High Performance Fortran

David B. Loveman
Digital Equipment Corporation

After an intensive one-year effort, the High Performance Fortran Forum has written a language specification that improves the performance and usability of Fortran-90 for computationally intensive applications on a wide variety of machines.

Since its introduction more than three decades ago, Fortran has been the programming language of choice for difficult scientific and engineering problems on sequential computers. The recent Fortran-90 standard "modernized" Fortran-77, but exploiting the full capabilities of distributed-memory parallel computers still requires more information than either language can provide (such as how to allocate data among processors and where to place data in a single processor).

A coalition of computer vendors, government laboratories, and academic groups founded the High Performance Fortran Forum in 1992 to improve the performance and usability of Fortran-90 for computationally intensive applications on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors. After an intensive one-year effort, the group has reached a consensus and written a language specification, and a number of vendors have announced plans for compilers supporting the language. The HPF language includes the full Fortran-90 standard plus HPF extensions; the specification also includes a minimum HPF subset and identifies work yet to be done.

This article presents Fortran-90, its basis in Fortran-77, its implications for parallel machines, and the HPF extensions. Sidebars describe SIMD and MIMD systems, previous attempts to develop languages for them, the genesis of the HPF Forum, how the group actually worked, and the HPF programming model.
**Fortran-90**

Fortran-90 is a new ISO and ANSI standard that includes as a subset all of Fortran-77 and MIL-STD-1753 for compatibility. (The Fortran standards committee uses "FORTRAN 77" to refer to the Fortran-77 standard language, "Fortran 90" to refer to the Fortran-90 standard language, and "Fortran" when a particular standard or product is not being referred to. The style of this magazine does not allow for the all-capital spelling, but my use of "Fortran-77" and "Fortran-90" throughout should alleviate any confusion.)

**Fortran-77**

Many of Fortran-77's language features have become well known to engineering and scientific programmers: array reference and arithmetic computation notation, iteration loops (Do), conditional statements (If ... Then ... Else If ... End If), subroutines and functions, global variables (Common), independent compilation of program units, complex numbers, character data type and operations, and formatted, unformatted, and direct-access file input and output.

Fortran-77 also provides direct visibility to a particular hardware architecture feature: a linear, one-dimensional, sequentially addressed memory. This feature is visible in Fortran-77 via

- Sequence association: the mapping of multidimensional arrays to a linear sequence ordering ("column-major order").
- Storage association: the mapping of Fortran data objects to underlying storage units.

Fortran-77 uses sequence and storage association in such language features as assumed size arrays, Common reshaping, Equivalence reshaping, and procedure argument reshaping. Whenever an architecture is directly visible in a programming language, we can expect good performance on that architecture and difficulty in porting applications to other computer architectures. Not surprisingly, Fortran-77 implementations tend to execute applications well on traditional CISC and RISC linear-memory architectures, but they are inadequate for distributed-memory architectures.

**MIL-STD-1753**

In 1978 the US Department of Defense published an addendum to the Fortran-77 standard with language features required by all compilers to be sold to the US government. Virtually every Fortran compiler supports these features: End Do statement, Do While statement, Include statement, Implicit None statement, an input/output semantic extension, 11 bit-manipulation procedures, and syntax for octal and hexadecimal constants.

---

**Soon after the Fortran-77 standard was published, work began on a revision to "modernize Fortran, so that it may continue its long history as a scientific and engineering programming language."**

**Fortran-90**

Soon after the Fortran-77 standard was published, work began on a revision with the working title of Fortran-8x. The objective was (in the words of the Fortran-90 standard) to "modernize Fortran, so that it may continue its long history as a scientific and engineering programming language." The Fortran-8x efforts were intended to provide modern language features that would let programmers stop using obsolescent and "politically incorrect" forms in Fortran-77, some of which are specifically identified in the standard as "decremental features": arithmetic If statements; real and double-precision Do loop control variables and expressions; shared Do termination and continuation on a statement other than End Do or Continue; branching to an End If from outside the If statement; alternate return; Pause and Stop statements; Assign and assigned Go To statements; assigned Format specifiers; and cH edit descriptor.

