High Performance Fortran

Proposal

presented at
Rice University High Performance Fortran Forum
January 27, 1992

presented by
Digital Equipment Corporation
Massively Parallel Systems Group
MLO1-5/U46
146 Main Street
Maynard, MA 01754-2571

for additional information, contact
David B. Loveman
at the above address, or
loveman@mpsg.enet.dec.com
508-493-3704

Table of Contents

Chapter 1 - Overview

1.1 Introduction

High Performance Fortran is a programming language for high performance computing applications. Such a language must be able to express classical scientific algorithms as well as state-of-the-art data parallel algorithms. The language described herein is conformant with both the FORTRAN 77 and Fortran 90 standards and includes minimal language extensions to Fortran 90 specifically for high performance scientific computing:

- a new intrinsic inquiry function to determine the number of processors,
- a set of data distribution and association directives, and
- a simple element array assignment construct (FORALL).

High Performance Fortran is designed to be comprehensive, yet pragmatic. A comprehensive language allows application programmers the greatest flexibility in developing or migrating scientific applications for these high performance computing environments. The features of the language must also be chosen pragmatically to allow for efficient implementations of the language on a wide variety of platforms. This language supports the *data parallel* programming style for massively parallel SIMD and MIMD supercomputers as well as for pipelined multiple instruction issue RISC processors and is designed to be a core language, a de facto standard subset of a number of Fortran offerings.

1.2 Motivation and Objectives

High Performance Fortran is a superset of Fortran 90: any program that conforms to the Fortran 90 standard is a legal High Performance Fortran program. Except for the NUMBER_OF_PROCESSORS intrinsic function, the data distribution and association directives, and the FORALL statement, High Performance Fortran does not contain any other proposals for “improving” Fortran 90. Thus, we do not propose to define nested WHERE statements, additional intrinsic functions, or any of the other frequently mentioned proposals for improving on Fortran 90. Of course, implementors are free to add any or all of these, as extensions to High Performance Fortran.

We aimed for a minimality of mechanism -- our objective is to focus on those language features that “everybody” agrees on. The goal is to define an expressive and useful language that can become widely available on parallel machines, so that programmers will be able to build software that can be ported easily over a wide range of both SIMD and MIMD machines. We hope to get agreement on what we all do, in fact, agree on, and hope to have standards compliance where possible. It is expected that vendors will offer their own extensions to High Performance Fortran, and that researchers will extend it to test out their ideas. If widely accepted, some of those changes could be incorporated in future versions of High Performance Fortran. Thus High Performance Fortran is a “core language” for parallel programming.

This document consists of 5 chapters and a bibliography:

- Chapter 1 is this overview
- Chapter 2 presents High Performance Fortran as Fortran 90 compliant, and discusses some issues related to Fortran 90 compliance, notably Fortran sequence and storage association.
- Chapter 3 describes a new class of intrinsic functions, system inquiry functions, and the specific function NUMBER_OF_PROCESSORS.
- Chapter 4 presents the High Performance Fortran approach to directives and describes the data distribution and association directives.
- Chapter 5 describes the FORALL construct.
- The Appendix is a brief bibliography.

In this document, the Fortran 90 (section 1.5.2) assumed syntax rules are used, where “xyz” stands for any legal syntactic class phrase:

	xyz-name	**is**	*name*
	xyz-list	**is**	*xyz* [, *xyz*] . . .

The following syntax rule from Fortran 90 is also used:

R304	*name*	**is**	*letter* [*alphanumeric-character*] . . .

Constraint: The maximum length of a *name* is 31 characters.

1.3 Contributors

Contributors to the definition of High Performance Fortran and the development of this document include Peter A. Belmont, M. Regina Bolduc, Robert H. Halstead, Jr., David B. Loveman, David L. Presberg, Paul St. Pierre, Ian Wells, and Joseph Wisnewski. The insights of the Fortran D projects at Rice University and Syracuse University were essential. The contributions of numerous reviewers are gratefully acknowledged.

Chapter 2 - Full Fortran 90 Support

The facilities for array computations in Fortran 90 will make it the programming language of choice for scientific and engineering numerical calculations on high performance computers. Indeed, some of these facilities are already supported in compilers from a number of vendors. The introductory overview in the Fortran 90 standard states:

> Operations for processing whole arrays and subarrays (array sections) are included in [Fortran 90] for two principal reasons: (1) these features provide a more concise and higher level language that will allow programmers more quickly and reliably to develop and maintain scientific/engineering applications, and (2) these features can significantly facilitate optimization of array operations on many computer architectures.

Other features of Fortran 90 that improve upon the features provided in FORTRAN 77 are:

- Additional storage classes of objects --- The new storage classes such as allocatable, automatic and assumed-shape objects, and the pointer facility of Fortran 90 add significantly to those of FORTRAN 77 and should reduce the styles of use of FORTRAN 77 constructs which can lead to less than full computational speed on high performance computers, such as EQUIVALENCEd array objects, COMMON definitions with non-identical array definitions across subprograms, and actual-argument/dummy-argument array shape transformations.
- Modules --- The module facilities of Fortran 90 enable the practice of design and implementation using data abstractions. These facilities support the specification of modules, including user-defined data types and structures, defined operators on those types, and generic procedures for implementing common algorithms to be used on a variety of data structures. In addition to modules, the definition of *interface blocks* enables the application programmer to specify the interfaces to subprograms explicitly, allowing a high quality compiler to use the information specified to provide better checking and optimization at the interface to other subprograms.
- Additional intrinsic procedures --- Fortran 90 includes the definition of a large number of new intrinsic procedures. Many of these new intrinsic procedures support mathematical operations on arrays, including the construction and transformation of arrays. Also, there are numerical accuracy intrinsic procedures designed to support numerical programming, and bit manipulation intrinsic procedures derived from MIL-STD-1753.

