WG14/N340
X3J11/94-025

From: Frank Farance
Date: 1994-04-29
WG14 Document Number: WG14/N340
X3J11 Document Number: X3J11/94-025
Subject: DPCE Array Slicing Proposal (Revision 3)

ABSTRACT

Array slices are features of Fortran 90 and APL that are desirable for applications. Array slices can choose rectangular subsections (e.g., columns 2-4 and rows 3-5), patterned subsections (e.g., all odd numbered rows 1-9), and irregular subsections (e.g., columns 2, 5, 1), multi-dimensional lists (e.g., elements {3,4}, {2,1}, {3,3}), or all the elements (e.g., elements 0-9 in an array of length 10).

The following is the third revision of the "Array Slicing Proposal" features to be added to the DPCE base document.

==========

TABLE OF CONTENTS

==========

1. ARRAY SLICING AND SELECTOR SEMANTICS

This section develops an abstract selector notation and semantics. The notation is used to demonstrate constraints that exist within DPCE with respect to shape, layout, and context.

1.1 Abstract Selection Notation

For the purposes of this discussion, I will choose the following abstract notation for "selectors" (i.e., a value that specifies what is selected). Note: This is *not* proposed notation for DPCE -- the purpose of using different notation is that it distinguishes the abstract notation from the actual proposal.

<F:L> means choose indexes F (first) through L (last). For example, <2:5> means choose indexes 2, 3, 4, 5.

<F:L:S> means choose indexes F (first) through L (last) skipping every S (stride) indexes. For example, <3:9:2> means choose 3, 5, 7, 9.

<> means select all the indexes.

{ A1, A2, ..., AN } chooses indexes A1, A2, ..., AN. For example, {2,5,1} chooses indexes 2, 5, 1.

{ {AX1,AY1,AZ1,...}, {AX2,AY2,AZ2,...}, ..., {AXN,AYN,AZN,...} } means choose by a multi-dimensional list. For example, { {3,4}, {2,1}, {3,3} } chooses 3 points from a 2 dimensional array.

1.2 Selection of Left, Right, Or Both Indexing

The first question is whether slicing applies to C arrays (right indexing), parallel objects (left indexing), or both.

Let's say slicing were to apply to C arrays (right indexes). The result of an array slice would produce some subset of the original array. We would need to talk about its rank and dimensions ("shape" in a Fortran and APL sense, not a DPCE sense). Splitting shape (rank and dimensions), layout, and context were rejected at the Pleasanton DPCE meeting (see APL/VLA proposal). Thus, the only choice remaining is slicing of parallel objects (left indexing).

The following are examples of using this notation for left indexing (again, note this notation is only a convenience for discussion). The result of slicing a parallel object is another parallel object.

```
int:(10) A;
int:(5)(5) B;

/*#1*/ <3:5>A        /* selects (3)A, (4)A, (5)A */
/*#2*/ <3:7:2>A      /* selects (3)A, (5)A, (7)A */
/*#3*/ <>A           /* selects (0)A, (1), ..., (9)A */
/*#4*/ {3,7,1}A      /* selects (3)A, (7)A, (1)A */
/*#5*/ {1,1,1,1,2,2,2,2,3,3,3,3} /* selects
                (1)A, (1)A, (1)A, (1)A,
```

```
                    (2)A, (2)A, (2)A, (2)A,
                    (3)A, (3)A, (3)A, (3)A */
/*#6*/ <1:3>{4,0}B    /* selects
                    (1)(4)B, (1)(0)B,
                    (2)(4)B, (2)(0)B,
                    (3)(4)B, (3)(0)B */
/*#7*/ {{3,4},{2,1},{3,3}}B /* selects (3)(4)B, (2)(1)B, (3)(3)B */
```

Examples #1, #2, #4, and #7 all produce a result that has a rank of 1 and dimensions of 3 (i.e., a one-dimensional array of length 3). Example #3 results in a one-dimensional array of length 10. Example #5 results in a one-dimensional array of length 12. Example #6 results in a two-dimensional array of length 3x2.