Fortran-90 is a significantly larger language than Fortran-77, and in many ways it is a technically difficult language to implement. Features that improve on Fortran-77 include array operations, numerical computation facilities, parameterized intrinsic data types, user-defined data types, modularization of data and procedures, additional storage classes of objects, and additional intrinsic procedures.

**Array operations**

Fortran-90 contains new syntax to allow numeric oper-
Programming for distributed-memory parallel computers

Distributed-memory parallel computers include multiple-instruction, multiple-data (MIMD) machines such as Intel’s Paragon XP/S, single-instruction, multiple-data (SIMD) machines such as the Thinking Machines CM-2, MasPar’s MP-1 and MP-2, and Digital’s DECmpp 12000/5x; and farms of workstations such as Digital’s Alpha AXP. Performance on all types of distributed-memory parallel computers can be boosted by keeping data local, which decreases costly interprocessor communication.

Distributed-memory MIMD systems must be programmed at quite a low level. Typically a developer must explicitly distribute data to the various processors, implement code that operates on local data on a per-processor basis, and provide for explicit message passing or other communication to access nonlocal data.

Programming for SIMD computers is at a somewhat higher level. The developer programs in a subset of Fortran-90, using Fortran-90 array operations to provide implicit parallelism. (The developer actually uses a subset of Fortran-8x, Fortran-90’s predecessor, which in addition to the array notation also included a statement that was eliminated as part of the Fortran-90 standardization process but added back by HPf.) Advanced compilers map the program to the underlying machine. For performance, the developer often must provide data locality information in the form of directives, which are vendor-specific although similar in style and intention.

In the early 1970s, the compiler developer Compass implemented language constructs for data placement and layout in its IVtran (pronounced “four-tran”) compiler for the Illiac IV SIMD computer. In the 1980s, Compass and Thinking Machines developed static layout directives for a subset of Fortran-8x for the Connection Machine. Compass’s technology was also used in Fortran compilers for MasPar and Digital Equipment Corporation.

Rice and Syracuse Universities have conducted research for several years on Fortran D, a language designed to support a data parallel programming style, providing linguistic support for data decomposition at two levels of parallelism: the problem mapping (the natural fine-grained parallelism defined by individual members of data arrays, independent of machine considerations) and the machine mapping (the coarse-grained parallelism defined by the physical parallel machine). They were specifically trying to define language constructs similar to those provided by SIMD vendors, but in a way that was neither vendor- nor SIMD-specific. Several other institutions have conducted research on data locality issues in Fortran, including the University of Vienna (Vienna Fortran), Syracuse University (Fortran-90 D), ICASE (Kali), and Yale University (the Yale extensions).

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.

Numerical computation facilities

Fortran-90 includes portability control over numeric precision specification, and provides intrinsic functions that let programmers determine the characteristics of numeric representation on a particular machine while a program is running.

Early in 1991, Digital and Compass began a project to develop an industrywide Fortran dialect that could support an array-based, data-parallel programming style on both SIMD and MIMD massively parallel machines, as well as more conventional architectures. Digital felt that a standard language would make application development for parallel computers easier, thus increasing the size of the market to the benefit of all vendors, including Digital.

The major objectives of this project — called High Performance Fortran — included hardware architecture independence, comprehensive language features, support for language standards, pragmatic selection of language features, staged implementation possibilities, performance-tuning directives defined in the language, and industry and academic consensus on the primary language features.

The language was to be based on Fortran-90, Fortran D and SIMD Fortran experience (especially ForAll), except that the Fortran D data decomposition statements were replaced by directives in the form of structured comments. Project efforts included a preliminary language design, multiple one-on-one presentations to other vendors and major users, and a “birds of a feather” session at Supercomputing ’91, where Digital agreed to cooperate with Rice University to set up the High Performance Fortran Forum.

operations on entire arrays without explicit Do loops: We can now say A = B + C to add two arrays together and store them in a third array. These features were introduced because many scientists find them a natural and readable way of expressing algorithms. They have also proven to have efficient implementations on a variety of computer architectures. Indeed, some of these facilities are already supported in product compilers. The introductory overview in the Fortran-90 standard states:

February 1993 27
Parameterized intrinsic data types
This feature lets vendors support additional data types, such as short integers, very large character sets (such as those used in China and Japan), more than two precisions for Real and Complex types, and packed logical.