In keeping with the goal of being a comprehensive language, High Performance Fortran is fully Fortran 90 conformant. The following subsections discuss some details of particular choices of language features as they pertain to this decision.

2.1 Sequence and Storage Association

For compatibility with FORTRAN 77, Fortran 90 defines two concepts that imply a linear memory model: sequence association and storage association. These concepts apply to some, but not all, data entities in Fortran 90. *Sequence association* describes the linearization of multi-dimensional objects into linear sequences of values and the mapping of linear sequences of values to multi-dimensional objects. *Storage association* describes the mapping of Fortran objects to an underlying linear memory of *storage units*. These concepts work well when the underlying computer has a linear memory; when, as is the case for distributed memory SIMD and MIMD computers, the memory is multi-dimensional, implementation and efficiency problems arise. Our approach in High Performance Fortran is to comply with the standard, but to provide a mechanism in the form of directives to allow the application developer to communicate to the compiler those cases in which the compiler can ignore sequence or storage association issues.

The Fortran 90 standard states that an actual argument that represents an element sequence (e.g., an array expression, an array element) and corresponds to a dummy argument that is an array-valued data object is *sequence associated* with the dummy argument if the dummy argument is an explicit-shape or assumed-size array. In this way, array expressions can be passed as actual arguments to subprograms whose dummy arguments are declared to be arrays of different shape and size than the array expression. The dummy has the values of the array expression in array-element order, in a shape corresponding to the declaration of the dummy. For example, a two dimensional, 10 x 10 array can be passed as an actual argument which corresponds to a one dimensional 100 element array. In addition, a single element of an array may be passed to a subroutine as the actual argument corresponding to a dummy variable that is declared to be an array. The resulting argument has the values of the actual array, beginning at the element selected, in array-element order, in a shape corresponding to the declaration of the dummy. This type of use results in dynamically reshaping arrays across procedure boundaries. For example:

```
REAL, DIMENSION(10,10) :: A, B
CALL SUBR(A,B(5,5))
. . . . .
SUBROUTINE SUBR(C,D)
REAL, DIMENSION(100), INTENT(INOUT) :: C
REAL, DIMENSION(10), INTENT(INOUT) :: D
```

The standard also defines *storage association* as "the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units." This storage association is caused by the use of EQUIVALENCE statements and by the specification of different array shapes in the same COMMON block as it is declared in different program units. For example:

```
DOUBLE PRECISION, DIMENSION(100) :: A
REAL, DIMENSION(2) :: B
COMPLEX, DIMENSION(50) :: C
EQUIVALENCE (A(1),B(1))
EQUIVALENCE (B(2),C(1))
```

or

```
SUBROUTINE SUBR1
COMMON /STOR_ASSOC/ IA
INTEGER, DIMENSION(10) :: IA
```

```
. . . . .
END SUBROUTINE SUBR1

SUBROUTINE SUBR2
COMMON /STOR_ASSOC/ CA
CHARACTER*40 :: CA
. . . . .
END SUBROUTINE SUBR2
```

A typical practice in Fortran compilers for massively parallel machines is to spread the data elements of an array across the processing units, allowing for parallel computation of array operations, but violating the rule of contiguous allocation which supports storage association on linear memory machines. Because High Performance Fortran is standards conforming, it will require the implementation of both storage and sequence association of data objects as defined in the Fortran 90 standard. A major goal of this language is to allow efficient implementation of data parallel algorithms on massively parallel machines. Thus, the ability to specify that storage association is not necessary in a given program unit, so that a compiler may make optimizing assumptions without regard to storage association, is provided through a command line switch[1]. In addition, directives are provided to specify objects which do not require storage association or sequence association, in order to allow a compiler to make optimizing assumptions regarding those objects, but still implement storage and sequence association for other objects in the program unit.

Full implementation of sequence and storage association in a compiler will have an impact on the efficiency of the generated code for programs that use these features. Equivalence of data objects will require that the compiler allocate overlapping, and possibly suboptimal, storage layouts to all objects in an equivalence ring. In addition, either true storage sequence order must be observed by the compiler, or extensive run-time checking and support, such as passing “dope” information for all parameters, both scalar and arrays, must be implemented to support the dynamic reshaping of arrays across procedure boundaries. The implementation of full storage association could be accomplished by completely “scalarizing” the program, or a more complicated run-time model may be implemented to allow parallelism where possible. It may be that early implementations of High Performance Fortran will limit the use of arrays that require sequence or storage association, as is now common practice in Fortran compilers for massively parallel machines.

2.2 Intrinsic Functions

High Performance Fortran incorporates all the intrinsic procedures described in Chapter 13 of the Fortran 90 standard. There are several issues regarding the support of vendor-specific Fortran intrinsic extensions in the context of Fortran 90, in particular the extension of the definition of these Fortran intrinsic procedures to support array arguments.

Whenever there is a semantic difference between the Fortran 90 intrinsic definitions and the

[1] Although not a part of our High Performance Fortran language proposal, we assume that compilers supporting High Performance Fortran will provide a rich collection of command line switches.

vendor-specific Fortran definitions, High Performance Fortran will use the Fortran 90 definitions in order to remain Fortran 90 conforming.

It is recommended that vendor-specific extensions to High Performance Fortran that add extra data type capabilities to intrinsic functions do so through extensions to the KIND type parameter mechanism. To maintain conformance with Fortran 90, generic intrinsic function names should be used for extensions as much as possible. In order to distinguish argument or result KINDs, as needed, the extension should provide optional KIND= parameters. For instance, the standard DPROD function takes two arguments of default REAL type and returns a result of default DOUBLE PRECISION type. To extend DPROD to form an extended-precision product of two DOUBLE PRECISION or default REAL arguments, it is useful to extend the DPROD argument list to optionally have a KIND=LONG_DOUBLE argument so that no extra intrinsic function names need be introduced.

