Data Parallel C Extensions

Numerical C Extensions Group of X3J11
DPCE Subcommittee
Technical Report, Version 1.6
X3J11/94-080
WG14/N395
December 31, 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 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 subcommittee began by adopting a general model that included the basic concept of parallel data aggregates that have structure (rank and dimensions), 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, pointers to parallel, and parallel pointers.

The process has been a lengthy one, and the culmination of the subcommittee'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
David Sarnoff Research Center, Maya Gokhale
Digital Equipment Corporation, Randy Meyers
Digital Equipment Corporation, Jeffrey Zeeb
Farance Inc, Frank Farance
Hewlett Packard, John Kwan
HyperParallel Technologies, Christian Fortunel
HyperParallel Technologies, Nicolas Paris
IBM, Pawel Molenda
IBM, Bill O'Farrell
Keaton Consulting, David Keaton
Lawrence Livermore National Laboratory, Linda Stanberry
MasPar, David Alpern
Mimosa Systems, Incorporated, Hugh Redelmeier
Open Software Foundation, Mike Meissner
Pacific-Sierra Research, David McNamara
Plum Hall, Tom Plum
SunPro, Bob Jervis
SunPro, Marino Segnan
Supercomputing Research Center, Howard Gordon
Supercomputing Research Center, Aaron Naiman
Syracuse University, Pankaj Kumar
Syracuse University, Nancy McCracken
Thinking Machines Corporation, James L. Frankel
Thinking Machines Corporation, Gary Sabot
Thinking Machines Corporation, Guy Steele
Tydeman Consulting, Fred Tydeman
University of Arizona, Peter Bigot
University of New Hampshire, Phil Hatcher
Unix System Laboratories, David Prosser
US Army, Doug Gwyn

Data Parallel C Extensions

Linda Stanberry, Technical Editor
Lawrence Livermore National Laboratory
PO Box 808, L-300
Livermore, CA 94551
lstanberry@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 supports programming of data parallel applications. The intent is to provide a set of machine-independent extensions that permit an efficient mapping to high-performance archictectures, especially massively parallel architectures.

1.2 SCOPE [ISO §1, ANSI §1.2]

This document describes 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).

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 Functions," 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).

31. "dbC Reference Manual," J.D. Schlesinger and M. Gokhale (Technical Report SRC-TR-93-109, November, 1993).

32. "Adding Shapes to Iterators," Bill Homer (X3J11/94-021).

33. "DPCE Array Slicing Proposal," Frank Farance (X3J11/94-025).

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.

> 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 subsection of the Standard, not replacing or modifying 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:

Revise:

These headings are used for clarity as needed, and omitted where it is obvious that the entire change is reflected in the text. For brevity, an ellipsis (...) is used to indicate that omitted text is the same as in the Standard.

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 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 component of a shape that designates the active positions of a parallel operand.

- dimension – the number of positions along an axis 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.

- node – an actual processing unit in the execution environment.

  A node usually refers to a hardware processing unit, but is implementation dependent to allow for diverse parallel environments.

- 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 pointer – a parallel operand whose element type is pointer type.

- 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.

- pointer to parallel – a pointer type whose referenced type is a parallel type.

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

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

- rank – the number of dimensions or axes in a 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, layout, and context.

  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 §3.5.4.4.

1.7 COMPLIANCE [ISO §4, ANSI §1.7]

In order to comply with this set of extensions, an implementation shall 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 portability 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. In addition to the single thread model, DPCE also offers two mechanisms which allow the programmer to directly utilize a multithreaded model: nodal 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.
>
> 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.

Examples

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 ISO C with arrays. The ISO C example is not truly equivalent to the DPCE example since the DPCE operations are not ordered as the operations are ordered in the ISO C loops. Further, in DPCE, operations are performed under context and the granularity is at the operation level rather than at the statement level. Although the two code segments are not equivalent for the reasons noted, both demonstrate the same effect.

```
/* DPCE */                  /* ISO C */
shape [100]S;               typedef int ShapedInt[100];
int:S x, y, z;              ShapedInt x, y, z;
                            int i;

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

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

At program startup, the DPCE execution environment is established by the inclusion of the header file **`<dpce.h>`**. This environment shall define **`physical`** to denote a predefined variable of type shape which is of rank 1 and dimension equal to the number of node s in the execution environment. The layout of **`physical`** is implementation-defined.

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 §4.14 of this document.

Add the following new keywords:

```
block
elemental
everywhere
nodal
scale
shape
shapeof
where
```

3.1.2.5 Types [ISO §6.1.2.5, ANSI §3.1.2.5]

Revise object types and incomplete types to include parallel objects:

... Types are partitioned into *object types* (types that describe objects or parallel objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects or parallel objects but lack information needed to determine their sizes).

Add shape type:

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

The shape type is an object type whose values consist of the following components: rank, dimensions, layout, and context. A value of type shape is referred to as "a shape."

A shape type 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 shape types 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 (see §3.5.4.4).

A **void** shape designator can be used to specify a generic shape type. See §3.5 and §3.3.2.2.

Examples

```
shape S;                  /* Fully unspecified shape */
shape []T;                /* Partially specified shape */
shape [100]U;             /* Fully specified shape */
shape [2][]V;             /* Constraint violation */

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

<u>*Revise derived types to include parallel types:*</u>

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. If its element type is *T* and its shape is *S*, the parallel type is called "a parallel *T*" or "a parallel *T* of shape *S*." The construction of a parallel type from an element type and a shape is called "parallel type derivation."

  For example, a parallel type with element type `int` and shape S is called "a parallel `int`" or "a parallel `int` of shape S." A parallel type with element type `struct` is called "a parallel `struct`." A parallel type with pointer element type is called "a parallel pointer."

<u>*Revise recursive applicability of constructing derived types:*</u>

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

- the element type of a parallel type shall not be or contain a parallel type or shape type;

- a structure or union type shall not contain a member that has parallel type or a member that is a shape;

  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 structs or unions or functions or pointers. But you can't have parallel types whose elements are parallel types or contain parallel types, or are shape types or contain shape types. Nor can you have parallel types whose elements are array types (see §3.5).

  Note that the element type of a parallel type may be or contain a pointer to parallel type, as a pointer to parallel is not a parallel type. Hence, a parallel pointer may have element type "pointer to parallel."

<u>*Revise integral and floating types to include parallel types:*</u>

Parallel types whose element types are integral types are called *parallel integral types*. Parallel types whose element types are floating types are called *parallel floating types*.

<u>*Revise arithmetic and aggregate types to include parallel types:*</u>

Parallel integral and floating types are collectively called *parallel arithmetic types.* Array, structure, and parallel types are collectively called *aggregate types.*

<u>*Revise pointer storage requirements;*</u>

A pointer to a parallel type need not have the same representation as a pointer to the corresponding nonparallel type. The equivalent for parallel types of a pointer to **void** is a pointer to **void:void**.

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

Add for compatible shape types:

Two shape types are compatible under the following conditions:

- A fully unspecified shape type is compatible with any other shape type.
- Two partially specified shapes are compatible only if they specify the same rank.
- A partially specified shape type is compatible with a fully specified shape type if the partially specified shape type has the same rank as the fully specified shape type.
- Two fully specified shape types are compatible only if they specify the same rank, dimensions, and layout (see §3.2.3).
- A **void** shape type is compatible with any other shape type.

Examples

```
/* Compatible and incompatible shape types */
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 */
```

Add for compatible parallel types:

Two parallel types are compatible if they are derived from compatible element types and compatible shape types.

Revise for qualifiers in composite function types and 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 satisfies the following conditions:

...

- If one type is a qualified function type and the other is an unqualified function type, the composite type is the qualified function type.

> To form a composite function type from an **`elemental`**-qualified or **`nodal`**-qualified function type and an unqualified function type, the unqualified versions of the function types must be compatible, and the composite type will be the qualified function type. This allows the inclusion of **`<dpce.h>`** to coexist with the unqualified declarations of the standard C library functions that it redeclares as **`elemental`**.

Examples

```
void *memset();
#include <string.h>
#include <dpce.h>         /* Provides elemental definitions */
#include <math.h>
...
{
  shape [100][100]S;
  double x;
  double:S y;

  ... sin(x) ...          /* Invokes nonparallel version of sin */

  ... sin(y) ...          /* Invokes sin elementally for each
                             element of y */

  memset(&x,0,sizeof(x));
                          /* Invokes nonparallel version of
                             memset */

  memset((double:S *:S)&y,0,sizeof(y));
                          /* Invokes memset elementally for each
                             element of y */
}
```

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

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

Examples

```
shape [10]S;
shape T;                  /* Composite type of S and T is S */
shape []U;                /* Composite type of S and U is S */
                          /* Composite type of T and U is U */
void f(int:void x) { ... }
int:S y;
f(y);                     /* Composite type of void and S is S */
```

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 Arithmetic Operands [ISO §6.2.1, ANSI §3.2.1]

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.

3.2.2 Other Operands [ISO §6.2.2, ANSI §3.2.2]

3.2.2.1 Lvalues and Function Designators [ISO §6.2.2.1, ANSI §3.2.2.1]

Add for parallel lvalues:

A *parallel lvalue* is an lvalue that designates a parallel object.

3.2.2.3 Pointers [ISO §6.2.2.3, ANSI §3.2.2.3]

Revise:

A pointer to **`void`** may be converted to or from a pointer to any incomplete nonparallel type or nonparallel object type. A pointer to any incomplete nonparallel type or nonparallel object type may be converted to a pointer to **`void`** and back again; the result shall compare equal to the original pointer.

Add:

A pointer to **`void:void`** may be converted to or from a pointer to any incomplete parallel type or parallel object type. A pointer to any incomplete parallel type or parallel object type may be converted to a pointer to **`void:void`** and back again; the result shall compare equal to the original pointer.

A pointer to an array of, or containing, any incomplete parallel type or parallel object type may be converted to a pointer to **`void:void`** and back again; the result shall compare equal to the original pointer.

> Since a pointer to an array is also a pointer to the first element of the array, a pointer to an array of parallel is also a pointer to parallel.

A null pointer constant cast to type **`void:void *`** is called a *null pointer-to-parallel constant*. If a null pointer constant or a null pointer-to-parallel constant is assigned to or compared for equality to a pointer to parallel, the constant is converted to a pointer to parallel of that type. Such a pointer, called a *null pointer to parallel* is guaranteed to compare unequal to a pointer to any parallel object.

Two null pointers to parallel, converted through possibly different sequences of casts to pointers to parallel types, shall compare equal.

3.2.3 Parallel Operands and Contextualization [NEW]

In general, in binary operations involving operands of parallel types, there are both compile-time and run-time requirements. At compile time, the shape types of the operands shall be compatible (see §3.1.2.6). At run time, the shapes of the operands shall be fully assigned (see §3.5.4.4) and their values shall be the same; otherwise the behavior is undefined. That is, for parallel operands a and b,

```
if (*shapeof(a) == *shapeof(b))
            /* a op b is defined */
else
            /* a op b is undefined */
```

Two shapes are the same only if their values are identical; that is, only if they have the same rank, dimensions, layout and context. The following function returns 1 if the two shapes **S1** and **S2** are the same, and 0 otherwise.

```
int same_shape(shape S1, shape S2)
{
  int rank;
  int i, j;
  int:S1 context1;
  int:S2 context2;
  int same_context;

  rank = rankof(S1);
  if (rank != rankof(S2)) return(0);

  for (i=0; i<rank; i++) {
    if (dimof(S1,i) != dimof(S2,i)) return(0);
    if (layoutof(S1,i) != layoutof(S2,i)) return(0);
  }

  everywhere (S1)
    context1 = 0;
  context1 = 1;
  everywhere (S2)
    context2 = 0;
  context2 = 1;
  everywhere(S1)
    everywhere(S2)
      same_context = &= (context1 == context2);

  return(same_context);
}
```

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 types of the two shapes are compatible. No requirement is placed on the generated code to perform run-time tests that shapes are equivalent; the user must guarantee the equivalence of shapes at run time to avoid undefined behavior. See §A.10 for compiler options for shape equivalence testing.

The shape type of the result of an operation is the composite shape type of the operands. The shape of the result of an operation is the shape of the operands.

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 applies only 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, where a position is indicated as active by a nonzero 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 initially all nonzero, indicating that all positions are active.