User-defined data types
These let a developer define arbitrary data structures and the operations that act on them.

Modularization of data and procedures
Many developers consider this one of Fortran-90’s best features. It is similar to Ada encapsulation. The basic idea is that related data declarations, derived type definitions, and procedures reside in one place — a Module program unit — the contents of which can be accessed by any program unit. This gives the Fortran developer an effective method for defining global data, global procedures, and encapsulated data abstractions. Modules are an improvement over Fortran-77’s Entry statement because they provide an easy way to tie data and procedures together. Fortran-90’s module facilities allow design implementation using data abstractions, and they 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, interface blocks let a programmer explicitly specify interfaces to subprograms, which a high-quality compiler can use to provide better checking and optimization at the interface to other subprograms.

Additional storage classes of objects
Fortran-90’s new storage classes (such as allocatable, automatic, and assumed-shape objects) and pointer facility add significantly to those of Fortran-77 and should reduce the use of Fortran-77 constructs that depend on a linear memory model, such as Equivalenced array objects, Common definitions with nonidentical array definitions across subprograms, and actual-/dummy-argument array shape transformations. The standards committee added pointers to Fortran-90, but it chose not to use the Cray pointer implementation that many Fortran vendors have adopted; Fortran-90 pointers have many of the same capabilities as Cray pointers, but with a different syntax and an architecture-independent semantics.

Additional intrinsic procedures
Fortran-90 defines a large number of new intrinsic procedures. Many of these support mathematical operations on arrays, including the construction and transformation of arrays. Also, there are numerical accuracy procedures to support numerical programming, and bit-manipulation procedures derived from MIL-STD-1753. Examples include Any, All, Bit_Size, Ceiling, Date_And_Time, Floor, Epsilon, MaxVal, MinVal, Range, and Sun.

Miscellaneous improvements
Fortran-90 also allows 31-character names, the use of "_" in names, end-of-line comments, a free source form, a larger character set including lowercase letters and operators such as "<," new control constructs such as the Case statement, and internal subprocedures.

**Fortran-90’s features for array calculation and dynamic storage allocation make it a natural base for HPF, which adds new directives and language constructs and imposes some restrictions.**

**FORTRAN-90 ON PARALLEL MACHINES**
Although the semantics of Fortran-90 is defined without reference to a particular underlying machine model, and can be viewed as providing a global name space and a single thread of control, efficient execution can be realized on parallel machines. Consider the context provided by these Fortran-90 declarations:

Real :: S ! a scalar floating-point variable
Real, Dimension (N) :: A, B ! two N-element arrays
Integer :: I, J ! two scalar integer variables
Integer, Dimension (N) :: P ! an integer index array

Fortran-90 provides for element-by-element operations on entire arrays, where the language does not specify the order of evaluation. The semantics of Fortran-90 allows these statements to be executed in parallel. The following array assignment statement multiplies each element of B by itself, adds that value to the square root

---

**IEEE Parallel & Distributed Technology**
The High Performance Fortran Forum

More than 120 people attended the first HPF Forum meeting in January 1992 and developed the basic working procedures. The group considered extending Fortran in several areas: data distribution, parallel statements, extended intrinsic functions and standard library, extrinsic procedures, parallel I/O statements, and changes in sequence and storage association.

The members agreed that they could not solve all possible problems; instead, it was understood that the group was "trying to agree on what it agrees on," and trying to raise the level of commonality. Thus the group sought to develop a technical report of consensus, not a formal language standard. The HPF Forum determined to complete the document within a year, with meetings every six weeks and extensive correspondence by electronic mail. The group also encouraged input from the high-performance computing community through widely distributed language drafts; the electronic-mailing list included more than 500 names.