Chapter 3 - New Intrinsic Function

In addition to all the intrinsic functions of Fortran 90, High Performance Fortran has a new intrinsic function, NUMBER_OF_PROCESSORS, which takes no arguments and returns an integer value giving the number of processors in the system. This number is expected to remain constant for (at least) the duration of one program execution. Accordingly, NUMBER_OF_PROCESSORS() is a constant expression and can be used wherever any other Fortran 90 constant expression can be used. In particular, NUMBER_OF_PROCESSORS can be used in an initialization expression or in a specification expression. None of the categories of intrinsic functions listed in Chapter 13 of the Fortran 90 standard seem quite apt to describe the nature of this new intrinsic function, so we add a new category of "system inquiry functions" and place NUMBER_OF_PROCESSORS in that category.

Note that treating NUMBER_OF_PROCESSORS as a constant expression does not force a compiler to bind the number of processors at compile time (although that is one possible implementation) -- with the right linker or code-generation technology the choice could be deferred until run time, possibly at some performance cost.

A definition of NUMBER_OF_PROCESSORS, in the style of Chapter 13 of the Fortran 90 standard is:

13.13.77a NUMBER_OF_PROCESSORS(DIM)

Optional Argument. DIM

Description. Returns the number of processors available to the program along a specified dimension or the total number of processors available to the program.

Class. System inquiry function.

Arguments.

DIM (optional) must be scalar and of type integer with a value in the range 1<=DIM<=*n* where *n* is the rank of the processor array.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result value. The result has a value equal to the extent of dimension DIM of the processor-dependent hardware processor array or, if DIM is absent, the total number of elements of the processor-dependent hardware processor array. The value is equal to or greater than one. The value of NUMBER_OF_PROCESSORS() need not be a constant, if the processor allows for a variable number of processors to execute the program. However, the value must not change during the execution of the program.

Example. For a DECmpp 12000 Model 8B with 8192 processors, the value of NUMBER_OF_PROCESSORS( ) is 8192, the value of NUMBER_OF_PROCESSORS(DIM=1) is 128, and the value of NUMBER_OF_PROCESSORS(DIM=2) is 64.

The list of alternatives in a Fortran 90 restricted expression is expanded to include NUMBER_OF_PROCESSORS, as follows:

A **restricted expression** is an expression in which each operation is intrinsic and each primary is:

1. A constant or subobject of a constant,
2. A variable that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute, or a variable that is a subobject of such as dummy argument,
3. A variable that is in a common block or a variable that is a subobject of a variable in a common block,
4. A variable that is made accessible by use association or host association or a variable that is a subobject of such a variable,
5. An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are either restricted expressions or implied-DO variables,
6. A structure constructor where each component is a restricted expression,
7. An elemental intrinsic function reference of type integer or character where each argument is a restricted expression of type integer or character,
8. One of the transformational functions REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, and TRIM, where each argument is a restricted expression of type integer or character,
9. A reference to an array inquiry function (13.10.15) other than ALLOCATED, the bit inquiry function BIT_SIZE, the character inquiry function LEN, the kind inquiry function KIND, or a numeric inquiry function (13.10.8), where each argument is either a restricted expression or a variable whose type parameters or bounds inquired about are not assumed or defined by an ALLOCATE statement or a pointer assignment, or
10. A restricted expression enclosed in parentheses, or
11. The HPF system inquiry function NUMBER_OF_PROCESSORS.

Chapter 4 - Directives

Syntactically, High Performance Fortran directives are structured comment lines that influence or specify actions to be performed by the compiler but do not change the result computed by the source program. Directives contain information that is difficult or impossible for the compiler to deduce from Fortran source code. The purpose of a directive is to advise the compiler about how the application manipulates data. The compiler can then use this advise along with automated compiler technology to improve the application's performance. Directives are a more portable form of language extension than new statements because a directive has the syntax of a comment and may be ignored by any compiler not engineered to process the directive.

A fundamental philosophy underlying the design of the High Performance Fortran directives is that they have no computational effect: in other words, a program that contains directives is expected to generate mathematically identical results (using the definition of "mathematical equivalence" from Section 7.1.7.3 of the Fortran 90 standard: "Evaluation of numeric intrinsic operations") whether directives are processed or not, provided only that the directives are used correctly (i.e., the directives are not used to make false statements about the program).

In certain cases, it is possible that the processing of a directive will cause arithmetic operations to be performed in a different order (but still a legal order according to the Fortran 90 standard) than if the directive had not been processed. Thus the results of executing the program with and without the directive may not be computationally equivalent (in the sense defined in Section 7.1.7.3 of the Fortran 90 standard), due to the non-associativity of floating-point arithmetic operations (for example), but they will be mathematically equivalent.

Compilation ("command line") options and directives may sometimes specify conflicting information to the compiler. Compilation options modify the default behavior for an entire program unit and directives allow these defaults to be locally overridden.

High Performance Fortran not only contains the definition of a standard set of directives but also has provisions for vendor-specific directives. Vendor-specific directives support performance tuning and application scaling for particular architectures. The standard directives are provided to support portability across high performance machines having various architectures. The design of the High Performance Fortran directives is based on the proposals for Fortran D from Rice and Syracuse Universities and from directives defined in existing Fortran dialects. Note that the Fortran 90 standard makes no attempt to define directives. The following categories of directives are supported in High Performance Fortran:

- Data distribution -- These directives are derived from Fortran D.
- Association -- These directives inform the compiler of situations where sequence or storage association are not required.
- Vendor-specific -- High Performance Fortran defines the framework syntax for vendor-specific directives in that it defines the syntax of all directives to be structured comment-lines, but does not define particular instances of these directives. The implementation of vendor-specific directives must coordinate any interaction with the standard directives. These directives, being vendor-specific, are not discussed further here.

