WG14/N353

X3J11/94-038

From: Frank Farance
Date: 1994-06-08
Document Number: WG14/N353, X3J11/94-038
Subject: Fortran 90 VLA/Data Parallel Report

This document is an analysis of features comparing the following language specifications:

MacDonald VLA Proposal
Ritchie VLA Proposal
Cheng VLA Proposal
Farance APL-VLA Proposal
DPCE Proposal
Fortran 90 (F90) Standard
High Performance Fortran (HPF) Document
APL Standard

At the 1993-12 meeting in Kona, several questions were raised:

1. Why was the APL standard used as a basis for the DPCE-APL (now APL-VLA) proposal? Answer: The F90 definition and APL definitions of "shape" (rank and dimensions) are basically the same. However, the F90 definition of "scalar to array promotions" are particularly weak -- the APL definition gives precise semantics.

2. Is there commonality between the VLA and DPCE proposals? Answer: Yes. From the perspective of data definition, the VLA proposal passes arrays by reference and the DPCE proposal passes arrays by value (ALO's -- array-like objects). Both proposals can pass shape with the pointer or the value. The following table summarizes the features with respect to *implicit* information the is passed across the function call boundary:

Layout	Shape	Pointer/Value	Description
No	No	Pointer	Traditional C pointer, or pointer to be used in subsequent explicit "paste" with shape (and layout).
No	No	Value	C object, or array passed by value. The array *may* be used subsequently with explicit "paste" with shape. Possibly an ALO (array-like object) where shape is known externally (explicit "paste").
No	Yes	Pointer	VLA prototype with "[?]", "[*]", "[:]", or "shapeis(?)", depending upon the proposal. Compiler generates code to pass shape info on the stack.
No	Yes	Value	ALO prototype with "[?]", "shapeis(?)", or "int:void", depending upon the proposal. Compiler copies value and its shape onto the argument stack.
Yes	No	Pointer	Pointer and layout info are passed. Pointer to a distributed object.
Yes	No	Value	Value and layout info are passed. Distributed object is passed on stack, e.g., pieces of object on stacks of different processors.
Yes	Yes	Pointer	Distributed VLA. Layout, shape, and pointer are passed on stack.
Yes	Yes	Value	Distributed ALO. Layout, shape, and value are passed on stack. Distributed object might be distributed on stacks of different processors.

3. How do these VLA proposals compare to each other? Answer: The attached matrix compares these proposals with respect to "shape" (rank and dimensions), "layout" (distributed memory objects), and "selectors" (indexing, slicing, scatter-gather). The following are details of the criteria used to evaluate the proposals.

This section evaluates the proposals on "shape" (i.e., rank and dimensions), pass by reference, and pass by value.

Comparing the proposals on "shape", i.e., rank and dimensions.

Variable size in array declaration: Can a non-constant expression be used in an array declaration?

Allows variable size in prototype declaration: Can a varying size array be used in a prototype?

Auto allocation/free of array: If varying size array is declared in a block, is memory allocated (e.g., on stack) on entry and freed on exit from the block?

Explicit pasting of pointer and shape: Given a separate pointer and shape, can a varying size array be declared by "pasting" the information, e.g.:

```
        int f ( int n, int *aa )
        {
                int a[n] = aa;
        }
```

Explicit pasting of pointer and shape in prototype: Can a pointer and shape be "pasted" in the prototype:

```
        int f ( int n, int a[n] )
        /*      shape, pointer */
```

Unordered explicit pasting of pointer and shape as argument: Can the shape appear after the pointer (Stallman proposal)?

```
        int f ( int a[n], int n )
        /*      pointer, shape */
```

Implicit combined value and shape as argument: The value and its shape are both passed on the stack. Also known as ALO's (array-like objects).

```
        main()
        {
                int a[17];

                f(a); /* passes shape (17) and value (whole array) */
        }

        int f ( int alo a[?] )
        {
                /* both shape and value passed */
        }
```

Pass whole array (by value) as argument: For arrays, are they passed by value (shape is not required to be passed):

```
        main()
        {
                int a[17];

                f(a); /* passes shape (17) and value (whole array) */
        }

        int f ( int alo a[17] )
        {
                /* only value passed - shape agreed upon otherwise */
        }
```

"Shape Of" operator/function: Is there some operator or function that returns the size of the dimensions of an array?

Variable rank declarations: Can arrays be declared at run-time with varying rank?

Variable rank arguments: Can prototypes define arguments with varying rank?

Reshape operation: Is there an operator or function that changes the rank and dimensions of an operand?

Explicit pasting of value and shape: Can a variable be declared by constructing is value and shape as separate components?

```
        int S:a = b; /* DPCE: shape == S, value == b */
        int shapeis(X) a = b; /* APL-VLA: shape == X, value == b */
```

Constant shapes: Once a shape is fully defined (for certain declarations), must the shape of these variables remain constant.

Shape checking on assignment: For expressions like "a = b" where the shapes must be compatible, is shape checking performed?
Legend: R=run-time, C=compile-time, O=optional

Bounds checking: Are array bounds checked a run-time?
Legend: Y=yes, N=no, O=optional, Q=quality of implementation

Fat pointers - global: Are there pointers that access data across all of distributed memory?