1.3 Shape/Type Compatibility

In DPCE C, the rank and dimensions of the result *and* which "shape" it is derived from are *very* important. For example:

```
    shape (10) S;
    shape (10) T;

    int:S C;
    int:T D;
    int:T E;
    int:(10) F;
    int:(10) G;

/*#8*/ D+E    /* OK because derived from same shape. */
/*#9*/ D+C    /* Even though same rank and dimensions, incompatible
            types because of derived from different shapes. */
/*#10*/ F+G   /* Even though same rank and dimensions, incompatible
             types because of derived from different shapes
             (different anonymous shapes). */
```

Even though F and G are of length 10, they are incompatible types (see examples of incompatible types in DPCE 3.1.2.6). So let's say you wanted to multiply the first 3 elements of A by the second 3 elements of A and sum the product:

```
    shape (10) S;
    int:S A;
    int I;

/*#11*/ <0:2>A;       /* has shape int:(3) */
/*#12*/ <3:5>A;       /* has shape int:(3) */

    /* Reminder: "+=" is the sum reduction operator */
/*#13*/ I = += <0:2>A * <3:5>A; /* Illegal - incompatible types */
```

Expressions #11 and #12 result in the same rank and dimensions (i.e., a one-dimensional parallel object of length 3), but incompatible because they because they have different anonymous shapes. To fix this, we need to add a shape for an intermediate value within the expression.

```
shape (10) S;
shape (3) U;
int:S A;
int I;

/*#14*/ <0:2>A;       /* has shape int:(3) */
/*#15*/ <3:5>A;       /* has shape int:(3) */

/*#16*/ I = += (int:U) <0:2>A * (int:U) <3:5>A;
```

(Pardon me if I got the exact syntax of the "reshape" incorrect, however the intent is clear: the purpose is to make the two slices type-compatible because the slices, otherwise, would have different anonymous shapes. See 3.3.4 in Proposed Changes, below.)

In expression #16, the subexpressions must be cast to the same type. Since slicing usually produces a result different (or bigger, see #5 above) than the original shape, temporary shapes would need to be declared for each intermediate value, OR the target shape is included in the selector notation:

```
shape (10) S;
shape (3) U;
int:S A;
int I;

/*
 * The following is a modified selector notation that includes
 * the target shape.
 */
/*#17*/ <U,0:2>A;     /* has shape int:U */
/*#18*/ <U,3:5>A;     /* has shape int:U */

/*#19*/ I = += <U,0:2>A * <U,3:5>A;
```

1.4 Interaction of Shape and Context

Although #19 *might* be slightly more convenient, the problem still remains: all intermediate expressions must have a shape declared for them. The constraints we are trying to work around are:

(Rule #1) Shape compatibility is dependent upon the *name* of the shape it is derived from, not the "value" of the shape (i.e., its rank and dimensions).

(Rule #2) Shapes must be explicitly declared (i.e., a name created) to facilitate shape compatibility.

Let's say we relax the type compatibility rules to: rank, dimensions, and layout must be the same (thus, removing context compatibility). The relaxed rules would cause us to run into other problems with contextualization:

```
shape (10) S;
shape (3) U;
shape (3) V;
int:S A;
int:V W;
int I;

/*
 * The following is a modified selector notation that includes
 * the target shape.  The type compatibility rules are reduced
 * to comparing only rank, dimensions, and layout, not context
 * equivalence (which would only come from same name
 * derivation in this example).
 */

(0)W = 3;
(1)W = 0;
(2)W = 1;

where ( W != 0 )
{
/*#20*/     <U,0:2>A;     /* has shape int:U == int:(3) */
/*#21*/     W;            /* has shape int:V == int:(3) */

        /* which positions are operated on? */
/*#22*/     I = += <U,0:2>A / W;
}
```

The "where" statement causes only those positions whose value is non-zero (to avoid the divide by zero) to participate in the operations. But, presumably all values of the shape U are turned on, so it is not clear what the result is in #22 when two expressions participate with different contexts. This is a result of:

(Rule #3) Context is tied to shape and, therefore, associated with the storage of the shape and all derived objects.