The context of the shape associated with a parallel operation 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]

Add for sliced expressions:

An expression that contains an operand that is a sliced expression (see §3.3.3.6) shall contain only operands that are sliced expressions or are of nonparallel type.

3.3.1 Primary Expressions [ISO §6.3.1, ANSI §3.3.1]

Revise as indicated:

Syntax

primary-expression:
    identifier
    .
    constant
    string-literal
    (*expression* **)**

Constraints

A '**.**' shall only be used in a *parallel-index-expression* (see §3.3.3.6).

Semantics

Add:

The type of the expression '**.**' is the parallel **int** equivalent to **pcoord(s,a)**, where **a** is the axis specifier (0–(n-1)) of the *parallel-index-expression* in which it is used, and **s** is the shape of the nearest enclosing parallel expression being indexed (see §3.3.3.6).

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 §3.3.2.1]

Revise to allow arrays subscripted 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 object *type*," the other expression shall have parallel integral type of shape *S*, and the result has type "parallel operand of type *type* and shape *S*."

Semantics

A postfix expression **`E1`** followed by an expression **`E2`** in square brackets **`[]`** is either a subscripted designation of an element of an array object, or a parallel subscripted designation of elements of an array object.

If **`E1`** or **`E2`** is an integral expression, then the definition of the subscript operator **`[]`** is that **`E1[E2]`** is identical to **`(*(E1 + (E2)))`**. ...

> Integral subscripts with pointers to parallel objects behave as do subscripts with pointers to nonparallel objects.

If **`E1`** or **`E2`** is a parallel integral expression, then **`E1[E2]`** is also identical to **`(*(E1 + (E2)))`**. In this case **`(E1 + (E2))`** denotes adding a parallel **`int`** to a pointer, and results in a parallel pointer. If the pointer expression is a pointer to nonparallel, the result is a parallel pointer to nonparallel, and if the pointer expression is a pointer to parallel, the result is a parallel pointer to parallel. The value at each position of the resulting parallel pointer is the sum of the pointer and the integral value at the corresponding position of the parallel integral expression, as described in §3.3.6. The behavior of dereferencing the resulting parallel pointer has position-oriented semantics as described in §3.3.3.2.

Examples

```
shape [20]S;
shape sarray[5];  /* Array of shapes */
int:S x;          /* Parallel int */
int:S y[50];      /* Array of parallel int */
int:S *z;         /* Pointer to parallel int */
int i;
int iarray[100];  /* Array of int */

sarray[i];        /* Designates the i-th member of sarray, which
                     is a shape */

z = &y[39];       /* z designates a subarray of y */

y[i];             /* Designates the i-th member of y, which is a
                     parallel int of shape S */

z[i];             /* Designates the i-th member of z, which is the
                     (39+i)-th member of y */

y[x];             /* Denotes an int:S, the elements of y
                     designated by the parallel int x */

iarray[x];        /* Denotes the elements of iarray designated by
                     the parallel int x */
```

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

Constraints

Add for elemental and nodal functions:

A value of type *T1* is *elementally assignable* to an object of type *T2* if *T2* is a nonparallel type and one of the following is true:

- a value of the type *T1* may be assigned to an object of the unqualified version of the type *T2*; or
- *T1* is a parallel type and a value of the element type of *T1* may be assigned to an object of the unqualified version of the type *T2*; or
- *T1* and *T2* are pointer types such that *T1* is either *pointer to T1'* or *parallel pointer to T1'*, *T2* is *pointer to T2'*, and a value of type *T1'* is elementally assignable to an object of type *T2'*; or
- *T1* and *T2* are array types with the same array size specifications and a value of the element type of *T1* is elementally assignable to an object of the element type of *T2*; or
- *T1* and *T2* are compatible elemental or nodal function types.

Add for elemental functions:

Each parallel argument to an elemental function shall be elementally assignable to an object 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 (see §3.5.4.3):

- the parameter type is a parallel type of shape `void`, such that an element value of the argument is elementally assignable to an object of the type of an element of the parameter; or
- the parameter type is a pointer to nonparallel type, such that an element value of the argument is elementally assignable to an object of the type pointed to by the parameter; or
- the parameter type is a nonparallel type, such that an element value of the argument is elementally assignable to an object of the type of the parameter.

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

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

Semantics

Add 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, and the shape of the argument must be the same as the shape of the parameter. A parameter or return value declared as having **`void`** shape assumes the shape of the corresponding argument or return expression, respectively.

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

Examples

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

int count_active_positionsof(int:void x)
{
      return(+= (int:(*shapeof(x))) 1);
}

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

int:S add_one(int:S x) {
      return(x + 1);
}

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

b = add_one(a);

where (a > 0) {
    printf("Number of positive elements in a is %d\n",
           count_active_positionsof(a));
    everywhere (S) {
        printf("Total number of elements in a is %d\n",
               count_active_positionsof(a));
    }
}
```

Add for elemental functions:

A function qualified with the **elemental** qualifier is called an *elemental function*. The function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.4.3, §3.6.6.4, and §3.7.1.

An elemental function can be executed either elementally or nonelementally. If none of the arguments are parallel operands, and the calling function is not executing elementally, the function is executed nonelementally (i.e., as a normal function). If one or more of the arguments are parallel operands, or the calling function is executing elementally, then the called function is executed elementally. All the parallel arguments must be of the same shape; otherwise, 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 function 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 nonelemental function.

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

An elemental function that is declared to return a type other than **void** may return either a parallel value or a nonparallel value. If the function is not executing elementally, or if the function is executing elementally and is returning to a function that is executing elementally, then a nonparallel value is returned. If the function is executing elementally and returning to a function that is not executing elementally, then a parallel value of the returning function's established shape is returned.

Examples

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

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

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

f(x,y);      /* Returns parallel int of shape S with sum of
                active positions of x and y; inactive
                positions are unspecified. */

f(x,1);      /* Returns parallel int of shape S 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 S 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 qualified with the **nodal** qualifier is called a *nodal function*: The function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.4.3, §3.6.6.4, and §3.7.1.

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 multithreaded 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 reestablished upon return from the nodal function.

A nonparallel argument to a nodal function is first promoted to a parallel value of **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. A pointer to this shape will be returned when the **`shapeof`** operator is applied to the parallel parameter during execution of the nodal function. 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. The shape object created is derived as follows:

- 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.

If there is more than one parallel argument of the same shape, all being passed to parameters of parallel type, then the same shape value shall be returned by dereferencing the return value of **`shapeof`** applied to any of these parameters; otherwise the behavior is undefined.

Note that parallel arguments are evaluated under context. If a reference is made to an element of a parameter that corresponds to an inactive position of its argument, the behavior is undefined.

For a parallel argument to a nodal function, if the corresponding parameter is of nonparallel 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 **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 **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 element of each thread's parameter receives a pointer to the object stored at the corresponding position of the parallel object pointed to by the argument's pointer value. That is, the argument is a parallel pointer to parallel and each thread's parameter is derived from the argument's value according to the position-oriented semantics described in §3.3.3.2.

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

```
void f1(int:void a, int b[], int *c, int d) nodal;
void f2(int:void *:void a, int *b[], int ** c) nodal;

shape [100]S;
int:S x;
int:physical y;
int i;
int:physical *:physical physical_Pp2Pi;
int:S *:S Pp2Pi;

call_one: f1(x, x, x, y);
      /* For each thread executing f1:  the first parameter
         receives a parallel of a new shape with number of
         positions equal to the number of positions of x
         stored on the node executing the thread; the second
         and third parameters receive arrays with lengths
         equal to the number of positions of x stored on the
         node; the fourth parameter receives a single value
         corresponding to the single position of physical
         stored on the node.  Note that if the shape of x
         has inactive positions, then there will be
         undefined values in the corresponding elements of
         the first three parameters.  That is, only the
         values of the active elements of x are assigned to
         the corresponding elements of the first three
         parameters.  Similarly, if the position of physical
         stored on the node is inactive, the value of the
         fourth parameter will be undefined; otherwise the
         value of y at that position is assigned to the
         fourth parameter. */

call_two: f1(y, y, y, y);
      /* For each thread executing f1:  the first parameter
         receives a parallel with a single element, which is
         the element of y stored on the node executing the
         thread, if that position is active; the second and
         third parameters each receive an array with a
         single element; the fourth parameter is as in
         call_one. */

f1(i, i, i, i);
      /* First, i is promoted to an int:physical for each
         argument.  Then each thread executing f1 gets the
         single value of i in each parameter, if the
         position of physical stored on the node executing
         the thread is active. */

f1(x, x, x, x);
      /* Undefined; x is not of shape physical */
```

```
f1(x, x, physical_Pp2Pi, y);
      /* The first two and the fourth parameters will be as
         in call_one.  For each thread executing f1, the
         third parameter will receive a single value derived
         from the element of the third argument stored on
         the node, if that element is active, and an
         undefined value otherwise.  When this parameter
         value is dereferenced inside f1, the object
         accessed will be the element stored on this node of
         the parallel int pointed to by the argument element
         from which the parameter was derived. */

f2(Pp2Pi, Pp2pi, Pp2Pi);
      /* For each thread executing f2:  each parameter will
         receive pointer values derived from the values of
         Pp2Pi stored on the node executing the thread
         (assuming all elements of Pp2Pi are active).  The
         value of each element of a parameter has position-
         oriented semantics and will reference objects only
         at its corresponding position. */
```

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

Constraints

Revise:

The first operand of the `.` operator shall have a qualified or unqualified, parallel or nonparallel structure or union type, and the second operand shall name a member of that structure or union type.

The first operand of the `->` operator shall have type “pointer to qualified or unqualified, parallel or nonparallel structure” or “pointer to qualified or unqualified, parallel or nonparallel union,” and the second operand shall name a member of the structure or union of the type pointed to.

Semantics

Add:

If the first operand of the `.` or `->` operator is of parallel structure or union 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. If the first operand is an lvalue, the result is an lvalue.

Examples

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

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.

Examples

```
shape [10]S;
int:S x;
int:S y[20];
int:S *p2Pi = &y[10];    /* Points to parallel int which is
                            the 10-th element of y */
int:S *:S Pp2Pi = y;     /* Every element of Pp2Pi points to
                            parallel int y[0] */

x++;                     /* Increments each element of x */

y[i]--;                  /* Decrements each element of y[i] */

p2Pi++;                  /* Increments p2Pi to point to the next
                            parallel int in the array y, namely
                            y[11] */

Pp2Pi++;                 /* Increments each element of Pp2Pi, which
                            would result in each element of Pp2Pi
                            pointing to the next parallel int in
                            the array y, namely y[1] */

[3]Pp2Pi = &y[5];        /* Assigns element 3 of Pp2Pi to point to
                            y[5] */

Pp2Pi++;                 /* Increments each element of Pp2Pi; now
                            all elements point to y[2], except
                            [3]Pp2Pi points to y[6] */
```

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* **)**
  shapeof (*parallel-variable-identifier* **)**
  parallel-index postfix-expression
  reduction-operator postfix-expression

parallel-variable-identifier:
  identifier

parallel-index:
  [*index-or-slice* **]** *parallel-index*
  [*index-or-slice* **]**

index-or-slice:
  index
  slice

index:
  expression

slice:
  first **:** *last* **:** *stride*
  first **:** *last*

first:
  expression

last:
  expression

stride:
  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:

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]S;
int:S x;
int:S y[20];            /* An array of parallel ints */
int:S *p2Pi = &y[5];    /* Pointer to the parallel int which is
                           the 5-th element of y */
int:S *:S Pp2Pi = &x;   /* Parallel pointer to parallel int where
                           every pointer points to the parallel
                           int x */

--x;                    /* Decrements every element of x */

--y[10];                /* Decrements every element of y[10] */

--p2Pi;                 /* Decrements p2Pi to point to the
                           preceding parallel int, y[4] */

--(*p2Pi);              /* Decrements every element of the
                           parallel int y[4] */

--(*Pp2Pi)              /* Decrements the elements of the parallel
                           ints denoted by *Pp2Pi */
```

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

Constraints

Revise for pointer to parallel:

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.

Add:

The **&** operator shall not be applied to a parallel lvalue that results from parallel indexing or from array subscripting with a parallel subscript.