4.1 General Syntax of All Directives

Directives are structured comment lines (entire lines) in both the fixed and free source form in Fortran 90. The first character is a “C”, “c”, “*”, or “!” in fixed source form, or “!” in free source form. Note that directives must appear as whole-line comments. Comments which are preceded by non-commentary statements on the same line are not processed as directives. This constraint places directives at the same syntactic level as Fortran statements.

Immediately following the first letter are a sequence of characters, with no embedded blanks, specifying the *directive-origin.* Following one or more blanks is the *directive-body* which starts with a *directive-keyword.* Within the *directive-body*, blank characters are significant[1]. The syntax of the remainder of the *directive-body* is particular to the specific directive. If a directive is continued, the last non-blank character in the comment line is an “&” and the next line repeats the *directive-origin*: thus the directive continuation line must also be in the form of a directive with the same *directive-origin* string as the initial line of the directive.

High Performance Fortran defines the *directive-origin* string to be the string “HPF$”. The letters may actually appear in upper or lower case.

Vendor-specific directives follow the same general syntax as High Performance Fortran directives, however they *may not* use “HPF$” as their directive-origin string.

For the sake of uniformity, we will henceforth assume free-form input in all examples; accordingly, all directive examples are shown using the initial “!” character that is required to begin a comment in the free Fortran 90 source form.[2]

Directives may only appear in a declarative context, as indicated in the following modified Fortran 90 syntax rule:

R207	*declaration-construct*	**is**	*derived-type-def*
		or	*interface-block*
		or	*type-declaration-stmt*
		or	*specification-stmt*
		or	*parameter-stmt*
		or	*format-stmt*
		or	*entry-stmt*
		or	*stmt-function-stmt*
		or	*data-distribution-directive*
		or	*association-directive*

[1] In contrast to Fortran 90 source form, blank characters are significant in directive bodies whether or not the program unit is in fixed source form or free source form.

[2] In later sections, syntax is given only for free-form input directives.

4.2 Data Distribution Directives

4.2.1 Overview

High Performance Fortran offers directives, derived from Fortran D, for specifying the mapping of array elements to processors in a distributed-memory multiprocessor. This mapping is specified indirectly. First, a **decomposition** must be defined. A decomposition is not an actual data object; rather, it is an abstract, Cartesian index space that defines the “shape” of a computation to be performed. A decomposition is declared using directives such as

```
!HPF$ DECOMPOSITION D1(N)
!HPF$ DECOMPOSITION D2(M,N)
!HPF$ DECOMPOSITION D3(10,10,10)
```

The above directives declare three decompositions D1, D2, and D3. D1 is a one-dimensional decomposition consisting of N points, D2 is a two-dimensional decomposition consisting of an array of M by N points, and D3 is a 10-by-10-by-10 three-dimensional decomposition.

Once a decomposition has been declared, the ALIGN directive can be used to **align** arrays or other entities with respect to it. A DISTRIBUTE directive must be used to specify the distribution of points of the decomposition among the physical processors of the system. Together, the alignment of the array to a decomposition and the distribution of that decomposition onto the processors define the **mapping** from array elements to processors.

The following are examples of ALIGN directives:

```
!HPF$ ALIGN V(I) WITH D1(I)
!HPF$ ALIGN W(I) WITH D1(I+1)
!HPF$ ALIGN X WITH D1(2)
!HPF$ ALIGN A(I,J) WITH D2(I,J)
!HPF$ ALIGN B(I,J) WITH D2(J,I)
```

When more than one array is aligned with respect to the same decomposition, an association among the elements of those arrays is established. For example, in the above directives, the array elements W(3) and V(4) -- or, more generally, W(I) and V(I+1) -- are aligned with the same point of the decomposition D1, which guarantees that they will be mapped onto the same processor, no matter what distribution is used for D1. Likewise, the above ALIGN directives ensure that the array elements A(I,J) and B(J,I) will always be allocated to the same processor -- in other words, the matrix A will be aligned with the transpose of B.

Alignment may also be specified for several kinds of entities other than arrays: for example, scalar variables and structures. However, since such entities cannot be subscripted directly, they cannot be spread across decomposition points the way arrays can -- the entire entity must be aligned with a single decomposition point[1]. Thus the third directive in the above set of ALIGN examples specifies that the scalar variable X be aligned with the decomposition point D1(2) -- the same point to which V(2) and W(1) are aligned.

[1] Unless, of course, it is replicated.

Whereas ALIGN directives allow the programmer to indicate logical relationships between the most closely associated array elements in the computation to be performed, DISTRIBUTE directives allow the programmer to specify a distribution of decomposition points onto the hardware that makes a good tradeoff between load balance (which generally favors spreading neighboring decomposition points across a large number of processors) and communication costs (which may favor keeping neighboring decomposition points on the same or neighboring processors). DISTRIBUTE directives are thus more likely than ALIGN directives to reflect the characteristics of the architecture on which the program is to be executed. Therefore the DISTRIBUTE directives in a program may be the most likely statements to be changed when the program is ported to a new architecture. (This is a principal reason for factoring the mapping process into separate alignment and distribution steps, instead of directly specifying the mappings of individual arrays.) Nevertheless, the vocabulary used in the DISTRIBUTE directive is quite architecture-neutral, so that no changes in DISTRIBUTE directives will be required when porting between architectures unless the tradeoffs between communication, computation, and synchronization costs on the new architecture are so different that they dictate a qualitatively different approach to the mapping of the computation.

Examples of DISTRIBUTE directives are

```
!HPF$ DISTRIBUTE D1(CYCLIC)
!HPF$ DISTRIBUTE D2(BLOCK,BLOCK)
```

The first directive specifies that the processors of the machine are to be considered as a one-dimensional array of processors (the implementor is encouraged to select that array so that communication between logically neighboring processors in the array is inexpensive), and that points of the one-dimensional decomposition D1 are to be associated with processors in that array so that neighboring points of D1 are associated with logically neighboring processors in the array, “wrapping around” to the beginning of the processor array if D1 has more points than the machine has processors.