The members agreed to devote as little as possible from other standards, minimizing (and avoiding, if possible) direct conflicts with Fortran-77 and Fortran-90. The group also wanted to make sure that compilers could be developed quickly, so it decided to define a HPF subset with the minimum essential Fortran-90 and HPF features. Taken as a whole, these goals were deemed quite aggressive when they were adopted in March 1992, and led to a number of compromises in the final language.

After the first meeting there were nine working-group meetings to specify the language. The group was led by Ken Kennedy of Rice University. There were never fewer than 30 participants, so to keep the process manageable, each major topic was addressed and debated at length by a subgroup, which reported its deliberations to the entire working group for review. After additional discussion, the subgroup would prepare one or more proposals for an initial formal vote. After a second formal vote a month later, the decision became part of HPF, subject to further review later in the process.

One person represented each organization, which agreed to support that person's attendance for the year. Representatives from Convex Computer, Cray Research, Digital Equipment Corporation, IBM, Rice University, Syracuse University, Thinking Machines, and the University of Vienna gave presentations at the first meeting, and many other organizations were represented at two or more meetings:

- permutation, when array section notation or index vectors are used:
  \[ A(I:J) = B(J:1) \]
  \[ A(P) = B \quad \text{forall } i = 1:N \]
- reduction, such as summing all of the elements of an array:
  \[ S = \text{Sum}(B) \]

High Performance Fortran

Fortran-90's features for array calculation and dynamic storage allocation make it a natural base for HPF, which extends the language by adding new directives and language constructs and by imposing some restrictions.

The HPF directives are structured comments that
The HPF programming model

HPF helps developers write parallel programs for distributed-memory systems almost as easily as writing sequential programs. The developer writes the program in Fortran-90 in a single-program, multiple-data (SPMD), data parallel style, where conceptually the code has a single thread of control and a global address space. The developer provides information about desired data locality or distribution by annotating the code with HPF data-mapping directives, and then runs it through an architecture-specific compiler:

- For MIMD machines, the result is a multithreaded message passing implementation with local data and compiler-generated synchronization and send/receive code.
- For SIMD machines, the result is single-threaded parallel code with communication optimized by compiler placement of data.

suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation, but they do not change the value computed by the program; as long as the directives are used correctly, the program should generate the same results whether the directives are processed or not.

HPF directives are consistent with Fortran-90 syntax. If HPF were to be adopted as part of a future Fortran standard, the only change necessary to include a directive as a full statement would be to remove the HPF comment prefix: !HPF$.

A few new language features were made first-class constructs rather than comments because they can affect a program's interpretation (for example, by returning a value used in an expression). These are direct extensions to the Fortran-90 syntax and interpretation.

The new directives, language constructs, and restrictions give Fortran-90 new data-distribution features, statements allowing the explicit expression of parallel computation, extended intrinsic functions and standard library routines, extrinsic procedures, and changes in sequence and storage association.

Data distribution
Parallel and sequential architectures attain their fastest speed when the data they access is local, but the sequential storage order defined in Fortran-77 and Fortran-90 often conflicts with this need.

HPF includes a distribution model and directives that let the user tell the compiler how to allocate data objects to processor memories. The compiler interprets these annotations to improve data storage allocation, minimizing communication while retaining parallelism (subject to the constraint that, semantically, every data object has only a single value at any point in the program).

The model is a two-level mapping of data objects to abstract processors (see Figure 1). Data objects, typically array elements, are first aligned relative to one another; a group of arrays is then distributed on a rectilinear arrangement of abstract processors. The implementation uses the same number of physical processors (or perhaps fewer) to implement these abstract processors. This mapping of abstract to physical processors is language processor-dependent.

The premise is that an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor, and that it might be possible to carry out many such operations concurrently if they can be performed on different processors.

A number of Fortran-90 features (such as array syntax) make it easier for a compiler to determine whether many operations can be carried out concurrently. The
HPF directives provide a way to tell the compiler to ensure that certain data objects reside in the same processor. If two data objects are mapped (via alignment and distribution) to the same abstract processor, that is a strong recommendation that they ought to reside in the same physical processor. The directives also provide a way to recommend that a data object be stored in multiple locations, which can make updating the object more complicated but makes it faster for multiple processors to read the object.