Because of Rule #3 and the ambiguity of expression #22, we have to reinstate rule #1 (compatible types means derived from the same named shape). With Rules #1 and #3, there would be no ambiguity because all subexpressions would be derived from the same shape and, therefore,

have identical context.

1.5 Interaction of Shape and Layout

The issue of layout has been ignored here. Shape/type compatibility requires identical layout. DPCE does not allow incompatible layouts to interoperate for performance reasons. It is assumed in all examples that the layouts are the same. If layout compatibility were relaxed, it would not change the semantics of slicing. Therefore, the discussion of shape vs. layout is unrelated to slicing.

1.6 Summary

(1) DPCE slicing can only apply to left indexes, not right indexes.
(2) Shape/type compatibility requires more than similar rank and dimensions.
(3) Shapes must be declared, i.e., create a name, to facilitate derivation, e.g., two objects have shape S, not "they both have 10 elements".
(4) Context is tied to shape, so it cannot be separated.
(5) Layout is unrelated to slicing semantics.

==========

2. SELECTION MECHANISMS WITHIN DPCE C

The section uses the existing "dot product" selection mechanism within DPCE. The dot product mechanism is developed as far as possible, but the end result is unsatisfactory: verbose syntax and/or large temporary values for simple selections. The remaining choice is to create a third type of array indexing: Parallel Cross Product Selection.

2.1 Dot Product Selection

When a parallel object is indexed with a scalar, the resultant object is *still* the same shape (exception: when all indexes are scalar; see DPCE 3.3.3.5).

```
shape (2)(3) S;
int:S A;

/*#23*/ (.)(1)S; /* selects (0)(1)S, (1)(1)S, (0)(1)S, (1)(1)S,
                     (0)(1)S, (1)(1)S */
/*#24*/ I = += (.)(1)S;
```

Expression #23 is *not* a one-dimensional array of length 2, but a two-dimensional array of length 2x3. In expression #24, "I" is assigned the value:

```
3 * ( (0)(1)S + (1)(1)S ) /* unexpected result of reduction */
```

rather than:

```
(0)(1)S + (1)(1)S
```

The reason for this is that DPCE C uses a "dot product" selection for left indexing and a "cross product" selection for right indexing.

2.2 Cross Product Selection

The terms "dot product" and "cross product" have specialized definitions for the purposes of this discussion. The mathematical definitions don't apply exactly. In the following cross product examples, I am selecting on "left" indexes. This is used for comparing the dot vs. cross products and for a rationale for creating a new feature in DPCE C: parallel cross product selection.

Cross Product Selection - Each dimension has its own selection criteria. In the following example, in the first dimension, indexes 1, 2, and 3 are selected. In the second dimension, indexes 0 and 1 are selected. The resultant slice has the dimensions of each of the selections, e.g., 3x2 array. The number of elements is the (multiplicative) product of each of the dimensions, e.g, 3 * 2 = 6.

```
shape (10)(20) S;
int:S A;

/*
 * The following example uses the abstract notation for
 * selection.  The actual notation will be introduced shortly.
 */

/*#25*/ <1:3><0:1>A    /* selects (1)(0)A, (1)(1)A, (2)(0)A, (2)(1)A,
                          (3)(0)A, (3)(1)A */
```

Dot Product Selection - The selection within an N-dimensional array is accomplished by N parallel objects (of shape T), each providing one dimension of the selection. In the example below, three elements (#27) are selected from a two-dimensional array (#26). The resultant shape of the selection (#30) is the shape of the selection lists (i.e., "int:(3)").

```
/*#26*/ shape (10)(20) S;
/*#27*/ shape (3) T;
    int:S A;
    int:T T1,T2;

    /*
     *    T1  T2   (S is two dimensions ==> two lists: T1, T2)
```

```
      *     | |
      *     V V
      *     (3)(4) --+---- List of positions selected.  There are
      *     (2)(1) --+     three positions selected, so each array
      *     (3)(3) --+     (i.e., T1, T2) must be of length 3.
      */
/*#28*/ (0)T1=3; (1)T1=2; (2)T1=3;    /* selecting first dimension */
/*#29*/ (0)T2=4; (1)T2=1; (2)T2=3;    /* selecting first dimension */
/*#30*/ (T1)(T2)S;          /* selects (3)(4)A, (2)(1)A, (3)(3)A */
```

2.3 Dot Product vs. Cross Product

Dot product selection is more general than cross product selection. However, dot product selection requires itemizing each position selected, whereas cross product selection can specify a pattern.