The second directive specifies that the processors are to be considered as a two-dimensional array, and the points of the two-dimensional decomposition D2 are to be associated with processors in this array in a “blocked” fashion: neighboring points of D2 are divided into blocks whose size along each dimension is the quotient of the size of that dimension of D2 divided by the size of that dimension of the logical processor array, and neighboring points of D2 are associated with the same processor except at a block boundary, in which case neighboring points of D2 are allocated to logically neighboring processors along that dimension of the logical processor array.

Once the distributions of D1 and D2 have been specified, the mappings of any arrays aligned with D1 or D2 are determined, and the elements of these arrays are mapped to the processors to which their corresponding decomposition points are associated by the DISTRIBUTE directives used to specify the distributions of D1 and D2.

In High Performance Fortran, the mapping directives are not executable statements. All of the mapping directives function like declarations, and generally follow the same rules about where they must appear in a program, what order they must appear in, and what kinds of arithmetic expressions may appear within them as are followed by other declarations, such as DIMENSION statements. As a result of this restriction, High Performance Fortran has the property that the form of the mapping of any array is always known statically, even if its numerical parameters (such as the block size) may not be known until run time.

Since there are likely to be a very large number of arrays declared in a realistic application program, it would be burdensome for the programmer to have to specify all of the data distribution details for every array. Thus, a compiler implementation should use a default data distribution rule (or automatic optimization methods, if available) for any arrays for which the user gives no further details. The data distribution directives provide the application programmer the means to tune data distribution effects for computationally intensive subprograms of the application where the compiler's decisions prove inadequate.

If no alignment is specified for a given data object, then the mapping to be used for that object is the implementation's free choice -- it should not be assumed that all such objects will be aligned in the same way, or that any given such object will be aligned in the same way at all moments in time. But (as always) the implementation must make sure that the mapping choices it makes have no visible semantic effect. The intent of allowing mappings to go unspecified, and of defining the meaning of unspecified mappings as we do, is to provide a growth path for smart compilers to take over more and more of the job of determining mappings, and over the long term to reduce the burden on programmers of doing this kind of tuning. In the shorter term, while programmers may often need to specify the mapping of some objects in order to optimize performance, the ability to leave other mappings unspecified allows the compiler to use the specified mappings as a framework around which to fit automatically the (unspecified) mappings of other objects as best it can, and implementors are encouraged to implement High Performance Fortran systems that have this capability.

The syntactic specification for the data distribution directives is

data-distribution-directive	**is**	*decomposition-directive*
	or	*distribution-directive*
	or	*alignment-directive*

4.2.2 The DECOMPOSITION Directive

decomposition-directive	**is**	!HPF$ DECOMPOSITION *decomposition-spec-list*
decomposition-spec	**is**	*decomposition-name* (*explicit-shape-spec-list*)

Constraint: The maximum rank is seven.

Constraint: The *decomposition-name* must not be the name of any other entity in the program unit.

R513	*explicit-shape-spec*	**is**	[*lower-bound* :] *upper-bound*
R514	*lower-bound*	**is**	*specification-expr*
R515	*upper-bound*	**is**	*specification-expr*
R734	*specification-expr*	**is**	*scalar-int-expr*

Constraint: The *scalar-int-expr* must be a restricted expression.

This directive specifies a list of decompositions, each of which specifies the name, rank, and extents of an independent decomposition. Each decomposition defines a problem domain as a named n-dimensional coordinate space. No storage is allocated for a decomposition.

4.2.3 The DISTRIBUTE Directive

Since a mapping cannot be known without knowing both the alignment and the distribution, it is mandatory for a program to include a DISTRIBUTE directive for every decomposition that has one or more entities aligned to it. (Since both the DECOMPOSITION directive and the DISTRIBUTE directive are declarations, there can be at most one DISTRIBUTE specification for each decomposition that is declared, so it will usually make sense to place the DISTRIBUTE directive for a decomposition close to the DECOMPOSITION directive that declares it.)

distribute-directive	**is**	!HPF$ DISTRIBUTE *distribution-spec-list*
distribution-spec	**is**	*decomposition-name* (*axis-descriptor-list*)
axis-descriptor	**is**	BLOCK [(*aspect-expr*)]
	or	CYCLIC [(*aspect-expr*)]
	or	BLOCK_CYCLIC (*size-expr* [, *aspect-expr*])
	or	:
aspect-expr	**is**	*specification-expr*
size-expr	**is**	*specification-expr*

This directive specifies the mapping of a named decomposition, and all arrays aligned to that decomposition, to the processor(s) and memory of the physical computer.

Each *distribution-spec* in a DISTRIBUTE directive specifies an association between points of a decomposition and processors of the machine. Each *axis-descriptor-list* must contain the same number of *axis-descriptor*s as the number of dimensions of the decomposition whose distribution is being specified. The BLOCK, CYCLIC, and BLOCK_CYCLIC *axis-descriptor*s specify distribution of the corresponding dimension of the decomposition across the processors of the machine. The ":" *axis-descriptor* specifies that all decomposition points differing only in their position along the corresponding dimension should be associated with the same processor.

In general, the DISTRIBUTE directive specifies the distribution of an n-dimensional decomposition onto an m-dimensional logical array of processors, where m<=n and m is determined by counting the number of *axis-descriptor*s in the specified distribution that are not ":". The m-dimensional logical array of processors is formed by selecting processors in some implementation-dependent way; however, implementors are encouraged to select the processors so that communication between processors that are neighbors in the logical array is favored.

If any *axis-descriptor* in a *distribution-spec* specifies an *aspect-expr*, then every *axis-descriptor* in that *distribution-spec* (except for those that are ":") must also specify an *aspect-expr*. In that case, the ratio between the values of the *aspect-expr*s specifies a desired aspect ratio for the logical processor array, which implementors are encouraged to make "reasonable efforts" to achieve as closely as is practical. Note that if the aspect ratio is not specified, then the implementation is free to pick a good one.