> Note: you cannot take the address of an element of a parallel operand, nor of a parallel lvalue produced by parallel indexing or by parallel subscripting.

Semantics

Add for pointer to parallel:

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

> A parallel pointer is not produced by application of **&** to a parallel object, although a parallel pointer value may be produced by replication of a pointer to parallel or during parallel subscripting of an array (see §3.3.2.1).
>
> Note that the application of **&** to a shape object produces a pointer to that shape.

Examples

```
shape [10]S;              /* Shape S */
shape *sp;                /* Pointer to shape */
float a, b;               /* Nonparallel floats */
int i;                    /* Nonparallel int */
int:S x;                  /* Parallel int */
double:S y, z;            /* Parallel double */
int:S *p2Pi;              /* Pointer to parallel int */
float *:S Pp2f;           /* Parallel pointer to float */
double:S *:S Pp2Pd;       /* Parallel pointer to parallel double */
int:S A[20];              /* Array of parallel int */
double:S B[30];           /* Array of parallel double */

sp = &S;                  /* Assigns sp to point to shape S */

p2Pi = &x;                /* Assigns p2Pi to point to parallel int x
                             */

p2Pi = A;                 /* Since A is an array, it decays to a
                             pointer to parallel int.  p2Pi is
                             assigned that pointer, which is the
                             address of the 0-th 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 */

Pp2Pd = &y;               /* Assigns each element of Pp2Pd to point
                             to parallel double y */

[2]Pp2Pd = &z;            /* Assigns 2nd element of Pp2Pd to point
                             to parallel double z */

Pp2Pd = B+x;              /* Assigns Pp2Pd to be the parallel
                             pointer to double returned as the sum
                             of B and the parallel int x */

&[3]x;                    /* Constraint violation */
&[x]y;                    /* Constraint violation */
&B[x];                    /* Constraint violation */
```

Add for pointer to parallel and parallel pointers:

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*.

The result of the * operator applied to a parallel pointer of shape *S* to nonparallel type *T* is a parallel value of type *T* and shape *S*. The value at each position of the resulting parallel *T* value is the result of dereferencing the pointer at the corresponding position of the parallel pointer.

The result of the * operator applied to a pointer P of type "parallel pointer of shape *S* to parallel type *T* also of shape *S*" is a parallel value of parallel type *T* and shape *S*. The value at each position of the resulting parallel *T* value is the result of position-oriented dereferencing of the parallel pointer, yielding the value at the corresponding position of the parallel *T* object referenced by the pointer at that position of P. That is, for each position *i* in shape *S*, the value at position *i* is:

```
[i](*[i]P)  /* Where P is the parallel pointer */
```

Add for pointer to shape:

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

Examples

```
float a,b;
shape [10]S;
shape *sptr = &S;        /* Initialized to point to S */
int:S x = 3;             /* Initializes all elements to 3 */
int:S *p2Pi = &x;        /* Initialized to point to
                            parallel int x */
double:S B[30];
float *:S Pp2f = &a;     /* Initializes all elements to point to
                            float a */
double:S *:S Pp2Pd;

[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 int x;
                            all elements get &B[3], except the
                            5th element of Pp2Pd will get &B[2] */

*sptr;                   /* Denotes the shape pointed to by sptr;
                            in this case S */

*p2Pi;                   /* Denotes the parallel int x */

*Pp2f;                   /* Denotes a parallel float of shape S; in
                            this case equal to a in all positions
                            except equal to b in position 7 */

*Pp2Pd;                  /* Denotes a parallel double of shape S;
                            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--i.e., the value
                            in position i will be [i]B[3], except
                            in position 5, the value will be
                            [5]B[2]. */
```

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

Constraints

Revise:

The operand of the unary **+** or **−** operator shall have parallel or nonparallel arithmetic type; of the **~** operator, parallel or nonparallel integral type; of the **!** operator, parallel or nonparallel scalar type.

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

```
shape [20][20]S;
float:S 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]

Constraints

Revise:

The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to a parallel type with incomplete element type, to the parenthesized name of such a type, or to an lvalue that designates a bit-field object.

Semantics

Add:

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

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

Examples

```
shape [10]S, [20]T;
int:S a;

/* The following expressions evaluate to 1 */
sizeof(int:S) == sizeof(int:T);
sizeof(a) == sizeof(int:S);

/* The following does not necessarily evaluate to 1 */
sizeof(int:S) == sizeof(int);
```

The result of the **`sizeof`** operator applied to an array object whose element type is parallel is the product of the array length and the **`sizeof`** operator applied to the element type.

Examples

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

/* The following expression evaluates to 1 */
sizeof(x) == (10 * sizeof(int:S));
```

The result of the **`sizeof`** operator applied to a shape variable or to a shape type is the number of bytes in a shape object.

Examples

```
shape *Sptr;
shape [10]S;

Sptr = (shape *) malloc(sizeof(shape));
                    /* Allocates new shape object */

sizeof(S);          /* Denotes the size of the shape object S */
```

3.3.3.5 The **`shapeof`** operator [NEW]

Constraints

The **`shapeof`** operator shall be applied only to an identifier of a previously declared variable of parallel type.

Semantics

The **`shapeof`** operator yields a pointer to the shape object associated with its operand.

The **`shapeof`** operator may be used wherever a pointer to a shape may be used.

Examples

```
shape [10]S;
int:S x;
shape *sp = shapeof(x);
int:(*sp) y;                  /* x and y are associated with the
                                 same shape object */
int:S *z;

/* The following evaluates to 1 */
dimof(*shapeof(x),0) == dimof(S,0) == 1;

/* Allocates an array of 20 parallel ints */
z = (int:S *)palloc(sp,20*sizeof(int:S));
```

3.3.3.6 Parallel indexing [NEW]

Constraints

The *postfix-expression* shall be of parallel type.

The *parallel-index* shall consist of *n* *index-or-slice* expressions where *n* is the rank of the shape of the nearest enclosing parallel expression being indexed. Each *index* expression shall be of integral or parallel integral type. Each subexpression of a *slice* expression shall be of integral type.

All *index* expressions of parallel integral type in the same *parallel-index* shall be of compatible shape type.

If any *index-or-slice* expression in a *parallel-index* is a *slice* expression, all *index* expressions in the same *parallel-index* shall be of integral type, and the *postfix-expression* shall denote a parallel lvalue.

If the *postfix-expression* is being sliced, it shall not be a sliced expression .

An expression may be sliced only once; nested slicing is not allowed.

If the *postfix-expression* is being parallel indexed and is itself a parallel-indexed expression, it must be enclosed in parentheses.

> This allows the compiler to determine the rank of each of the *parallel-index* expressions. Note that in order to parallel index an expression more than once, the intermediate results must be parallel, which occurs when one or more index expressions is of parallel type. See the examples below.

Semantics

A *postfix-expression*, **`E2`**, preceded by a *parallel-index*, **`E1`**, which consists of a sequence of one or more *index* expressions enclosed in square brackets **`[]`**, is a *parallel-indexed expression*. The shape of **`E2`** is herein denoted by **`S2`**, and the shape of each of the *index* expressions of parallel integral type in **`E1`**, if any exist, is herein denoted by **`S1`**.

> Note that such an expression is called a parallel-indexed expression because the operand being indexed is parallel, regardless of whether any *index* expression is of parallel integral type.

A *postfix-expression*, **`E2`**, preceded by a *parallel-index*, **`E1`**, which contains one or more *slice* expressions, is a *sliced expression*. The shape of **`E2`** is herein denoted by **`S2`**.

<u>*Parallel-indexed expressions with a nonparallel index:*</u>

If all the *index* expressions of **`E1`** are of integral type, a 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 designates a selected element of **`E2`**; if **`E2`** is an lvalue, the result is an lvalue.

Note that parallel indexing by a nonparallel index always selects the designated position of the expression being indexed, independent of the context of the shape of that expression.

If the *index* expression designates a nonexistent position of the expression being indexed, the behavior is undefined.

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 position (i,j) of b */

[100]a;           /* Undefined behavior since index is 100 */

[5]b;             /* Constraint violation - improper indexing for
                     rank 2 object */
```

Parallel-indexed expressions with a parallel index:

In a *parallel-index*, *index* expressions of parallel integral type need not be the same shape as the parallel expression being indexed.

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

If more than one of the *index* expressions in **`E1`** is of parallel type, all the parallel *index* expressions must be of the same shape; otherwise, the behavior is undefined.

If one or more of the *index* expressions in **`E1`** is of parallel type, each nonparallel integral *index* expression is first promoted to a parallel **`int`** of the same shape as the parallel *index* expressions by replicating the integral expression value. The resulting *parallel-index* **`E1`** designates a mapping from the shape **`S2`** of the *postfix-expression* **`E2`** to the shape **`S1`** of **`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 value designating elements of **`E2`** whose selection and order is as directed by the *index* expressions **`E1`**; if **`E2`** is an lvalue, the result is an lvalue.

When a *parallel-index* is of parallel type, the mapping behaves according to the following:

```
/* When used as a modifiable lvalue, updated by E3 */
for (i1=0; i1<dimof(S1,0); i1++)
  ...
    for (im=0; im<dimof(S1,m-1); im++)
      [[i1]...[im]index1]...[[i1]...[im]indexn] E2 =
        [i1]...[im]E3;

/* When used as an rvalue, as if temporary T3 created */
Type:S1 T3;
for (i1=0; i1<dimof(S1,0); i1++)
  ...
    for (im=0; im<dimof(S1,m-1); im++)
      [i1]...[im]T3 =
        [[i1]...[im]index1]...[[i1]...[im]indexn] E2;
```

where **`S1`** is of rank m, **`S2`** is of rank n, index$_j$ indicates the j-th parallel index value of **`E1`**, and Type is the element type of **`E2`**. That is:

- when used where a modifiable lvalue is expected (i.e., on the left side of an assignment), the mapping selects elements of the object designated by **`E2`** to be updated. It may select the same element to be updated more than once, and some elements may not be selected to be updated at all. The result for an element selected more than once is unspecified; it will be chosen in an implementation-defined manner from one of the values of the elements selected from **`E3`** to be mapped to that position of **`E2`**.

- when used as an rvalue, the mapping produces a temporary parallel value of shape **`S1`** whose elements are selected from **`E2`** according to the *index* expressions, where the same element of **`E2`** may be selected multiple times, and not all elements of **`E2`** need be selected.

Examples

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

index1 = 3 - pcoord(S,0);
                         /* Assigns index1 to be {3,2,1,0} */

index2 = 2;              /* Assigns 2 to all elements of
                            index2; {2,2,2,2} */

[index2]index1;          /* Denotes a parallel int of shape
                            S whose values at each position
                            are the value of [2]index1;
                            {1,1,1,1} */

[3]([index2]index1);     /* Denotes element 3 of the parallel
                            int denoted by [index2]index1 */

[2][index2]index1;       /* Constraint violation: improperly
                            parenthesized */

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; b gets
                            {[3][2]a,[2][2]a,[1][2]a,[0][2]a} */
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 a below */
for (i=0; i<4; i++)
  [[i]index1][[i]index2]a = [i]b;
```

The context of **S1** also affects which positions are selected in a parallel-indexed expression. There is an implicit contextualization of the resulting parallel **int** denoted by **E1**.

If a value in the parallel **int** denoted by **E1** selects a nonexistent position of **E2**, the behavior is undefined.

Examples

```
shape [5]S;
int:S a, b, c;

a = pcoord(S,0) + 1;     /* a gets {1, 2, 3, 4, 5} */

[2]a = 10;               /* a is now {1, 2, 10, 4, 5} */

b = 1;                   /* b gets {1, 1, 1, 1, 1} */

where (a<5) {            /* Updates elements 0,1, and 3 of b */
   [a]b = 0;             /* to be 0; elements 2 and 4 are not */
}                        /* updated; b is now {0, 0, 1, 0, 1} */

a = pcoord(S,0) + 5;     /* a gets {5, 6, 7, 8, 9} */

b = pcoord(S,0) %% 3;    /* b gets {0, 1, 2, 0, 1} */

c = pcoord(S,0);         /* c gets {0, 1, 2, 3, 4} */

[b]a = c;                /* Updates elements 0, 1, and 2 of a;
                            element 0 gets either 0 or 3;
                            element 1 gets either 1 or 4  */

a = pcoord(S,0) + 5;     /* a gets {5, 6, 7, 8, 9} */

c = [b]a;                /* Updates all elements of c;  elements 0
                            and 3 of c get [0]a; elements 1 and 4
                            of c get [1]a; element 2 of c gets
                            [2]a; c is now {5, 6, 7, 5, 6} */

[a]b;                    /* Undefined; tries to select nonexistent
                            positions of b */
```

Sliced expressions:

Each component *slice* expression or *index* expression in the *parallel-index* of a sliced expression denotes a sequence of positions along the axis of shape **S2** that corresponds to the placement of the component within the sequence of components comprising the *parallel-index* .