For any rectangular array subsection, a dot product selection will require a list the size of the "product" of the dimensions, while a cross product selection will require a list the size of the "sum" of the dimensions. To select a 10x20 subsection of the array in #31 (e.g., "<0:9><0:19>A" using the notation above), the dot product selection requires two lists of 200 elements, 400 in total (10x20x2). The cross product selection requires two lists, one of 10 elements (i.e., the numbers 0 through 9) and one of 20 elements (i.e., the numbers 0 through 19), 30 in total (10+20). In short, dot product selection may require large temporary arrays for simple array subsections.

```
      shape (100)(100) S;
/*#31*/ int:S A;
```

2.4 Adapting Dot Product to Get Cross Product

To use existing DPCE dot product selection to get array subsections would require:

```
      shape (100)(100) S;
      shape (10)(20) T;
      int:S A;
      int:T T1,T2;

      /*
       * GI(dimN,selector list,...) is an index generator function
       * that generates indexes for all selectors, then strips out
       * only the desired dimension "dimN".
       */
/*#32*/ T1 = GI(1,<0:9><0:19>);
/*#33*/ T2 = GI(2,<0:9><0:19>);
/*#34*/ (T1)(T2)A;
```

Expressions #32 and #33 would require an index generator function. Note that the index generator function must know about *all* selectors even if it is generating the first index. Expression #34 could be written on a single line as in expression #35. The cast to "shape T" is required because of Rules #1-3 above.

```
/*#35*/ ((int:T) GI(1,<0:9><0:19>))((int:T) GI(2,<0:9><0:19>))A
```

2.5 Parallel Cross Product Selection Notation

This notation (#35) lacks brevity. The only choice left is to create the third type of array indexing: parallel cross product selection. This notation uses double brackets to indicate the cross product selection method. The following notation is adapted from above:

```
/*#36*/ ((first;length;stride))
/*#37*/ ((first;length))
/*#38*/ ((first:last:stride))
/*#39*/ ((first:last))
/*#40*/ ((index))
/*#41*/ (())
```

The notation that uses colons is designed to look like the Fortran notation. The notation that uses semicolons is similar to the C "for" statement syntax.

```
shape (10)(20) T;
int:T B;

((2:3))((3;5))B; /* shape is int:(2)(5) */
```

The single index notation (#40) is different from the DPCE notion of a single index:

```
shape (5)(10) S;
int:S A;

(,)(1)A; /* shape is still int:5:10 */
(())((1))A; /* shape is scalar int:5 */
```

The all indexes notation (#41) selects all indexes for that dimension.

```
shape (10)(20) T;
int:T B;

((4))((3))B;   /* shape is scalar int */
(())((3))B;    /* shape is int:(10) */
((4))(())B;    /* shape is int:(20) */
(())(())B;     /* shape is int:(10)(20) */
```

2.6 Intermediate Types are Still Required

Even with parallel cross product selection, shapes still must be declared for intermediate and temporary values -- there is no working around this because of Rules #1-3. So the following is still required:

```
shape (10) S;
int:S A;
shape (3) T;
int I;

I = += (int:T)((0:2))A * (int:T)((3:5))A;
```

2.7 Summary

(1) The DPCE all indexes notation (either "." or "pcoord(i,S)") is not the same as the cross product notation. The cross product notation reduces the result by one dimension for each dimension that has a scalar index.
(2) The use of dot product selection with index generators would might require large temporary values that would be unnecessary with a cross product notation.
(3) The use of dot product selection with index generator functions would produce verbose syntax.
(4) A third type of array indexing, Parallel Cross Product Selection, must be created since the existing DPCE dot product selection cannot be adapted.
(5) Intermediate shapes must still be declared. This problem cannot be worked around because shape/type compatibility requires the same context.