The BLOCK *axis-descriptor* divides the decomposition into contiguous blocks, assigning one block to each processor. The block size in this dimension is S = ceiling (N / P), where N is the size of this dimension of the decomposition and P is the number of processors in the dimension of the logical processor array that is assigned to this dimension of the decomposition[1].

The CYCLIC *axis-descriptor* specifies a round-robin division of the decomposition, assigning every P^{th} element to the same processor.

The BLOCK_CYCLIC *axis-descriptor* is similar to CYCLIC, but takes a *size-expr* M which determines a block size. These blocks are then assigned round-robin to processors.

4.2.4 The ALIGN Directive

alignment-directive	**is**	!HPF$ ALIGN *array-align-list* WITH *align-spec*
array-align	**is**	*array-name* [(*align-subscript-list*)]
	or	*scalar-name*
align-subscript	**is**	*align-subscript-name*
align-spec	**is**	*decomposition-name* [(*align-axis-spec-list*)]
align-axis-spec	**is**	*align-spec-expr*
	or	:
align-spec-expr	**is**	*specification-expr*

Constraint: An *align-subscript-name* cannot be used more than once in an *align-axis-spec-list*.

Constraint: Each *align-subscript-name* in an *array-align* must be a distinct simple variable name.

Constraint: An *align-spec-expr* is limited to a linear expression involving one *align-subscript-name.*

The general form of an allowable *align-spec-expr* is m*v+n, where m and n are Fortran 90 specification expressions and v is one of the *align-subscript-name*s appearing in the same ALIGN directive. (Simplified forms of this syntax that arise when m is -1, 0, or 1, or when n is 0, are also acceptable: thus, v, v+1, 2*v, v-1, and 7 are all examples of valid *align-spec-expr*s.) In the interpretation of each *align-spec-expr* in the *align-spec*, the *align-subscript-name*s appearing in that ALIGN directive locally hide any declarations of variables by the same name in whose scope the ALIGN directive appears.

This directive specifies the alignment of arrays with respect to each other by aligning them to positions in a named decomposition.

In the case where an ALIGN directive contains more than one *array-align* item in its *array-align-list*, the directive is viewed as "syntactic sugar" for a sequence of ALIGN directives, each

[1] For example, if we have a 105 element vector X and a 10 processor machine, the block size will be ceiling(105/10), or 11, and we will block X into 9 blocks of 11 elements each, and a final block of 6 elements.

with just one of the *array-align* items in its *array-align-list*. Thus an ALIGN directive of the form

```
!HPF$ ALIGN x, y, z WITH s
```

where each of x, y, and z is an *array-align* and s is an *align-spec*, is equivalent to the following sequence of ALIGN directives:

```
!HPF$ ALIGN x WITH s
!HPF$ ALIGN y WITH s
!HPF$ ALIGN z WITH s
```

Henceforth in our description, we consider only ALIGN directives containing a single *array-align.*

If the *align-spec* in an ALIGN directive is just a *decomposition-name*, with no *align-axis-spec-list*, then the *array-align* must also be a simple name, one that names an array with the same number of dimensions as the decomposition, and the ALIGN directive as a whole specifies a simple alignment of elements of the array with corresponding points in the decomposition. Thus the directive

```
!HPF$ ALIGN A WITH D2
```

is equivalent to

```
!HPF$ ALIGN A(I,J) WITH D2(I,J)
```

if D2 is a two-dimensional decomposition.

Note that the constraints rule out "diagonal" alignments such as

```
!HPF$ ALIGN V(I) WITH D2(I,I)
```

as well as "skewed" alignments such as

```
!HPF$ ALIGN A(I,J) WITH D1(I+J)
```

If an *align-axis-spec* is ":" instead of being an *align-spec-expr*, a "replicated" alignment over the corresponding dimension of the decomposition is specified. An entity for which a replicated alignment is specified will be represented in the machine by a replicated representation, with the number of replicas equalling the number of processors among which that dimension of the decomposition is distributed. (If replication along more than one dimension of a decomposition is specified, then the number of replicas equals the product of the numbers of processors among which each of the replicated dimensions of the decomposition is distributed.)[1]

When a write operation is performed to a replicated object, all replicas[2] of the object are updated; when a read operation is performed on a replicated object, the implementation chooses a convenient replica for performing the read. Replicated alignments are thus useful when mini-

[1]Note that replication specifications of the form m:n as in Fortran D are not allowed, because they complicate the language specification and implementation, and the need for them is not clear to us.

[2]If an implementation is incapable of handling replicated objects, it may implement only a single replica of an object even though a replicated alignment was called for, but a warning should be emitted that the specified replication was not performed.

mizing the communication caused by read operations is more important than optimizing write operations to the object.

It is illegal for an ALIGN directive to specify an alignment that would map any element of an object to a decomposition point outside the bounds of the specified decomposition. Consider the following combination of declarations:

```
!HPF$ DECOMPOSITION D(N)
      REAL, DIMENSION(N) :: A
!HPF$ ALIGN A(I) WITH D(I+1)
```

The ALIGN directive in this example is illegal because it specifies that A(N) should be mapped to the nonexistent decomposition point D(N+1). This alignment specification is illegal even if the array element A(N) is never accessed[1].

ALIGN directives can only be used to specify the alignment of "top-level" entities of the program -- they cannot be used to specify alignments of arrays or other objects inside of derived type declarations. Thus, if an alignment is specified for an object that includes a structure, all elements of the structure -- even if the structure contains arrays -- will be aligned to the same decomposition point (and hence mapped to the same processor).

