsof(S);      /* Returns 300 */
```

### 4.14.1.6 The prand() function [NEW]

**Synopsis**

```
#include <dpce.h>
int:void prand(shape s);
```

**Description**

The **prand** function provides a parallel version of the **rand** function. It constructs a parallel int of the same shape as s.

**Returns**

The **prand** function returns a parallel int of the same shape as s.

#### 4.14.1.7 The psrand() function [NEW]

**Synopsis**

```
#include <dpce.h>
void psrand(unsigned seed);
```

**Description**

The **psrand** function provides the same functionality with respect to the **prand** function as the **srand** function does with respect to the **rand** function.

**Returns**

The **psrand** function does not return a value.

#### 4.14.1.8 The rankof() function [NEW]

**Synopsis**

```
#include <dpce.h>
int rankof(shape s);
```

**Description**

The **rankof** function extracts the rank component value of the shape s.

**Returns**

The **rankof** function returns the extracted rank value.

**Examples**

```
shape [10][50]S;
int rank;

rank = rankof(S); /* returns rank of shape S == 2 */
```

#### 4.14.1.9 The salloc() function [NEW]

**Synopsis**

```
#include <dpce.h>
shape *salloc(int rank, int *dimensions,
              int *block)
```

**Description**

The **salloc** function allocates space for a shape object of the specified rank and dimensions, initializes the context so that all positions are active, and initializes the layout for the shape object as specified by the block argument. The rank argument

must be an integer greater than 0. The dimensions must be an integer array of *n* values, where *n* is the specified rank; each dimension shall be an integer greater than 0. The block argument is either a null pointer or an array of *n* block specifications for the shape to be allocated; each block specification shall be an integer greater than or equal to 0. If these specifications are not met, the behavior is undefined.

**Returns**

The **salloc** function returns a pointer to the allocated, initialized space.

**Examples**

```
shape *sp;
shape U;
shape []P;
int ten = 10;
int rank = 1000;
int dims[3] = {50,50,20};

sp = salloc(0, 0, 0);          /* Undefined */
sp = salloc(1, &ten, 0);       /* allocate 1-dimensional
                                  shape object with default
                                  distribution */

P = *salloc(1, &rank, &ten);   /* allocate 1-dimensional
                                  shape object with blocks of
                                  10 elements per partition;
                                  makes P a fully specified
                                  shape */

U = *salloc(3, dims, dims);    /* allocate 3-dimensional
                                  shape object with no
                                  distribution; makes U a
                                  fully specified shape */
```



### 4.14.1.10 Thesfree() function [NEW]

**Synopsis**

```
#include <dpce.h>
void sfree(shape *sptr);
```

**Description**

The **sfree** function deallocates the space pointed to by sptr. If sptr is a null pointer, no action occurs. If sptr does not match a pointer returned earlier by **salloc**, or if the sapce has been deallocated by an earlier call to **sfree**, the behavior is undefined.

**Returns**

The **sfree** function does not return a value.

**Examples**

```
shape *sp, S;
int dims[] = { 2, 2};

sp = salloc(2, dims, 0);       /* Allocate space */
```

```
S = *sp;                    /* Copy space, making S fully
                               specified */
sfree(sp);                  /* Free allocated space */
sfree(&S);                  /* Undefined */
sfree(sp);                  /* Undefined */
```

#### 4.14.1.11 Theshapeof() function [NEW]

**Synopsis**

```
#include <dpce.h>
shape shapeof(void:void pvar);
```

**Description**

The **shapeof** function extracts the shape of the generic parallel variable pvar.

**Returns**

The **shapeof** function returns a shape value which is equivalent to the shape of pvar.

**Examples**

```
shape [10]S;
int:S x;
shape sp;

sp = shapeof(x);  /* sp is assigned the same shape as S */
```

### 4.14.2 DPCE mathematics [NEW]

The following functions from <math.h> are redeclared here as elemental functions. The intent is that they will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.

```
elemental double acos(double x);
elemental double asin(double x);
elemental double atan(double x);
elemental double atan2(double y, double x);
elemental double cos(double x);
elemental double sin(double x);
elemental double tan(double x);
elemental double cosh(double x);
elemental double sinh(double x);
elemental double tanh(double x);
elemental double exp(double x);
elemental double frexp(double value, int *exp);
elemental double ldexp(double x, int exp);
elemental double log(double x);
elemental double log10(double x);
elemental double modf(double value, double *iptr);
elemental double pow(double x, double y);
elemental double sqrt(double x);
elemental double ceil(double x);
elemental double fabs(double x);
```

```
elemental double floor(double x);
elemental double fmod(double x, double y);
```

### 4.14.3 DPCE string handling [NEW]

**The following functions from <string.h> are redeclared here as elemental functions. The intent is that they will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.**

```
elemental void *memcpy(void *s1, const void *s2, size_t n);
elemental void *memmove(void *s1, const void *s2, size_t n);
elemental int memcmp(const void *s1, const void *s2, size_t n);
elemental void *memset(void *s, int c, size_t n);
```

# APPENDICES

## A. OTHER PROPOSED EXTENSIONS

There were numerous other extensions that the committee considered and, for the reasons cited, decided not to include at the current time. References to the text of the actual proposals are given for each category of extension in the subsections of this appendix.

### A.1 PARALLEL CONTROL FLOW CONSTRUCTS

### A.2 FORALL STATEMENT

### A.3 ITERATORS

### A.4 BIT TYPES

### A.5 I/O LIBRARY

### A.6 DYNAMIC LAYOUT

### A.7 ARRAYS AS FIRST CLASS OBJECTS

### A.8 OVERLOADING

### A.9 CURRENT SHAPE

<<TBS>>

# INDEX