The sequence denoted by a *slice* expression is as follows:

- The *first* expression denotes the first position in the sequence.
- The *last* expression denotes a limit (ceiling or floor) for the last position in the sequence.
- The *stride* expression denotes a stride or increment between successive positions in the sequence.

If the *stride* expression is omitted, a stride of 1 is assumed.

If the stride is 0, the sequence denoted is empty.

If the stride is positive, the sequence consists of the regularly spaced sequence of positions beginning with that denoted by the *first* expression and proceeding in increments of the stride up to the largest position not greater than that denoted by the *last* expression.

If the stride is negative, the sequence begins with the position denoted by the *first* expression and proceeds in increments of the stride down to the smallest position equal to or greater than that denoted by the *last* expression.

The sequence denoted by an *index* expression consists of the single position denoted by the expression.

Examples

```
shape [100][100]S;
shape [10][10][10]T;
int:S x;
float:T y;

[1][0:99:10]x;     /* Denotes the sequence {1} in the first axis,
                      and {0,10,20,30,40,50,60,70,80,90} in the
                      second axis; the shape of the result is
                      "shape [1][10]". */

[9:5:-1][3:7:2]x; /* Denotes the sequence {9,8,7,6,5} in the first
                      axis, and {3,5,7} in the second axis; the
                      shape of the result is "shape [5][3]". */

[1:10:2][5][10:5:-1]y;
                   /* Denotes the sequence {1,3,5,7,9} in the first
                      axis, {5} in the second axis, and
                      {10,9,8,7,6,5} in the third axis.  Since 10
                      is a nonexistent position in the third axis
                      of T, the behavior is undefined. */
```

If the sequence denoted by any component of a *parallel-index* is empty or contains nonexistent positions for the corresponding axis in shape **S2**, the behavior is undefined.

Otherwise, these sequences designate a mapping from the shape **S2** of **E2** to a conceptual shape **S3**, which has rank and dimensions as described below, an unspecified layout, and context in which all positions are active.

> No shape object is created for the shape **S3**; the concept of the shape **S3** is used only for expository purposes in describing the semantics of sliced expressions. A sliced expression denotes a subobject of a parallel object and may only be used in expressions involving nonparallel operands and other sliced expressions of shape **S3** (see §3.3).

The shape **S3** has rank equal to that of **S2**, but the dimension along each axis is equal to the number of positions in the sequence denoted by the corresponding *slice* expression for that axis in the *parallel-index*.

The type of the result is a parallel type of shape **S3** whose element type is the element type of **E2**. The result is a parallel value designating elements of **E2** whose selection and order is as directed by the sequences of positions denoted by the components of **E1**; if **E2** is an lvalue, the result is an lvalue.

The mapping designated by a sliced expression behaves according to the following:

```
/* When used as a modifiable lvalue, updated by E3 */
for (i1=0; i1<dimof(S3,0); i1++)
  ...
    for (ir=0; ir<dimof(S3,r-1); ir++)
      [seq1i1]...[seqrir]E2 = [i1]...[ir]E3;

/* When used as an rvalue, as if temporary T3 created */
Type:S3 T3;
for (i1=0; i1<dimof(S3,0); i1++)
  ...
    for (ir=0; ir<dimof(S3,r-1); ir++)
      [i1]...[ir]T3 = [seq1i1]...[seqrir]E2;
```

where *r* is the **rankof(S3)**, Type is the element type of **E2**, and $seqk_j$ is the j-th element of the k-th sequence of positions denoted by the *parallel-index*, corresponding to axis (k-1).

Note that element selection from **E2** is independent of the context of **S2**; the elements at the positions designated by the mapping are always selected.

If the result of the sliced expression **E1 E2** is used in an expression in which another parallel operand is the result of a sliced expression, but does not have shape equivalent to **S3** as described, the behavior is undefined.

Examples

```
shape [10][10]S;
shape [1024]T;
shape [512]U;
int:S x, y;
float:T z;
float:U w;
float result;

[0:9:2][0:9:2]x = 5;     /* Assigns elements at even positions of x
                            to be 5 */

[1:9:2][1:9:2]x = 10;    /* Assigns elements at odd positions of x
                            to be 10 */

[0:4:1][0:4:1]x = [5:9:1][5:9:1]x;
                         /* Assigns upper corner of x from the
                            lower corner of x */

[9:0:-1][0:9:1]x = [0:9:1][0:9:1]y;
                         /* Assigns x to be the transpose of y */

[9:0:-1][0:9:1]x = [0:9:1][0:9:1]x;
                         /* Assigns x to be its own transpose; note
                            that all reads from x will be done
                            before writes to x */

[0][0:9:1]x = [1][0:9:1]x + [2][0:9:1]x;
                         /* Assigns first row of x to be the sum of
                            the second and third rows of x */

[9:0:-1][0:9:1]x = x;    /* Undefined behavior; all operands must
                            be nonparallel or sliced expressions of
                            equivalent shape */

[0:1023:1]z += 1;        /* Add one to every element of z */

result = += [0:511]z;    /* Finds sum of first 512 elements of z */

result = (+= [0:511:1]z) - (+= [512:1023:1]z);
                         /* Finds difference of sums of first and
                            second halves of z */

w = [0:511:1]z + w;      /* Undefined behavior; all operands must
                            be nonparallel or sliced expressions of
                            equivalent shape */
```

3.3.3.7 Unary reduction operators [NEW]

Constraints

The operands of unary **+=**, **-=**, ***=**, and **/=** shall be of parallel arithmetic type.

The operands of unary **&=**, **^=**, and **|=** shall be of parallel integral type.

The operands of unary **<?=** and **>?=** shall be of parallel arithmetic or parallel pointer 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. If no positions of the operand are active, the result is 0, cast to the element type of the operand.

The result of the **-=** operator is the negation of the sum of all the elements of the operand. If no positions of the operand are active, the result is 0, cast to the element type of the operand.

The result of the ***=** operator is the product of all the elements of the operand. If no positions of the operand are active, the result is 1, cast to the element type of the operand.

The result of the **/=** operator is the reciprocal of the product of all the elements of the operand. If no positions of the operand are active, the result is 1, cast to the element type of the operand.

The result of the **&=**, **^=**, and **|=** operators are the bitwise AND, XOR, and OR, respectively, of all the elements of the operand. If no positions of the operand are active, the result of **&=** is ~0, and the result of **^=** and **|=** is 0.

The result of the **<?=** and **>?=** operators is the minimum and maximum, respectively, of all the elements of the operand. If no positions of the operand are active, the result is the minimum or maximum value for the element type, respectively, as defined in **<limits.h>** for integral types and **<float.h>** for floating types.

Examples

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

result = +=x;            /* Assigns result to be the sum of all the
                            elements of x */

result = <?=x;           /* Assigns result to be the minimum valued
                            element of x */
```

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

Constraints

Revise:

Unless the *type-name* specifies void type, the *type-name* shall specify qualified or unqualified, parallel or nonparallel, scalar type and the operand shall have parallel or nonparallel scalar type.

A pointer to parallel type shall not be cast to a pointer to nonparallel type. A pointer to nonparallel type shall not be cast to a pointer to parallel type.

A parallel pointer shall not be cast to a nonparallel pointer.

Semantics

Add:

If the *type-name* specifies a parallel type and the operand is also of parallel type, then if the shape of the parallel type and the shape of the operand are the same, 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*; if the shape of the type and the shape of the operand are not the same, the result is undefined.

Parallel to parallel casts should not imply data movement or communication.

Examples

```
shape [20]S;
shape [10][2]T;
int:S a;

(float:S) a;            /* Denotes the parallel float whose
                           elements are the converted int values
                           of corresponding positions of a */

(int:physical) a;       /* Undefined behavior */
(int:T) a;              /* Undefined behavior */
```

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.

Examples

```
shape [200]S;

(int:S) 1.0;            /* Denotes a parallel int of shape S
                           whose elements all have value 1 */
```

If the *type-name* specifies a nonparallel type and the operand is of parallel type, the result is to 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.

Examples

```
shape [200]S;
int:S a;

(double) a;               /* Selects one of the 200 elements of a,
                             and converts it to double */
```

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

Syntax

Revise:

multiplicative-expression:
  cast-expression
  multiplicative-expression `*` *cast-expression*
  multiplicative-expression `/` *cast-expression*
  multiplicative-expression `%` *cast-expression*
  multiplicative-expression `%%` *cast-expression*

Constraints

Revise:

Each of the operands shall have arithmetic or parallel arithmetic type. The operands of the `%` and `%%` operators shall have integral or parallel integral type.

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]S;
float:S f;
int:S 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 S 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 S
                       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 pointer to parallel types and parallel pointer types:

For addition, one of the following shall hold:

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

For subtraction, one of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers to qualified or unqualified versions of compatible object types;
- the left operand is a pointer to an object type, and the right operand has integral type or parallel integral type; or
- the left operand is a parallel pointer to an object type, and the right operand has integral or parallel integral type.

Semantics

Add for pointer to parallel types and parallel pointer types:

Adding an integral value to, or subtracting an integral value from, a pointer to parallel type results in a pointer to a subsequent or preceding element of an array whose element type is the parallel type. Subtracting two pointers to parallel, when both point to elements of the same array object, results in the difference of the subscripts of the two array elements. Subtracting two pointers to parallel which do not point to elements of (or one past the last element of) the same array object will result in undefined behavior.

Pointer arithmetic with pointers to parallel objects behaves just as does pointer arithmetic with pointers to nonparallel objects.

Adding a parallel integral value to, or subtracting a parallel integral value from, a parallel pointer results in a parallel pointer of the same type and shape whose value at each position is the sum or difference of the pointer value at the corresponding position of the parallel pointer and the integral value at the corresponding position of the parallel integral value. As usual, both operands must be of the same shape or the behavior is undefined.

If a parallel integral value is added to or subtracted from a nonparallel pointer value, or if a nonparallel integral value is added to or subtracted from a parallel pointer value, the nonparallel operand is first promoted to parallel, and then the addition or subtraction proceeds as described above.

Subtracting two parallel pointers results in a parallel `int` whose value at each position is the difference of the subscripts of two array elements, provided the pointer values at the corresponding positions of the operands point to elements of the same array object; otherwise the value at this position is undefined.

Examples

```
shape [10]S, [20]T;
float:S f;
double:S g;
int:S h[100];
int:S *p2Pi;
int:S index;
int:T index2;
int:S *:S Pp2Pi;
int:S *:S Pp2Pi_2;

f - g;                  /* Denotes a parallel double whose
                           value at each position is the
                           difference of the values at the
                           corresponding positions of f and g. */

p2Pi = &h[5];           /* Assigns p2Pi to point to 5th element of
                           array of parallel ints h */

Pp2Pi = &h[50];         /* Assigns every element of Pp2Pi to point
                           to the 50th element of h */

Pp2Pi_2 = &h[100];      /* All point to h[100] */

[5]Pp2Pi_2 = &index;    /* The 5th element points to index */

p2Pi + 5;               /* Denotes a pointer to the 10th element
                           of h */

Pp2Pi - 5;              /* Denotes a parallel pointer where every
                           element points to the 45th element of h
                           */

p2Pi - index;           /* Denotes a parallel pointer whose value
                           at each position is the difference of
                           &h[5] and the int at the corresponding
                           position of index */

Pp2Pi + index;          /* Denotes a parallel pointer whose value
                           at each position is the sum of the
                           pointer value and the int value in the
                           corresponding positions of Pp2Pi and
                           index, respectively */

&h[0] - p2Pi;           /* Denotes the difference of subscripts;
                           in this case, the integral value -5 */

Pp2Pi_2 - Pp2Pi;        /* Denotes a parallel int whose value at
                           each position is the difference of the
                           subscripts; in this case the difference
                           is 50 in every position, except in
                           position 5, where the value is
                           undefined */