==========

3. PROPOSED CHANGES

The section contains the edits to the base document, dated 1994-04-28.

3.3.3.5 PARALLEL INDEXING

Dot Product Selection:

(expression)

Cross Product Selection:

((first;length;stride))
((first;length))
((first:last:stride))
((first:last))
((index))

(())

Constraints:

When combining parallel indexes for multi-dimensional arrays, e.g., (i)(j)A, or ((x1:x2))((y1;y2))A, all indexes must be dot product selection or all indexes must be cross product selection.

...

The expressions "first", "length", "last", "stride", and "index" shall be of integral type.

Semantics:

...

Cross Product Selection

Each notation specifies a method of selecting a subsection of an array. Each dimension of indexing may be any combination of cross product selection methods.

First/Length/Stride Selection:

"first" is the index of the first element. "length" specifies the number of elements. "stride" is the increment between each selected index. If "stride" is not specified, it is equal to 1. The following indexes are selected:

F = first, L = length, S = stride
F, F+S, F+2*S, F+3*S, ..., F+(L-1)*S

First/Last/Stride Selection:

"first" is the index of the first element. "last" is the index of the last element. "stride" is the increment between each selected index. If "stride" is not specified, it is equal to 1. The following indexes are selected:

F = first, L = last, S = stride

C = (S+L-F)/S
F, F+S, F+2*S, F+3*S, ..., F+(C-1)*S

Single Index Selection:

"index" specifies the index of a single element selected.

All Index Selection:

"(())" selects all indexes to be selected.

Resultant Shape:

For each dimension, the selection method determines the size of that dimension in the result:

First/Last/Stride - The length is Last-First+1

First/Length/Stride - The length is Length.

Single Index - This dimension is eliminated from the result. If all indexes are a single index, the result is a scalar.

All Indexes - This dimension is unchanged.

The resultant shape is the combination of dimensions of the selectors with single index selectors removed.

Examples:

Strides can be positive, negative, or non-existent:

```
shape (10) S;
int:S A;

/* first, last, stride: */
#1   ((3:9:2))A     /* selects indexes: 3, 5, 7, 9 */
#1   ((9:3:-2))A    /* selects indexes: 9, 7, 5, 3 */
#2   ((3:9))A       /* selects indexes: 3, 4, 5, 6, 7, 8, 9 */

/* first, length, stride: */
#1   ((3:4:2))A     /* selects indexes: 3, 5, 7, 9 */
#1   ((9:4:-2))A    /* selects indexes: 9, 7, 5, 3 */
#2   ((3:6))A       /* selects indexes: 3, 4, 5, 6, 7, 8, 9 */
```

The resultant shape of #1 is "int:4". The resultant shape of #2 is "int:6".

The following example shows combining a first/last selector in the first dimension with a first/length/stride selector in the second dimension.

```
shape (10)(20) T;
int:T B;

/*
```

```
 * ((1:3)) selects 1, 2, 3
 * ((0;2;4)) selects 0, 4
 */
((1:3))((0;2;4))B
/*
 * complete cross product selection:
 * (1)(0)B, (1)(4)B,
 * (2)(0)B, (2)(4)B,
 * (3)(0)B, (3)(4)B
 * resultant shape: int:(3)(2)
 */
```

The single index cross product selection is different than single index dot product selection.

```
shape (5)(10) S;
int:S A;

(.)(1)A;       /* shape is still int:(5)(10) */
(())((1))A;    /* shape is int:(5) */
```

All index selections can be used individually on any parallel dimension.

```
shape (10)(20) T;
int:T B;

((4))((3))B;   /* shape is scalar int */
(())((3))B;    /* shape is int:(10) */
((4))(())B;    /* shape is int:(20) */
(())(())B;     /* shape is int:(10)(20) */
```

3.3.4 Cast Operators

Semantics

(Remove the sentence: "The shape of the parallel type and the operand must be the same; the result is undefined if they are not they same shape.")