Fat pointers - wider than "void *": Are there pointers whose size is wider than "void *".

Fat pointers - address and shape: Are there pointers that have more information than the address, i.e., shape information is included (some kind of dope vector).

Fat pointers - address and layout: Are there pointers that have more information than the address, i.e., layout (memory distribution) information is included (some kind of dope vector).

Fat pointers - address, shape, and layout: Are there pointers that have more information than the address, i.e., shape and layout information is included (some kind of dope vector).

This section evaluates the proposals on layout (i.e., data in distributed memory systems).

Distributed memory: Can objects be distributed in discontiguous memory?

Specify layout: Can the data distribution preferences be specified in the object declaration?

Extract layout: Can the actual data distribution scheme be retrieved from (e.g., "layoutof") a distributed object?

Operate with incompatible layouts: Can two operands interact (e.g., addition) if they have incompatible layouts?

Near pointers (local and fast): Are there pointers to objects in local memory (short pointers) that have fast pointer increments?

Far pointers (global and fast): Are there global pointers (long pointers) that have fast pointer increments, but cannot walk the object across distributed memory segments?

Huge pointers (global and slow):Are there global pointers (long pointers) that can be incremented and walk the object across distributed memory segments?

This section evaluates the proposals on selectors, e.g., indexing, slicing, scatter-gather.

Vector value subscripts: Can specify arbitrary elements (e.g., elements 7, 2, 3).

```
        A[7 2 3]        /* APL */
```

Cross product: Can specify multi-dimension vector value:

```
        B[1 3][2 4 6]
```

specifies:

```
        B[1][2]
        B[1][4]
        B[1][6]
        B[3][2]
        B[3][4]
        B[3][6]
```

Dot product: Can specify arbitrary elements for multiple dimensions:

```
        [0]A1 = 1;
        [1]A1 = 2;
        [2]A1 = 3;
        [0]A2 = 4;
        [1]A2 = 5;
        [2]A2 = 6;
        [0]A3 = 7;
        [1]A3 = 8;
        [2]A3 = 9;
        [A1][A2][A3]X;          /* DPCE Specifies: */
                                /* [1][4][7]X */
                                /* [2][5][8]X */
                                /* [3][6][9]X */
```

Conbine cross and dot product: Can both styles be used simultaneously?

First-last-stride triplet slice: Array slicing specified by the first element, the last element, and the increment (stride) between elements.

First-length-stride triplet slice: Array slicing specified by the first element, the number (length) of elements, and the increment (stride) between elements.

MacDonald	Ritchie	Cheng	Farance	DPCE	F90	HPF	APL	"shape" criteria
✓	✓	✓	✓	✓	✓	✓	✓	Variable size in array declaration
✓	✓	✓	✓	✓	✓	✓	✓	Allows variable size in proto decl
✓	✓	✓	✓	✓			✓	Auto alloc/free of array
✓	✓	✓	✓	?			NA	Explicit pasting ptr + shape
✓	?	✓		?			NA	Explicit pasting ptr + shape in proto
?							NA	Unordered explicit pasting ptr + shape in proto
			✓	✓			✓	Implicit combined value + shape arg
			✓	✓			✓	Pass whole array (by value) as arg
			✓	✓	✓	✓	✓	"Shape Of" operator / function
			✓				✓	Variable rank decl's
			✓	✓			✓	Variable rank arguments
✓	✓	✓	✓		✓	✓	✓	Reshape operation
			✓	✓			✓	Explicit pasting value + shape
✓	✓	✓	✓	✓	✓	✓		Constant shapes

MacDonald	Ritchie	cheng	Farance	DPCE	F90	HPF	APL	"shape" criteria
			✓				✓	Shapes can change (cause free/alloc)
			O	C				Shape checking on assignment C=compile-time, R=run-time, O=optiona
Q	Q	Q	O	N	?	?	Y	Bounds Checking on indexes Y=Yes, N=No, O=Optional Q=Quality of implementation
			✓	?			NA	Fat pointers - global
			✓				NA	Fat pointers - wider than void *
✓	✓	✓	✓	✓	✓	✓	NA	Fat pointers - address+shape
			✓			?	NA	Fat pointers - address + layout
			✓			?	NA	Fat pointers - address, layout, shap

Farance	DPCE	F90	HPF	APL	
					Data Distribution
✓	✓		✓	✓	Distributed memory
✓	✓		✓		Specify Layout
✓					Extract Layout
✓			✓	✓	Operate with incompatible types
✓	✓	✓	✓	NA	Near pointers (local and ~~slow~~ fast)
✓				NA	Far pointers (global and fast)
✓				NA	Huge pointers (global and slow)
					~~Sect~~ Selectors
✓	✓	✓	✓	✓	Vector value subscripts
✓		✓	✓	✓	Cross product
✓	✓			✓	Dot product
✓	●			✓	Combine cross & dot products
✓		✓	✓		First, last, stride triplet slice
✓				✓	First, length, stride triplet slice