Pp2Pi_2 + index2;       /* Undefined, shapes are not the same */

p2Pi - &index;          /* Undefined, not same array object */
```

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

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

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

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

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

Syntax

<u>*Revise as indicated:*</u>

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

Constraints

<u>*Revise:*</u>

One of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers or parallel pointers to qualified or unqualified versions of compatible object types; or
- both operands are pointers or parallel pointers to qualified or unqualified versions of compatible incomplete types.

Semantics

If one operand is of parallel type and the other operand is of nonparallel type, the operand of nonparallel type is first promoted to a parallel type of the same shape as the parallel operand, as described in §3.2.3.

The **`<?`** operator results in the minimum value of its operands.

The **`>?`** operator results in the maximum value of its operands.

Operands which are pointers to parallel observe the same semantics as pointers to nonparallel.

Operands which are parallel pointers observe the same semantics elementally as ordinary pointers, producing a parallel result whose value at each position is the result of applying the operator to the values at the corresponding positions of the operands.

Examples

```
shape [100]S;
int:S x, y;
int:S z[100];
int:S *ptr1, *ptr2;
int:S *:S pptr1, *:S pptr2;

x > 0;               /* Denotes a parallel int value of shape S
                        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 S
                        whose value at each position is the minimum
                        of the values at the corresponding positions
                        of x and y. */

ptr1 > ptr2;         /* Denotes an int value, which is 1 if ptr1
                        points to an element of the same array as
                        ptr2 but with a higher subscript, and 0 or
                        undefined otherwise. */

pptr1 > pptr2;       /* Denotes a parallel int value of shape S
                        whose value at each position is the result of
                        comparing the pointer values at the
                        corresponding positions of the operands. */
```

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

Constraints

<u>Revise:</u>

One of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers or parallel pointers to qualified or unqualified, parallel or nonparallel, versions of compatible types;
- one operand is a pointer or a parallel pointer to an object or an incomplete type, and the other is a pointer or a parallel pointer to a qualified or unqualified version of **`void`**;
- one operand is a pointer or a parallel pointer to nonparallel type and the other is a null pointer constant ; or
- one operand is a pointer to parallel type and the other is a null pointer-to-parallel constant (see §3.2.2.3).

Examples

```
shape [100][100]S, [100][100]T;
shape [100] U, V;
shape *ps1, *ps2;
float:S x, y;
float:S *ppf = &x;

x == y;            /* Denotes a parallel int value of shape S
                      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. */

/* Assign the following values */
V = T;
ps1 = &S;
ps2 = &T;
x = 1.0;
[0][0]x = 0.0;

/* The following all evaluate to 1 */
S == T;
S != U;
S == V;
where (x > 0) {
   S != T;
}
*shapeof(x) == *shapeof(y);
ps1 != ps2;
*ps1 == *ps2;
ppf == &x;
ppf != &y;
```

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

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

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

x & 0x01;         /* Denotes a parallel int value of shape S
                     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]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

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

x ^ y;            /* Denotes a parallel int value of shape S
                     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]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

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

x | y;              /* Denotes a parallel int of shape S  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]

Constraints

Revise:

Each of the operands shall have parallel or nonparallel scalar type.

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 narrowed context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. That is, the active positions in which the LHS evaluates to zero are made inactive for evaluating the RHS. The result is a parallel **`int`** which is 1 in those positions for which the values in the corresponding active positions of both the LHS and RHS are nonzero, and 0 in all other active positions.

Examples

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

x && y;             /* Denotes a parallel int of shape S 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 second operand is only
                       evaluated in those active positions for which
                       the first operand is nonzero. */
```

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

Constraints

Revise:

Each of the operands shall have parallel or nonparallel scalar type.

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 narrowed context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. That is, the active positions in which the LHS evaluates to one are made inactive for evaluating the RHS. The result is a parallel value which is 0 in those active positions for which the values in the corresponding positions of both the LHS and RHS are 0, and 1 in all other active positions.

Examples

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

x || y;          /* Denotes a parallel int of shape S 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 second operand is only
                    evaluated in those active positions for which
                    the first operand is zero. */
```

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

Constraints

Revise:

The first operand shall have nonparallel scalar type or parallel type with scalar element type.

If both of the second and third operands are of parallel type, they shall be of compatible parallel types.

One of the following shall hold for the second and third operands or, if the second or third operands are of parallel type, the following shall hold for the element types of the second and third operands:

...

Semantics

Add:

If the first operand has parallel type, the context of the shape of the first operand is constrained by the expression denoting the first operand for the purpose of evaluating the second and third operands. That is, the behavior is as if the second and third operands are evaluated in the scope of a **where** statement whose mask expression is the first operand. For example,

```
opnd1 ? opnd2 : opnd3
```

behaves as if evaluated with the same contextualization of the following:

```
where ( opnd1 )
    opnd2;
else
    opnd3;
```

Note that both opnd2 and opnd3 are evaluated in this case, whereas if opnd1 is nonparallel, only one of opnd2 and opnd3 is evaluated.

Examples

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

x < 0 ? -x : x          /* Denotes a parallel int whose value at
                           each position is the absolute value of
                           the corresponding value 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 whose value at
                           each position is the value at the
                           corresponding position of x, if that
                           value is less than y, and otherwise it
                           is the value of z. */
```

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 of parallel type *T1* and the right operand is of parallel type *T2* such that *T1* and *T2* have compatible shapes and such that an operand having the element type of *T1* can be the left operand of simple assignment where the right operand has the element type of *T2*;
- the left operand is the identifier of a shape variable which is fully unspecified or partially specified when declared, and the right operand has compatible shape type;

...

A shape variable shall not be assigned a new shape value after it has been used in declaring objects of parallel type.

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.

If the left operand is of parallel type and the right operand is of parallel type, the object denoted by the left operand is assigned elementally by the right operand, including any necessary type conversions.

If the left operand is a parallel type of shape **`void`**, which can only happen if the left operand is a parameter in a function prototype specification, the left operand assumes the shape of the right operand.

If the left operand is the identifier of a shape variable, and the right operand denotes a shape object of compatible type, 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, and the value of the shape object becomes fully unassigned, partially assigned, or fully assigned, according to the type of the right operand. If the right operand is not shape compatible with the left operand, the behavior is undefined.

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

Examples

```
shape [10]S, []T, [][]U, V;
int:S x;
float:S y;
typedef struct { int a, b; float f; } stype;
stype z = { 5, 8, 2.5 };
stype:S w;

x = 10;            /* 10 is first converted to a parallel int */

w = z;             /* Each element of w is assigned a copy of z */

z = w;             /* Constraint violation */

y = x;             /* Assigns elements of y from corresponding
                      elements of x, converting ints to floats */

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;             /* Constraint violation, incompatible shapes */
T = U;             /* Constraint violation, incompatible shapes */
S = T;             /* Constraint violation, 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]

Constraints

Revise:

For the operators **+=** and **-=** only, either the left operand shall be a pointer or a parallel pointer to an object type, and the right shall have integral or parallel integral type; or the left operand shall have qualified or unqualified, parallel or nonparallel arithmetic type, and the right shall have parallel or nonparallel arithmetic type.

For the other operators, each operand shall have parallel or nonparallel arithmetic type consistent with those allowed by the corresponding binary operator.

Recall that the **%=**, **%%=**, **<<=**, and **>>=** operators were previously constrained in §3.3.16.

Semantics

<u>*Add:*</u>

When the left operand is of nonparallel scalar type and the right operand is of parallel type, the operation is a *scalar reduction assignment*. The value of the left operand is included in the reduction value computation (see §3.3.3.7). That is, the behavior of :

```
LHS op= RHS
```

is equivalent to:

```
LHS op= op= RHS
```

When the left operand is of parallel type and the right operand is of compatible parallel type, a *parallel reduction assignment* is performed. If the left operand is parallel indexed and specifies collisions (two positions of the parallel lvalue designate the same object), then values with the same destination are combined with each other and with the value at the destination using the associated operator. If there are no collisions, then a compound assignment is performed elementally on the values at the corresponding positions of the operands.

Examples

```
shape [10][20]S;
int:S x, y;
int:S *p2Pi;
int:S *:S Pp2Pi;
int result;
shape [10]T;
int ai[5];
int:T aP[5];

result += x;            /* Equivalent to result += +=x */

result -= x;            /* Equivalent to result += -=x */

result <?= x;           /* Equivalent to result <?= <?= x */

p2Pi += 1;              /* Increments pointer by 1 */

Pp2Pi += 1;             /* Increments all elements of parallel
                           pointer by 1*/

Pp2Pi += x;             /* Increments all elements by ints at
                           corresponding positions of x */

y *= x;                 /* Equivalent to y = y * x */

[x]y += x;              /* Equivalent to the following loop, in
                           the absence of collisions */
{ int i, j;
  for (i=0; i<10; i++)
    for (j=0; j<20; j++)
      [[i][j]x]y += [i][j]x;
}
```

```
ai[pcoord(T,0 % 5] += x;
                           /* ai[0] <-- ai[0] + [0]x + [5]x;
                              ai[1] <-- ai[1] + [1]x + [6]x;
                              ...
                              (there are collisions) */

aP[pcoord(T,0 % 5] += x;
                           /* [0]aP[0] <-- [0]aP[0] + [0]x;
                              [1]aP[1] <-- [1]aP[1] + [1]x;
                              ...
                              (there are no collisions) */

result %= x;               /* Constraint violation */
```

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

Examples

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

Semantics

Add for parallel constants:

Parallel constants are introduced implicitly by use of nonparallel constant expressions in binary or ternary operations involving a parallel operand; in this case the nonparallel constant is promoted to a parallel constant by replicating the constant value.

Parallel constants may also be introduced explicitly by the use of cast expressions where the operand of the cast is a nonparallel constant.

A parallel address constant is an address constant designating a parallel object.

Examples

```
shape [10][10]S;
float:S x;
static int:S y;
int:S *p = &y;             /* Parallel address constant */

x = 1.0;                   /* Constant 1.0 is replicated to form
                              parallel double of shape S */

x = (float:S)1.0;          /* Explicit cast to parallel float */
```

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 declaration-specifiers$_{opt}$
  type-specifier shape-specifier$_{opt}$ *declaration-specifiers*$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

shape-specifier:
  : *shape-expression*

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

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 or in pointer declarations.

The *declaration-specifiers* portion of a declaration shall contain at most one *shape-specifier* (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 *type*`:void *` type specifier can be used to specify a pointer to a parallel type of generic shape. A `void:void *` type specifier can be used to specify a pointer to a generic parallel type.

Semantics

Add:

If the specification of an array type includes any shape specifiers, the element type is so - specified, not the array type.

> This can only happen through the use of a `typedef`. The resulting type is an array of parallel.

Examples

```
shape [10]S;

typedef:S a;                  /* Syntax error */

typedef int:S Pi;             /* Type Pi 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;              /* Constraint violation, multiple
                                 shape specifiers */

int:void f1();                /* f1 returns a parallel int of
                                 unspecified shape */

int:void *p2Pigen;            /* p2Pigen is a pointer to parallel
                                 int of generic shape */

void:void *p2Pgen;            /* p2Pgen is a pointer to a generic
                                 parallel type */

typedef int atype[5];         /* atype is an array of int */

atype:S x;                    /* x is an array of 5 parallel ints
                                 of shape S */

int:S y[5];                   /* y is also an array of 5 parallel
                                 ints of shape S */
```

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

Syntax

Add:

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

shape-type-specifier:
    shape

Constraints

Add to the type specifier sets:

- shape-type-specifier

Add:

A shape type specifier shall only be used in a declaration of a shape object (see §3.5.4.4).

3.5.4 Declarators [ISO §6.5.4, ANSI §3.5.4]

Syntax

<u>*Revise as indicated:*</u>

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

direct-declarator:
  ...
  direct-declarator **(** *parameter-type-list* **)** *function-qualifier*$_{opt}$

function-qualifier:
  `elemental`
  `nodal`

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

pointer-qualifier:
  `elemental`

...

Constraints

A *shape-dimension-and-layout* shall only be used in a shape declarator (see §3.5.4.4).

3.5.4.1 Pointer declarators [ISO §6.5.4.1, ANSI §3.5.4.1]

Constraints

Add for parallel pointers, pointers to parallel, and elemental functions:

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`**. If **`T`** or **`D`** contains pointer declarators containing more than one shape specifier, then all the shape specifiers shall have compatible shape types.

Semantics

A declaration of the form "**`T D1`**" such that **`T`** is a parallel type and **`D1`** has the form

> **`*`** *type-qualifier-list*$_{opt}$ *pointer-qualifier*$_{opt}$ **`D`**

declares a pointer to parallel.

A declaration of the form "**`T D1`**" such that **`T`** is a nonparallel type and **`D1`** has the form

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

declares a parallel pointer to nonparallel (see §3.5.4.5).

A declaration of the form "**`T D1`**" such that **`T`** is a parallel type and **`D1`** has the form

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

declares a parallel pointer to parallel (see §3.5.4.5).

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

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

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

Examples