ALIGN directives may not be used to specify alignments for any of the following kinds of objects:

- Objects that enter into sequence or storage association relationships, or which have not been declared to be free of such relationships (using the NO_SEQUENCE_ASSOCIATION or NO_STORAGE_ASSOCIATION directives).
- Objects appearing in any COMMON block that contains one or more objects that enter into sequence or storage association relationships. If an alignment is specified for any object named in a COMMON block, then the ALIGN directive specifying that alignment must occur in identical form in every program unit where that COMMON block is declared (and every declaration of that COMMON block must name all the same variables, in the same order, and with the same dimensions, in order to avoid exploiting storage association).
- Objects that have the TARGET attribute and therefore could be the targets of pointers. (However, note that alignments may be specified for objects with the ALLOCATABLE attribute, provided that they do not also have the TARGET attribute.)
- Assumed-size arrays.

Alignments may be specified for dummy arguments of procedures. When a procedure is invoked, if the mapping specified for a dummy argument differs from that in effect for the corresponding actual argument, the actual argument is re-mapped to conform to the mapping specified for the dummy argument before beginning execution of the procedure body. Upon return from the procedure, the argument is again re-mapped, to conform once again to the mapping in

[1]This is because we have elected not to include the Fortran D RANGE and OVERFLOW specifications. Therefore, the burden is on programmers to make their decompositions big enough to fit their arrays without overflowing or wrapping around the end of the decomposition. However, wrap-around at the machine-mapping level is still available via the CYCLIC and BLOCK_CYCLIC distributions, and implementors who wish to extend High Performance Fortran with RANGE or OVERFLOW features are welcome to do so.

effect at the invoking location. Note that this re-mapping may involve the partial or complete copying of the actual argument into new locations, and/or the partial or complete rearrangement in place of the actual argument. When the argument's original mapping is restored, data will be copied back from the locations to which it had been re-mapped to the locations where it was originally kept. All of these transformations are legal according to the restrictions stated in Section 12.5.2.9 of the Fortran 90 standard[1]; therefore, programmers who may have grown accustomed to implementations not requiring full adherence to these restrictions should take note.[2]

If a mapping is specified for a dummy argument but not for the corresponding actual argument, then the argument must be re-mapped (if necessary) on procedure entry to conform to the mapping specified for the dummy argument, but the implementation is free to choose whether to re-map it upon procedure exit. If a mapping is specified for an actual argument but not for the corresponding dummy argument, then the argument may or may not be re-mapped at procedure entry, but it must be re-mapped (if necessary) at procedure exit, to conform to the mapping specified at the invoking location. If neither the actual nor the dummy argument has a specified mapping, then the implementation is free (as always) to choose whether and when to re-map the argument.

If a procedure specifies a mapping for any of its dummy arguments, then the declaration of the procedure, including the specification of the mapping(s), must be visible at the point of every invocation of that procedure (either because the procedure is an internal procedure in the same program unit as the invocation, or through suitable use of interface blocks). If a procedure is invoked from a place where the declaration of that procedure (including applicable mapping specifications) is not visible, then that procedure may not specify mappings for any of its dummy arguments.

4.3 Association Directives

The only association directives proposed for High Performance Fortran are those required to provide control over sequence and storage association.

association-directive	**is**	*sequence-assoc-directive*
	or	*storage-assoc-directive*

[1] **12.5.2.9 Restrictions on entities associated with dummy arguments**
While an entity is associated with a dummy argument, the following restrictions hold:
(1) No action may be taken that affects the value or availability of the entity or any part of it, except through the dummy argument. . . .
(2) If any part of the entity is defined through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument. . . .

[2] Of course, specific implementations of High Performance Fortran are free to relax these restrictions. But the substantial performance cost in some cases of relaxing these restrictions dissuades us from specifying any relaxation of these restrictions as a mandatory part of High Performance Fortran. In general, relaxing these restrictions would require run-time interpretation of "dope vectors" describing the object's current mapping whenever performing any reference to an object that might be re-mapped as a result of a procedure invocation.

4.3.1 SEQUENCE_ASSOCIATION Directive

Two directives are provided for making assertions about whether specific entities may, or may not, enter into sequence association relationships. Their syntax is

sequence-assoc-directive	**is**	!HPF$ SEQUENCE_ASSOCIATION [*entity-list*]
	or	!HPF$ NO_SEQUENCE_ASSOCIATION [*entity-list*]
entity	**is**	*name*
	or	/ *name* /

Constraint: In the second form, *name* must be the name of a common block.

By default, it is assumed that entities may enter into sequence association relationships, unless it can be proven by inspection of the structure of the program that they do not. A program may assert that designated entities do not enter into sequence association relationships by listing the names of those entities in the name-list of a NO_SEQUENCE_ASSOCIATION directive. Alternatively, if the name of a COMMON block appears in the name-list, the effect of the directive applies to all entities in that COMMON block. Although sequence association is the default assumption, a program can explicitly assert that an entity may enter into sequence association relationships by naming it or a COMMON block containing it in the name-list of a SEQUENCE_ASSOCIATION directive.

If no name-list is given, the NO_SEQUENCE_ASSOCIATION directive declares that, until further notice, the default assumption about entities whose first declaration appears after the NO_SEQUENCE_ASSOCIATION directive is that they do not enter into sequence association relationships. This default assumption can be reversed in the case of specific entities by naming them in the name-list of a SEQUENCE_ASSOCIATION directive, or can be reversed generally by giving a SEQUENCE_ASSOCIATION directive with no name-list, which will restore the original default assumption that, until further notice, entities whose first declaration appears after that point may enter into sequence association relationships.

4.3.2 STORAGE_ASSOCIATION Directive

Directives are also provided for making assertions about whether specific entities may, or may not, enter into storage association relationships. Their syntax is

storage-assoc-directive	**is**	!HPF$ STORAGE_ASSOCIATION [*entity-list*]
	or	!HPF$ NO_STORAGE_ASSOCIATION [*entity-list*]

Their meanings are entirely analogous to the sequence-association directives described above, except that they pertain to storage association rather than to sequence association.