There is a clear separation between directives that serve as specification statements and those that serve as executable statements. Specification statements are carried out on entry to a program unit, pretty much as if all at once; only then are executable statements carried out. (While it is often convenient to think of specification statements as being handled at compile time, some of them may contain expressions that depend on runtime quantities. The values of these expressions may not be known until runtime when program control enters the scoping unit.)

Every array (indeed, every object) is created with some distribution onto some arrangement of processors. If the specification statements contain explicit directives specifying the alignment of array A with respect to array B, then the distribution of A will be dictated by the distribution of B; otherwise, the distribution of A itself may be specified explicitly. In either case, any such explicit declarative information is used when the array is created.

An allocatable object is created whenever it is allocated. Specification directives for allocatable objects (and allocated pointer targets) may appear in the declaration part of a program unit, but take effect each time the array is created, rather than on entry to the scoping unit. If object A is aligned (statically or dynamically) with object B, which in turn is already aligned with object C, this is regarded as a direct alignment of A with C, with B serving only as an intermediary at the time of specification. This matters when B is subsequently realigned; the result is that A remains aligned with C.

Every object is created as if according to some complete set of specification directives; if the program does not include complete specifications for mapping an object, the compiler provides defaults. By default, an object is not aligned with any other object. The default distribution is language processor-dependent, but must be expressible as explicit directives for that implementation. Identically declared objects need not be given identical default distribution specifications; the compiler may, for example, take into account the contexts in which objects are used in executable code. The programmer can force identically declared objects to have identical distributions by specifying such distributions explicitly. Identically declared processor arrangements are guaranteed to represent "the same processors arranged the same way."

An object can be realigned or redistributed. Redistributing an object causes all objects aligned with it also to be redistributed to maintain the alignment relationships.

Alignment is considered an attribute (in the Fortran-90 sense) of an array or scalar. Distribution is technically an attribute of the index space of the array. Sometimes we might speak loosely of the distribution of an array, but this really means the distribution of the index space of the array, or of another array to which it is aligned. The relationship of an array to a processor arrangement is properly called the mapping of the array.

Sometimes we want to consider a large index space with which several smaller arrays are to be aligned, but not to declare any array that spans the entire index space. HPF provides the notion of a Template, which is like an array whose elements have no content and therefore occupy no storage; it is merely an abstract index space that can be distributed and with which arrays may be aligned. (Guy Steele notes, "If an array is a cat, then a Template is a Cheshire cat, and the index space is the grin.")

HPF's Dynamic attribute is much like Fortran-90's Allocatable attribute: An array that has not been declared as Dynamic cannot be realigned, and an array or template that has not been declared as Dynamic cannot be redistributed.
The Processors directive and processor views
The Processors directive declares rectilinear processor arrangements, specifying their name, rank (number of dimensions), and size in each dimension. The intrinsic functions Number_Of_Processors and Processors Shape inquire about the total number of actual physical processors used to execute the program. This information can be used to calculate sizes for the abstract processors arrangements. For example:

\[ \text{Processors } P(N) \]
\[ \text{Processors } 0/\text{Number}_\text{Of}_\text{Processors}(), \& \]
\[ \text{Processors } 1/(8, \text{Number}_\text{Of}_\text{Processors}()/8) \]
\[ \text{Processors ScalarProc} \]
\[ \text{a scalar-processors arrangement} \]

A scalar-processors arrangement can be used to indicate that certain scalar data should be kept together but need not interact strongly with distributed data. Data distributed onto such a arrangement can reside in a single "control" or "host" processor (if the machine has one), in an arbitrarily chosen processor, or it can be replicated over all processors. For computers that have a set of computational processors and a separate scalar host computer, a natural implementation is to map every scalar-processors arrangement onto the host processor. For computers without a separate scalar host computer, data mapped to a scalar-processors arrangement might be mapped to some arbitrarily chosen computational processor or replicated onto all computational processors.

A specific HPF implementation can provide a way to specify at compile time the number of physical processors on which the program is to be executed. This might be either by a language processor-dependent directive or through a command-line argument. Such facilities are beyond the scope of HPF. It might also be desirable for the user to have a way to specify the precise mapping of the processor arrangement declared in a Processors statement to the physical processors of the executing hardware. Again, such facilities are beyond the scope of the HPF specification.