```
shape [10]S, [20]T;
int:S * p2Pi;           /* p2Pi is a pointer to parallel int */

int *:S Pp2i;           /* Pp2i is a parallel pointer to int */

int:S *:S Pp2Pi;        /* Pp2Pi is a parallel pointer to parallel
                           int */

int *:S *:S Pp2Pp2i;    /* Pp2Pp2i is a parallel pointer to a
                           parallel pointer to int */
```

```
int *:S *:T ptr1;        /* Constraint violation, incompatible
                            shapes */
```

Add for elemental pointers:

If a pointer declaration specifies the type "**`elemental`**-qualified pointer to *T*" and the pointer type is parallel, either directly or indirectly, then *T* also is considered to be a parallel type of the same shape as the pointer type. Cases where the pointer type is considered to be indirectly parallel are illustrated below.

If *T* has a shape specifier, then the **`elemental`** pointer qualifier adds no additional meaning to the declaration.

The **`elemental`** pointer qualifier only has meaning when determining the type of the enclosing declaration. In particular, it adds no meaning with regard to operations upon values.

The **`elemental`** pointer qualifier is removed from the type when a variable, parameter, etc., is declared. It can only remain in a type for a **`typedef`** name or a structure member. However, when an object is declared using the **`typedef`** name or the structure type, the **`elemental`** pointer qualifier is removed from the type.

Examples

```
shape [10]S;
typedef int * elemental PTR;
struct s1 {
  PTR m1;
  int *m2;
};

int *:S elemental p1;   /* Pointer is directly parallel so p1 is
                           parallel pointer to parallel int */

int * elemental *:S p2; /* Inner pointer is directly parallel so
                           outer pointer indirectly becomes
                           parallel and therefore p2 is parallel
                           pointer to parallel pointer to parallel
                           int */

PTR:S p3;               /* p3 is parallel pointer to parallel
                           int */

PTR p4;                 /* p4 is pointer to int */

struct s1:S Ps;         /* m1 is indirectly parallel because
                           enclosing struct was declared parallel.
                           Therefore Ps.m1 is parallel pointer to
                           parallel int.  Note that Ps.m2 is
                           parallel pointer to nonparallel int. */

struct s1 s;            /* s.m1 is pointer to int, s.m2 is pointer
                           to int */

int:S *:S elemental p5; /* p5 is parallel pointer to parallel; no
                           additional meaning is added by the
                           elemental qualifier */
```

3.5.4.2 Array declarators [ISO §6.5.4.2, ANSI §3.5.4.2]

Semantics

Add:

An array whose element type is parallel follows the usual semantics of arrays, requiring (conceptual) contiguity of array elements that allow subscripting and usual pointer arithmetic to be used interchangeably. However, since the element type is parallel, the elements of an element of the array are allowed to be discontiguous. This is consistent with the semantics of parallel objects. (See also §3.3.2.1, §3.3.3.2, and §3.3.6.)

Examples

```
shape [100][200]S;
int:S array[50];

/* The following evaluates to 1 */
&array[0] == &array[1] + sizeof(array[0]);
```

3.5.4.3 Function declarators (including prototypes) [ISO §6.5.4.3, ANSI §3.5.4.3]

Constraints

Add for qualified functions:

A function declarator qualified with the **`elemental`** or **`nodal`** qualifiers shall contain a parameter type list.

> A qualified function must provide a prototype specification of its parameters.

Add for elemental functions:

An **`elemental`**-qualified function shall not declare parameters of parallel or shape types, nor shall it return a parallel type.

Add for nodal functions:

A **`nodal`**-qualified function shall declare parameters only of the following types:

- parallel type of shape **`void`**; or
- pointer to nonparallel type; or
- nonparallel type.

Semantics

Add for elemental functions:

A function 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

```
int f1(int) elemental;   /* Elemental function returning int */

int *f2(int) elemental;  /* Elemental function returning pointer
                            to int */
```

<u>*Add for nodal functions:*</u>

A function 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 [1024]S;

int f1(int:S) nodal;    /* Nodal function returning int:physical
                           if called from nonnodal, and an int if
                           called from nodal */

int:S f2(int:S) nodal;  /* Nodal function returning an int:S */
```

3.5.4.4 Shape declarators [NEW]

Syntax

shape-dimension-and-layout:
  shape-dimension-and-layout-specifier shape-dimension-and-layout *opt*

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

block-expression:
  expression

scale-expression:
  expression

Constraints

The *expression* used to specify the dimensions 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 *shape-dimension-and-layout-specifier* in a *shape-dimension-and-layout* specifies a dimension, all the specifiers shall specify a dimension.

If at file scope or in an extern or static declaration, **block** and **scale** expressions shall be constant integral expressions. If any *shape-dimension-and-layout-specifier* includes a **block** layout specification, then no other *shape-dimension-and-layout-specifier* in the same *shape-dimension-and-layout* shall include a **scale** layout specification, and vice versa.

Mixing of **block** and **scale** specifiers in a shape declaration is not allowed.

Semantics

A shape declarator denotes an object of type shape. 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 type may be fully unspecified, partially specified, or fully specified, depending on the rank and dimensions specified.

The **`block`** and **`scale`** expressions used to specify the layout of a shape shall be nonnegative integral expressions, and **`scale`** expressions shall be greater than zero; otherwise, the behavior is undefined.

The object denoted by a shape declarator is *fully unassigned* if its shape type is fully unspecified; all component values of the shape object are undefined. The object denoted by a shape declarator is *partially assigned* if its shape type is partially specified; its rank component is assigned the rank of the shape type, and all other components are undefined. The object denoted by a shape declarator is *fully assigned* if its shape type is fully specified; its rank, dimensions, and layout components are assigned the rank, dimensions, and layout specified in the shape type, and its context is assigned a value such that all positions are initially active.

The rank of a shape is determined from the number of *shape-dimension-and-layout-specifier*'s given. The dimension in each axis is optionally specified as the *expression* in a *shape-dimension-and-layout-specifier*.

The **`block`** and **`scale`** specifiers suggest a layout or data distribution for parallel operands associated with this shape declarator. The **`block`** specifier suggests that data is to be distributed in blocks of the indicated size. The **`scale`** specifier suggests that data be partitioned into maximal blocks with block dimensions proportional to the scale expressions in the corresponding dimensions; each partition receives the indicated block size, if the shape is multidimensional; or a single block, if the shape is one-dimensional. Partitions of blocks are distributed in round-robin fashion to the nodes of the target architecture.

> Note: if the number of partitions is not a multiple of the number of nodes, not all nodes receive an equal number of partitions; if the data size is not a multiple of the determined block size, not all partitions are necessarily the same size.

A **`block`** specifier equal to the dimension in a given axis suggests that no distribution is desired across that axis of an object.

If no **`block`** or **`scale`** specifier is given, or if a **`block`** specifier is equal to zero, the implementation determines an appropriate distribution for parallel operands of this shape on the target architecture.

The implementation is free to use a different layout than what is suggested. However, for identical layout specifications, an implementation shall use identical layout. (See §4.14.1.2 and §4.14.1.4.)

Examples

```
shape [100 block (10)]S;
        /* Suggests 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;
        /* Suggests objects of shape T are not to be
           distributed; all elements are in the same
           partition */

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

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

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

shape [100 scale (1)][100 scale (2)]Y;
        /* Suggests objects of shape Y are to be distributed
           in blocks with kx2k elements; k will be
           approximately sqrt((100x100)/2n) where n is the
           number of partitions */

shape [n block (m)]Z;
        /* Constraint violation if at file scope */

shape [0 scale(0)][10 block (-10)]ZZ;
        /* Constraint violations:  dimension and scale
           expressions must be greater than 0, and can't mix
           block and scale specifiers in a declaration */
```

3.5.4.5 Parallel object declarators [NEW]

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 structure 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 shape type.

Each *shape-specifier* in the declaration shall be an lvalue of type shape.

> If an expression is used in the *shape-specifier*, the expression must denote an lvalue of type shape. A shape-valued expression must denote an object in addressable memory to be used in a parallel object declaration.

Semantics

A declaration that includes a *shape-specifier* is a parallel object declarator. Parallel object declarators denote parallel objects–i.e., variables of parallel type. If the *shape-specifier* is a part of a *pointer*, the parallel object is a *parallel pointer*, a parallel object whose element type is a pointer type (see §3.5.4.2).

If the shape object denoted by the *shape-specifier* is not fully assigned when the parallel object declarator is declared, the behavior is undefined.

Examples

```
shape [100][4][25]S;
shape []T;
typedef int:S parint_t;
shape f();

int:S x;                /* x is a parallel int */

int *:S Pp2i;           /* Pp2i is a parallel pointer to int */

parint_t:S y;           /* Constraint violation; element type is
                           parallel type */

struct { parint_t e; }:S w;
                        /* Constraint violation; element type is
                           struct with parallel member */

int:(f()) z;            /* Constraint violation, f returns a shape
                           value, not a shape object */

int:T z;                /* Undefined behavior, T is not fully
                           assigned */
```

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

Syntax

The syntax for abstract declarators was not extended to allow a *shape-specifier* as with declarators, due to ambiguities that would otherwise be introduced in recognizing (optional) expressions in square brackets `[]`. These could be either a shape dimension specifier or an array dimension specifier. Therefore, abstract declarators for shape types or arrays of shape types must first define the shape type with a **`typedef`**.

Semantics

Add:

In a *direct-abstract-declarator* containing square brackets `[]` the abstract declarator being constructed designates an array type and not a shape type.

Examples

```
typedef shape [] shape_t1;
typedef shape [10] shape_t2;

void f1(shape);          /* Abstract declarator of fully
                            unspecified shape type */

void f2(shape []);       /* Abstract declarator of type array of
                            fully unspecified shape type */

void f3(shape_t1 [])     /* Abstract declarator of type array of
                            partially specified shape type */

void f4(shape_t2 [])     /* Abstract declarator of type array of
                            fully specified shape type */

void f5(int:void);       /* Abstract declarator of type parallel
                            int of void shape */

void f6(int:void []);    /* Abstract declarator of type array of
                            parallel int of void shape */

void f7(int *:void);     /* Abstract declarator of type parallel
                            having element type pointer to int and
                            void shape */

void f8(int:void *);     /* Abstract declarator of type pointer to
                            parallel int of void shape */
```

3.5.7 Initialization [ISO §6.5.7, ANSI §3.5.7]

Constraints

<u>*Revise as indicated:*</u>

...

All the expressions in an initializer for an object or parallel object that has static storage duration, or in an initializer list for an object or parallel object that has aggregate or union type shall be constant expressions .

...

Semantics

<u>*Add for parallel object declarators:*</u>

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
                            initialized to 1.5 */

union { int i; float f; }:S uu = { 1 };
                         /* 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

3.6.4 Selection statements [ISO §6.6.4, ANSI §3.6.4]

Add clarification:

Constraints

The controlling expression of an **`if`** or **`switch`** statement shall be of nonparallel type. The expression of each **`case`** label shall be of nonparallel type.

3.6.5 Iteration statements [ISO §6.6.5, ANSI §3.6.5]

Revise for clarification:

Constraints

The controlling expression of an iteration statement shall have nonparallel scalar type.

3.6.5.3 The `for` statement [ISO §6.6.5.3, ANSI §3.6.5.3]

Add for clarification:

Both *expression-1* and *expression-3* are allowed to have parallel type.