Chapter 5 - Parallel Array Assignment - FORALL

Experience with the array operations of Fortran 90 in extended Fortran 77 compilers has demonstrated the limitations of the array-section syntax and semantics (colon-separated subscript "triples") for many applications. This has led to the FORALL statement extension found in such compilers. Recent review of proposed High Performance Fortran features has suggested that a "block" form of the FORALL would also be useful. There are a number of proposals for iterated, parallel-semantic, tightly- and loosely-synchronized constructs currently under discussion among the designers and reviewers of Fortran D.

High Performance Fortran supports a simple FORALL statement that is a subset of all widely known proposals for a FORALL statement. Indeed, High Performance Fortran's FORALL statement is probably equal to the intersection of all those proposals. So we can adopt at least this FORALL statement now, and postpone specifying extensions of it until there is greater consensus in the community on how it should be extended.

In addition to the single-statement FORALL, High Performance Fortran contains a limited block-structured FORALL, the *forall-construct.* In this initial definition, the semantic facilities provided by the forall-construct are identical with those of the single-statement FORALL. In conformance with all Fortran 90 block-structured constructs, the forall-construct syntax is headed by a forall-construct-stmt and is terminated by an end-forall-stmt. The current limitation in High Performance Fortran is that the body of the forall-construct consists of a single executable assignment statement of limited form, the forall-assignment. (The forall-assignment is also the syntactic form for the single statement body of the forall-stmt.) The semantics of the forall-construct are identical to the semantics of the forall-stmt.

The rationale for providing the forall-construct in addition to the forall-stmt is to standardize the outer syntax of the block structured construct. The full semantic redundancy of the currently limited forall-construct with respect to the forall-stmt will permit further discussion of the evolution of High Performance Fortran in this area.

A parallel array assignment can be specified in terms of array elements or array sections using a FORALL statement or a FORALL construct, either of which can optionally have a mask. The FORALL statement has the syntax

forall-stmt	**is**	FORALL (*triplet-spec-list* [, *mask-expr*])
		forall-assignment-stmt

the FORALL construct has the syntax

forall-construct	**is**	FORALL (*triplet-spec-list* [, *mask-expr*])
		forall-assignment-stmt
		END FORALL

where

triplet-spec	**is**	*subscript-name* = *subscript* : [*subscript* [: *stride*]]

Constraint: The *subscript-name*s must be scalar, must be of type integer, and must be distinct. This name is valid only within the scope of the FORALL statement. The stride cannot be zero. If omitted, the stride has a default value of 1. A bound or stride cannot contain a reference to a subscript-name.

mask-expr **is** *scalar-logical-expr*

Note: The *mask-expr* can reference the *subscript-name*(s)

forall-assignment-stmt **is** *array-element* = *expr*

Constraint: The *forall-assignment-stmt* must not contain a character expression. The array-element must reference all subscript-name(s) included in the *triplet-spec*(s).

If any of the *triplet-spec*s specifies an empty range of FORALL indices, then the *forall-assignment-stmt* is not executed. If the *mask-expr* is false for all specified values of all the FORALL indices, then the *forall-assignment-stmt* is not executed.

If a user-defined function occurs on the right hand side of a *forall-assignment-stmt*, then it will be executed once for each valid set of FORALL indices for which the *mask-expr* is true, even if the FORALL indices do not occur in the arguments to the function.

The *forall-assignment-stmt* must not cause the value of any *subscript-name* to become redefined during the execution of the *forall-stmt* or *forall-construct*.

When the FORALL statement or FORALL construct is executed, the bounds and strides are evaluated first. Then the mask, the right-hand expression of the *forall-assignment-stmt,* and the left-hand expression of the *forall-assignment-stmt* are evaluated, in that order. This way the mask can guard both the expression and the assignment, and all values of the expression are evaluated before any assignments are made to left-hand values. The expression, the assignment, and the mask (if any) are evaluated over all possible combinations of the *subscript-name*(s) values; the expression evaluation and the assignment are further restricted to those combinations where the mask, if present, is TRUE. It is illegal for a FORALL statement or a FORALL construct to attempt to write any memory location more than once.

The following examples show valid element array assignments:

```
FORALL(I=1:N,J=1:N) H(I,J) = 1.0 / REAL(I+J-1)

FORALL(I=1:N,J=1:N, A(I,J) .NE. 0.0) H(I,J)=1.0/A(I,J)
```

The FORALL construct is provided both as a convenience to help avoid continuation lines caused by lengthy FORALL statements and as a syntactic “hook” to support extensions of High Performance Fortran featuring FORALL loops with multiple statements in their bodies.

Appendix - Bibliography

Standards

ISO. *Fortran 90*, May 1991. [ISO/IEC] 1539: 1991 (E).

US Department of Defense. Military Standard, *MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978*, November 9, 1978.

X3H5 Technical Committee. *Parallel Extensions for Fortran 77, X3H5 Language Binding*, July 1991. [X3H5/91-0040-C].

American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, approved April 3, 1978.

Vendors

Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp 12000 - DECmpp Parallel Fortran Reference Manual*, December 1991.

MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. *MasPar Fortran Reference Manual*, May 1991. [Software Version 1.1, 9303-0000, Rev. A2].

Digital Equipment Corporation, Maynard, Massachusetts. *DEC Fortran Language Reference Manua*l, March 1991. [AA-PE1BA-TK].

Digital Equipment Corporation, Maynard, Massachusetts. *VAX FORTRAN Language Reference Manual*, June 1988. [AA-D034E-TE].

Fortran D

Min-You Wu and Geoffrey Fox. Fortran 90D compiler for distributed memory MIMD parallel computers. Report SCCS-88b, Syracuse Center for Computational Science, Syracuse University, Syracuse, New York, July 1991.

Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D Language Specification. Report COMP TR90-141 (Rice) and SCCS-42c (Syracuse), Department of Computer Science, Rice University, Houston, Texas, and Syracuse Center for Computational Science, Syracuse University, Syracuse, New York, April 1991.