The View attribute provides a way to allow the same set of abstract processors to be viewed as having different rectilinear geometries, possibly of differing rank. This feature is sometimes loosely called "Equivalence for processors arrangements."

In the following example, the processor arrays \( P, Q \), and \( R \) are equivalenced to designate the same set of 8,192 abstract processors. The first View specifies that \( P \) and \( Q \) are the same processor set, viewing \( P \) in column-major order. The second View specifies that \( P \) and \( R \) are the same processor set, viewing \( P \) in column-major order after permuting its dimensions.

\[ \text{HPF} \text{Processors } P(128,64), Q(8192), R(8192) \]
\[ \text{HPF View of } P :: Q \]
\[ \text{HPF View of } P(2,1) :: R \]

The Align directive
The Align directive specifies that certain data objects are to be distributed in the same way as certain other data objects. Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned, since two aligned objects will necessarily be mapped to the same abstract processor. The Align directive is designed to make it particularly easy to specify mappings for all the elements of an array at once.

The Align directive appears in the declaration part of a scoping unit and can specify alignment of an array with another array or with a template. HPF provides a variety of syntactic forms for specifying simple alignments, offsets, axis collapse, axis transposition, axis reversal, and replication:

\[ \text{HPF Align } X(:, :) \text{ with } D2(:, :) \] simple case \]
\[ \text{HPF Align } A(1) \text{ with } B(1+2) \] offset \]
\[ \text{HPF Align } X(:, :) \text{ with } D1(:, :) \] collapse second axis of \( X \) \]
\[ \text{HPF Align } X(J, K) \text{ with } D2(K, J) \] transpose two axes \]
\[ \text{HPF Align } X(J, K) \text{ with } D2(M+1, N-K+1) \] reverse both axes \]
\[ \text{HPF Align } X(J, K) \text{ with } D3(J, :, K) \] replicate \( X \) along second axis of \( D3 \) \]
\[ \text{HPF Align } A(1) \text{ with } D(:, :) \] align a copy of \( A \) with every column of \( D \)

Replication allows an optimizing compiler to arrange to read whichever copy is closest. (Of course, when an element of \( A \) is written, all copies must be updated, not just one. Replicated representations are useful for such tricks as small lookup tables, where it is much faster to have a copy in each physical processor but you don’t want to be bothered giving it an extra dimension that is logically unnecessary to the algorithm.)

In the following, there are \( N \) processors and we want arrays of different sizes \( (3, 4, 43) \) within each. The numbers 3, 4, and 43 may be different, because those axes will be collapsed so that array elements with indices differing only along that axis will all reside in the same processor:

\[ \text{HPF Template } D1(N) \]
\[ \text{Real } A(3, N), B(4, N), C(43, N) \]
\[ \text{HPF Align } (\cdot, :) \text{ with } D1 :: A, B, C \]

IEEE Parallel & Distributed Technology
Alignment of dummy and actual arguments at a sub-program interface is particularly important. HPF provides a number of alternatives, including two extremes in which:

- the actual argument must be realigned to match the requirements of the dummy, and then realigned again on return, or
- the caller and callee agree on alignment and no action is required.

There is also an Inherit directive, specifying that a dummy inherits a copy of the index space of the actual argument.

The Distribute directive
The Distribute directive specifies a mapping of data objects to abstract processors in a processor arrangement. While Align directives let a programmer indicate logical relationships among the most closely associated array elements in a computation, Distribute directives let the programmer specify a distribution of data points onto the hardware to get a good tradeoff between load balance (which generally favors spreading neighboring data points across many processors) and communication costs (which might favor keeping neighboring data 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, and so are the most likely statements (along with Processors directives) 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 mapping of individual arrays. Nevertheless, the vocabulary used in the Distribute directive is quite architecture-neutral, so that no changes in Distribute directives are 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.