3.6.6 Jump statements [ISO §6.6.6, ANSI §3.6.6]

Semantics

Add for clarification:

If a jump statement is used to leave the nested context of a **`where`** or **`everywhere`** statement, the contextualization at the destination is restored for all shapes that were constrained by the contextualization statement(s) exited to the contextualization of the destination.

If a jump statement is used to enter the nested statement part of a contextualization statement or to enter a block declaring a parallel variable or shape, the behavior is implementation-defined.

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 nonelementally executes a **`return`** statement, the semantics are the same as a return for a nonelemental 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 nonelemental 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) (see §3.3.2.2). When a **`return`** statement with an expression is executed at a position, and the caller is a nonelemental 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 assignment. Control is not returned to the caller until all active positions have executed a **`return`** statement.

Examples

```
typedef struct coord { double s, y; } COORD;

double distance(COORD pt1, COORD pt2) elemental
{
  double x_delta, y_delta;

  x_delta = pt1.x - pt2.x;
  y_delta = pt1.y - pt2.y;
  return sqrt(x_delta*x_delta + y_delta*y_delta);
}

main()
{
  COORD a, b, sresult;
  COORD:S c, d, presult;

  sresult = distance(a,b);     /* Executes in nonparallel mode */

  presult = distance(c,d);     /* Executes in parallel mode,
                                  elementally */
}
```

<u>*Add for nodal functions:*</u>

When a nodal function that is declared to return a nonparallel value is returning to a nonnodal execution environment, then a parallel value of **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. 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 . Note that if returning to a nodal environment, S will be identical to T. 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 element type of the return expression is different than the element type of the declared return type.

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 multithreaded version of the single-threaded semantics for C.

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

Examples

```
int [10]S;
int [100]T;
int:S x,y,z;
int:T w;
int:physical one_per_node;

int:void nodal_add(int:void a, int:void b) nodal
{
  return(a + b);         /* Shape of return value as seen by caller
                            will be the same as the shape of the
                            two arguments, which in this case must
                            be the same */
}

int:physical physical_increment(int a) nodal
{
  return(a + 1);         /* Caller will pass an int:physical and
                            get an int:physical in return */
}

int:void nodal_bug(int:void a) nodal
{
  shape [64]S2;

  return((int:S2)19);    /* Undefined behavior:  shape of return
                            value must be derived from shape of a
                            parameter */
}

main()
{
  x = nodal_add(y,z);    /* OK, because x, y, z have same shape */

  w = nodal_add(y,z);    /* Undefined behavior, w not of same shape
                            as y, z */

  one_per_node = physical_increment(one_per_node);
                         /* OK */

  nodal_bug(w);          /* Undefined behavior */
}
```

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.

All parallel values have an associated shape and this shape contains its current context. Therefore, a contextualization statement first modifies the context of the shape of its mask expression, and then a function in the substatement is called. If this function accesses a value of the same shape as the mask expression, then the modified context will be used when evaluating expressions of this shape.

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 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;

everywhere (S) {
   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 variable declarations of parallel or shape types, or types derived from parallel or shape types.

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

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

An elemental function shall call only 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 shall call only nodal functions and elemental functions.

A nodal function shall not reference any identifier with file scope, except nodal or elemental functions, and any of the functions listed in §4.14.

Semantics

Add for elemental functions:

An elemental function that is executed nonelementally has the same semantics as a nonelemental 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 accesses a nonparallel object that was not defined within an elemental function, the behavior is undefined.

Since an elemental function executing elementally executes as if called once per active position, pointer dereferences have special position-oriented semantics. When the function is being executed as if for a particular active position, a dereference of a pointer that was created outside the elemental function as a “parallel pointer to parallel” accesses only the object in the corresponding position of the pointer target.

Examples

```
/* Linked list node for use either in parallel or
   nonparallel setting */

#include <dpce.h>

typedef struct node * elemental LINK;

struct node {
    int count;
    LINK next;    /* Pointer to either parallel or
                     nonparallel struct node */
};

/* Elemental function to build a linked list:
     n linked nodes are built
     count fields are initialized from values argument
*/
struct node *build(int n, int values[]) elemental
{
  /* Automatic "flow through" for elemental function locals:
     in elemental function everything is parallel or
     everything is nonparallel
  */
  struct node *last = NULL, *new;
  int i;

  for (i = n-1; i >= 0; i--) {
    new = (struct node*) malloc(sizeof(struct node));
    new->count = values[i];
    new->next = last;
    last = new;
  }
  return last;
}

/* Elemental function to sum "count" fields of a "struct
   node" list
*/
int sum(struct node *list) elemental
{
  int ret = 0;

  while (list != NULL) {
    ret += list->count;
    list = list->next;
  }
  return ret;
}

shape [100]S;

int n;
int values[10];
int:S many_values[10];
int:S many_n;
```

```
LINK head;         /* Nonparallel pointer to nonparallel
                      struct node */

LINK:S many_head; /* Parallel pointer to parallel struct
                      node */

main()
{
  /* Call to init values, many_values, many_n */
  init_stuff();

  head = build(n,values);                 /* nonparallel */
  many_head = build(many_n,many_values);  /* parallel */

  printf("sum of head is %d\n", sum(head));

  printf("sum of many_head is %d\n", +=(sum(many_head)));

  printf("sum of just [4]many_head is %d\n",[4](sum(many_head)));

  /* Or perhaps more efficiently */
  where (pcoord(S,0) == 4) {
    printf("sum of just [4]many_head is %d\n",+=(sum(many_head)));
  }
}
```

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 nonnodal execution environment is dereferenced in a nodal environment, the behavior is undefined.

Examples

```
struct x {
      int *p;            /* p is a pointer to nonparallel */
};

struct x a;
int n;

void nodal_f(struct x param) nodal
{
  struct x local;

  local = param;
  *local.p = 19;         /* Undefined behavior if value of a
                            pointer to nonparallel originated in
                            nonnodal environment; therefore can't
                            dereference it in nodal environment */
}

void another_nodal_f(void) nodal
{
  struct x local;
  int i;

  local.p = &i;
  nodal_f(local);        /* OK, the pointer value in local.p
                            originated in nodal environment */
}

main()
{
  a.p = &n;
  nodal_f(a);            /* Argument will be promoted to struct
                            x:physical but undefined behavior will
                            result when nodal_f dereferences the
                            member */

  another_nodal_f();     /* OK */
}
```

Inside nodal functions, pointer values that were created in a nonnodal execution environment as “pointer to parallel” have special position-oriented semantics. All pointer values accessed, directly or indirectly, for a parameter value derived for a particular position reference only objects in that position of the pointer target.

> Since nodal functions may not reference identifiers with file scope except for nodal and elemental functions, all pointer values that were created in a nonnodal environment may only be accessed via a data value passed as an argument to the nodal function that caused nodal execution to begin. All values of arguments to a nodal function are derived for a particular position of a particular shape. Therefore parameter values also have this association with positions of shapes.

Examples

```
/* Using a nodal function to implement the
   Sieve of Eratosthenes */

#define N 4800000

typedef unsigned char BOOL;

shape [N scale (1)] natural;

void StrikeOut ( int local_dim;
                 BOOL *local_primes[],
                 BOOL *local_active[],
                 int local_coords[],
                 int current_prime
               ) nodal
{
  int lowbound = local_coords[0];
  int offset = lowbound % current_prime;
  int i;

  /* Where is the first number that is a multiple of the
     current prime? */
  if (offset == 0) i = 0;
  else i = current_prime - offset;

  /* Stride through to knock out multiples of current prime */
  for ( ; i < local_dim; i += current_prime) {
    *local_primes[i] = 0;
    *local_active[i] = 0;
  }
}

main ()
{
  /* Each position of prime represents itself and is to be
     evaluated for primality.  The algorithm requires each
     node to have a single contiguous block of numbers. */

  int next_prime = 2;
  BOOL:natural active;
  BOOL:natural prime;

  where (pcoord(natural,0) < 2)
    prime = active = 0;
  else where (pcoord(natural,0) == 2) {
    prime = 1;
    active = 0;
  }
  else
    prime = active = 1;
```

```
    while (|= active)
        where (active) {

          /* Could strike out nonprimes by testing all active
             elements using %, but nodal function can apply more
             efficient method exploiting contiguous
             distribution.  Casts are needed on first two
             parameters to cause pointers to each active prime
             to be generated. */

          StrikeOut(nodepositionsof(natural),
                    (BOOL:natural *:natural) &prime,
                    (BOOL:natural *:natural) &active,
                    pcoord(natural,0),
                    next_prime);

          where (prime)
            next_prime = (<?= pcoord(natural,0));

          where (next_prime == pcoord(natural,0))
            active = 0;
        }

        printf("There are %d primes less than %d\n",
               (+= prime), N);
      }
    }
```

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 redeclares 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 translation units that use DPCE.

4.1.6 Use of Library Functions [ISO §7.1.7, ANSI §4.1.6]

There is one exception to the rule that library functions may be declared without the use of their associated header. If a translation unit declares DPCE functions without first including **<dpce.h>**, its behavior is undefined.

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

The header **<dpce.h>** defines the predefined shape identifier **physical** and the following types, and declares the functions specified below. Note: all the functions declared here are accessible to elemental and nodal functions as described in §3.7.1.

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;
```

The types **dpce_layout_specs_t** and **dpce_layout_t** are defined as follows:

```
typedef enum { DPCE_BLOCK, DPCE_SCALE } dpce_layout_specs_t;

typedef struct {
                 size_t size;
                 dpce_layout_specs_t spec;
               } dpce_layout_t;
```

4.14.1 General purpose DPCE utilities [NEW]

There are seven 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.

```
int abs(int j) elemental;
int atoi(const char *nptr) elemental;
long int atol(const char *nptr) elemental;
```

```
void qsort(void *base, size_t nmemb, size_t size,
         int (*compar)(const void*, const void*) elemental);
void *calloc(size_t nmemb, size_t size) elemental;
void *malloc(size_t size) elemental;
void *realloc(void *ptr, size_t size) elemental;
```

When executing elementally, the functions **`malloc`**, **`calloc`**, and **`realloc`** return a parallel pointer to parallel. However, each element of this pointer value has only position-oriented semantics and may only be used to reference an object in its same position. In particular, if a single element of the value is converted (for example, via parallel indexing), to a nonparallel pointer to parallel and the result is dereferenced, then the behavior is undefined. If it is necessary to create a pointer value with both position-oriented and "global" semantics, then the functions **`palloc`** (§4.14.1.6) and **`pfree`** (§4.14.1.8) should be used.

Examples

```
shape [10]S;
int:S *:S many_ptr_to_many;
int:S * one_ptr_to_many;
int:S many;
int:S many2;

one_ptr_to_many = &many;

*one_ptr_to_many = 19;         /* All positions of many receive 19
                                  */

many_ptr_to_many = &many;      /* All positions point to many */

[5]many_ptr_to_many = &many2;  /* Position 5 points to many2 */

*many_ptr_to_many = 199;       /* Positions 0-4, 6-9 of many get
                                  199, position 5 gets 199 */

one_ptr_to_many = [5]many_ptr_to_many;
                               /* Assign pointer to many2 */

*one_ptr_to_many = 1999;       /* All positions of many2 get 1999
                                  */

many_ptr_to_many = (int:S *:S)malloc((size_t:S)sizeof(int:S));
                               /* Each position points to an int in
                                  its own position */

one_ptr_to_many = [5]many_ptr_to_many;
                               /* Converts position-oriented
                                  parallel pointer to nonparallel
                                  pointer */

*one_ptr_to_many = 19999;      /* Undefined behavior */
```

There are twelve 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 a);
```

Description

The **`dimof`** function extracts the dimension of the shape **s** in the specified axis **`a`**. If the axis specified by **`a`** is outside the range of the rank specified for shape **`s`**, or if shape **`s`** is partially assigned or fully unassigned, the behavior is undefined.

Returns

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

Examples

```
shape [10][20][30]S;
shape []T, U;
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 */

dimof(S,5);                    /* Undefined behavior */
dimof(T,0);                    /* Undefined behavior */
dimof(U,0);                    /* Undefined behavior */
```

4.14.1.2 The `layoutof` Function [NEW]

Synopsis

```
#include <dpce.h>
dpce_layout_t layoutof(shape s, axis a);
```

Description

The **`layoutof`** function extracts the layout specification for shape **`s`** along the designated axis **`a`**. The return value's spec member indicates whether the layout specification is a **`block`** or **`scale`** specifier, and its size member indicates the corresponding specification. If shape **`s`** is fully assigned, the returned layout specification is what was specified in the declaration of the shape type; otherwise it will be a **block** specification of 0. If the axis specified in **`a`** is outside the range of the rank of shape **`s`**, or if the shape **`s`** is not fully assigned, the behavior is undefined. This function gives access to the user-defined layout specification. (See §3.5.4.4.)

Returns

The **`layoutof`** function returns the extracted layout specification.

Examples

```
shape [1024 block (256)]S;
shape T;
dpce_layout_t S_layout;

layoutof(S,1);                /* Undefined behavior */
layoutof(T,1);                /* Undefined behavior */

S_layout = layoutof(S,0);     /* Extract layout */

/* Use extracted layout */
{ shape [2048 block (layoutof(S,0))]T;
  ...
}

T = S;                        /* T becomes fully assigned, gets
                                 layout specification from S, so
                                 the next expression evaluates
                                 to 1 */
layoutof(S,0) == layoutof(T,0);
```

4.14.1.3 The `newshape` Function [NEW]

Synopsis

```
#include <dpce.h>
shape newshape(int rank, int *dimensions,
               dpce_layout_t *specs);
```

Description

The **`newshape`** function returns a shape value of the specified **`rank`** and **`dimensions`**, initializes the context so that all positions are active, and initializes the layout for the shape value as specified by the **`specs`** 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 **`specs`** argument is an array of *n* **`dpce_layout_t`** specifications for the shape value to be returned; each layout specification shall be a valid **`dpce_layout_t`**; otherwise, the behavior is undefined.

Returns

The **`newshape`** function returns the initialized shape value.

Examples

```
shape [10]S;
shape []T;
shape U;
int ten = 10;
int thousand = 1000;
int dims[3] = {50,50,20};

dpce_layout_t specs_1[] = { {0, DPCE_BLOCK} };
dpce_layout_t specs_2[] = { {10, DPCE_BLOCK} };
dpce_layout_t specs_3[] = { {50, DPCE_BLOCK},
                            {50, DPCE_BLOCK},
                            {20, DPCE_BLOCK} };

S = newshape(0, 0, 0);   /* Undefined */

S = newshape(1, &ten, specs_1);
                        /* Returns 1-dimensional shape value with
                           default distribution */

T = newshape(1, &thousand, specs_2);
                        /* Returns 1-dimensional shape value with
                           blocks of 10 elements per partition;
                           makes T a fully assigned shape */

U = newshape(3, dims, specs_3);
                        /* Returns 3-dimensional shape value with
                           no distribution; makes U a fully
                           assigned shape */
```

4.14.1.4 The `nodeof` Function [NEW]

Synopsis

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

Description

The **`nodeof`** function extracts the mapping of positions to nodes from the layout component of the given shape **`s`**, where each node in the implementation has a unique integral designation. If the shape **`s`** is not fully assigned, the behavior is undefined. This function gives access to the implementation-defined layout for the given shape.

When this function is executed in a nodal environment the node designation in each position will be that of the node on which it is executing.

Returns

The **`nodeof`** function returns a parallel **`int`** of shape **`s`** whose value at each position is the unique node designation on which each position of the given shape is mapped by the shape's layout.

Examples

```
/* Assume for the following example that <dpce.h> defines:

    shape [2]physical;

   and

    nodeof(physical) => { 0, 1 }

   That is, the unique node designation for the two nodes
   of this implementation are 0 and 1.
*/

shape [10]S, [10 block 3]T, U;

nodeof(S);              /* Denotes parallel int with values:
                           { 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 } */

nodeof(T);              /* Denotes parallel int with values:
                           { 0, 0, 0, 1, 1, 1, 0, 0, 0, 1 } */

nodeof(U);              /* Undefined behavior */
```

4.14.1.5 The `nodepositionsof` Function [NEW]

Synopsis

```
#include <dpce.h>
int:physical nodepositionsof(shape s);
```

Description

The **`nodepositionsof`** function computes the total number of positions of the given shape **`s`** that are mapped to each node of the implementation environment. If the shape **`s`** is not fully assigned, the behavior is undefined.

Returns

The **nodepositionsof** function returns a parallel **`int`** of **physical** shape whose value at each position is the total number of positions of shape **`s`** mapped to the node that contains that element of shape **`physical`**.

Examples

```
/* Assume for the following example that <dpce.h> defines:

    shape [2]physical;

   and

    nodeof(physical) => { 0, 1 }

   That is, the unique node designation for the two nodes
   of this implementation are 0 and 1.
*/

shape [10]S, [10 block 3]T, U;

nodepositionsof(S);           /* Denotes parallel int {5, 5} */
nodepositionsof(T);           /* Denotes parallel int {6, 4} */
nodepositionsof(U);           /* Undefined behavior */
```

4.14.1.6 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 the shape pointed to by **`s`** and having element size of **`bsize`** bytes, and associates the given parallel object with the shape object pointed to by **`s`**.

The shape pointed to by **`s`** must be fully assigned when **`palloc`** is called; otherwise, the behavior is undefined.

Returns

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

The return value usually needs to be cast to a particular parallel type.

Examples

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

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

x = (int:S *)palloc(&S,sizeof(*x)); /* Casting to pointer to
                                       specific parallel type */

palloc(&U, sizeof(*x));             /* Undefined */
```

4.14.1.7 The `pcoord` Function [NEW]

Synopsis

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

Description

The **`pcoord`** function is a parallel axis coordinate value constructor. For a given axis specifier **`a`** and a given shape **`s`**, this function returns a parallel **`int`** of shape **`s`** whose value at each position is initialized to the coordinate at the position in the given axis. If **`a`** is an axis number that is outside the range of the rank of shape **`s`**, or if **`s`** is not fully assigned, the behavior is undefined.

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(S,0);        /* Parallel int of shape S having value 0
                           in position [0], 1 in position [1],
                           ..., and 9 in position [9]. */

y = pcoord(T,1);        /* Parallel int of shape T having value 0
                           in positions [0,0] and [1,0], and 1
                           in positions [0,1] and [1,1]. */

pcoord(T,3);            /* Undefined behavior */
```

4.14.1.8 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 behavior */
pfree(x);                            /* Undefined behavior */
```

4.14.1.9 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. If the shape **s** is not fully assigned, the behavior is undefined.

Returns

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

Examples

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

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

positionsof(T);                /* Undefined behavior */
```

4.14.1.10 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**. The sequence of values generated in the active positions of the return value is implementation-defined. If the shape **s** is not fully assigned, the behavior is undefined.

Returns

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

4.14.1.11 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. A call to **psrand** affects all subsequent calls to **prand** regardless of the shape argument to **prand**.

Returns

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

4.14.1.12 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`**. If the shape **`s`** is not fully assigned or partially assigned, the behavior is undefined.

Returns

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

Examples

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

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

rankof(T);                /* Undefined behavior */
```

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.

```
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) elemental;
```

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.

```
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) elemental;
```

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, if any, are given for each category of extension in the subsections of this appendix.

A.1 PARALLEL CONTROL FLOW CONSTRUCTS

Although proposed in [13], [20], and [23] and generally thought to be useful, the group did not reach consensus on the desired semantics. Implementors may choose to revisit this issue. Meanwhile, the **where** statement provides adequate functionality.

A.2 FORALL STATEMENT

This was proposed in [26], but was not adopted because it added no new expressive power to DPCE and because it lacked performance transparency. These limitations were noted in [26] itself.

A.3 ITERATORS

Iterators are being separately pursued by another X3J11 subgroup whose working document is [30, 32]. The idea was not added to DPCE for the same reasons as the FORALL statement.

A.4 BIT DATA TYPES

Although not a formal X3J11/NCEG paper [31], a formal proposal was presented to DPCE on SRC's bit oriented extensions at its meeting in September, 1993. The proposal was not incorporated into the current DPCE proposal because of time and resource constraints.

A.5 I/O LIBRARY

Although considered desirable, a parallel I/O library was left for future extensions.

A.6 DYNAMIC LAYOUT

Dynamic layout would provide a facility for reorganizing data, e.g., between stages of a calculation. Such a feature is likely to be communication-intensive, and therefore would lack performance transparency. The user is always free to explicitly move data with the existing functionality to redistribute it.

A.7 ARRAYS AS FIRST CLASS OBJECTS

The DPCE proposal could be implemented more elegantly and robustly if C arrays were simply made first class objects. This would break enormous amounts of existing C code, though, so parallel objects were introduced as a replacement for first class arrays.

A.8 OVERLOADING

Adding a general overloading mechanism would provide a way to manage the redeclared library functions proposed here, as well as allowing user-written overloaded libraries. However, this was deemed to place an excessive burden on the implementation, and was one of the C* features that was not included in DPCE [10]. The use of elemental functions for providing both parallel and nonparallel functionality for library functions and user-written functions alleviates the immediate need for this extension.

A.9 CURRENT SHAPE

The concept of a current shape can be useful in code intended to be adaptable to various shapes at run time. However, it is more restrictive than the language needs to be. Most of its functionality was subsumed by shape **`void`**. Current shape is discussed in [19].

A.10 INTERMEDIATE SHAPE EQUIVALENCE TEST

The intermediate shape equivalence test in C* requires that the shapes of operands in binary or ternary operations must be derived from the same shape variable name. This name equivalence test provides a compile time means for detecting when two shapes may not be the same at run time. The test is only performed when all operands are derived from shape variables; the compiler is not required to track all possible shape assignments to determine the equivalence of shapes and, hence, may report errors where none exist. Implementations are encouraged to provide a compile-time option that would enable this test, issuing warnings where C* reports errors.

A.11 C++ COMPATIBLE PARALLEL INDEXING SYNTAX

The committee adopted numerous syntax changes to ANSI/ISO C. In many cases these syntax changes will also conflict with C++ syntax. The decision to add syntax was not capricious, but rather in each case the addition was viewed as necessary to best support programmers in the writing of DPCE programs.

For example, the **`where`** statement was added because the committee felt the difference between conditionalization (altering the control flow) and contextualization (deactivating positions of a shape) is significant. Therefore, providing contextualization by overloading the **`if`** statement was rejected as being potentially confusing to programmers. Also, the option of providing contextualization via a set of library functions to explicitly manipulate the context of a shape was rejected as being too low-level and error-prone. In particular, the use of statement syntax allows the compiler to automatically reestablish the prior context upon exit from the **where** statement.

There was much discussion about the choice of syntax for the parallel index operator. Initially the discussion centered on whether parallel indexing needed to be separated from array subscripting, and to some extent mirrored the debate on whether arrays and parallel objects needed to be supported separately. Eventually the committee concluded that the potentially different memory layout requirements of arrays and parallel objects justified separating the concepts. Therefore, the committee also separated array subscripting and parallel indexing, which aided the transparency of programs by in effect highlighting the potentially expensive parallel index operation.

However, even after these decisions were reached, questions remained as to which syntax to use for the parallel index operator. The choice of using square brackets as a prefix operator, as opposed to the use of square brackets as a postfix operator for array subscripting, reflects the committee's desire to build upon the similarity of parallel indexing and array subscripting while emphasizing the fundamental differences.

A proposal was made to better support compatibility with C++ by providing parallel indexing via the overloading of the function call syntax. The committee felt that the square bracket indexing notation would be more intuitive to the C programmer. In addition, while overloading the function call operator might seem to better support C++ implementations of the DPCE concepts for initial experimentation and evaluation, the committee noted that a C++ emulation of parallel indexing would be unable to support parallel indexing as both lvalues and rvalues. To get the correct semantics requires knowledge of how the result will be used, which is not available to the implementation of an overloaded C++ operator. Furthermore, use of function call syntax for parallel indexing would change the precedence of this operation. Finally, the committee felt that, since parallel indexing was just one of several desired syntax changes (such as the **`where`** statement discussed above), there was no overriding reason to take extraordinary steps to obtain C++ compatibility in this one case.

A.12 SLICING OF ARRAYS

Extending the slicing proposal to be allowed in array subscripting as well as parallel indexing was discussed, and while deemed a desirable and straight-forward extension, it was determined to be beyond the scope of the DPCE proposal.

INDEX