The Distribute directive
$!HPF$ Distribute D1(Cyclic)

specifies that the machine's processors are to be considered a one-dimensional array (the implementation is encouraged to select that array so that communication between logically neighboring processors is inexpensive), and that points of the one-dimensional 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 Distribute directive
$!HPF$ Distribute D2(Block, Block)

specifies that the processors are to be considered a two-dimensional array, and the points of the two-dimensional 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 the 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 the corresponding points of D1 or D2 are mapped.

HPF provides for "*" distribution, in which all points of the indicated dimension are mapped to a single processor; Block(n) distribution, in which blocks of the specified size are distributed; and explicit distribution to a processor arrangement specified in a Processors directive.

Dynamic, Realign, and Redistribute directives
If an array is mentioned in a Dynamic directive, it might be the object of an executable Realign or Redistribute directive:
HPF Dynamic :: A, B, C

The Realign directive is similar to the Align directive but is executable; similarly, the Redistribute directive is similar to the Distribute directive but is executable. A Dynamic array or template can be realigned or redistributed at any time. Any other arrays currently aligned with an array or template when it is redistributed are also remapped to reflect the new distribution, in such a way as to preserve alignment relationships.

A Dynamic directive can be combined with other directives, with the attributes stated in any order, consistent with Fortran-90 attribute syntax. For example:

HPF Align with Sneedy, Dynamic :: X, Y, Z
HPF Distribute(Block, Block), Dynamic :: X, Y

Also, the three directives

HPF Template A(64, 64), B(64, 64), C(64, 64)
HPF Distribute(Block, Block) onto P :: A, B, C
HPF Dynamic A, B, C

can be combined into the single directive:

HPF Template, Dimension(64, 64), &
HPF Distribute(Block, Block) onto P, Dynamic :: A, B, C

Template directive

The Template directive declares templates, or abstract index spaces, specifying their name, rank (number of dimensions), and size of each dimension.

Examples are useful when several arrays must be aligned relative to one another but there is no need to declare a single array that spans the entire index space of interest. For example, we might want four $N \times N$ arrays aligned to the four corners of an index space of size $(N+1)^2$:

HPF Template, Distribute(Block, Block) :: Earth(N+1, N+1)
Real, Dimension(N, N) :: NW, NE, SW, SE
HPF Align NW(l, J) with Earth(l, J)
HPF Align NE(l, J) with Earth(l, J+1)
HPF Align SW(l, J) with Earth(l+1, J)
HPF Align SE(l, J) with Earth(l+1, J+1)

As another example, consider the requirement to replicate a rank-two array across all processors:

HPF Template, Distribute(Cyclic) :: &
HPF T(Number_Of_Processors(),)
HPF Align A(:,*) with T(*)

Templates are not passed through a subprogram argument interface; the template to which a dummy argument is aligned is always distinct from the template to which the actual argument is aligned. On exit from a subprogram, an actual argument is aligned to the same template to which it was aligned before the call.

Unlike arrays, templates cannot be in Common. Two templates declared in different scoping units will always be distinct, even if given the same name. The only way for two program units to refer to the same template is to declare the template in a module that is then used by the two program units. For example, suppose we need to map a dummy a in a subroutine, a global array b, and an array c in the main program so that their second dimensions are all within the same processor. An HPF solution using modules is:

Module Global_Subrs_And_Data
HPF Template, Dimension(10000), Distribute(Cyclic) :: d
Real, Dimension(10000, 20) :: b
HPF Align b(l, j) with d(l)
Contains
Subroutine sub1(a)
Real, Dimension(:,:) :: a
HPF Align a(l, j) with d(l)
b(:,2) = b(:,2) + a(:,1)
End
End Module Global_Subrs_And_Data

Program Caller
Use Global_Subrs_And_Data
Real, Dimension(10000, 10) :: c
HPF Align c(l, j) with d(l)
c = 2; b = 1; Call sub1(c)
End

Parallel statements

HPF provides a new statement and new directives to allow the explicit expression of parallel computation. The single- and multistatement ForAll constructs express assignments to sections of arrays; similar in many ways to array assignment, they allow more general sections to be specified. The Independent directive asserts that the statements in a particular section of code do not exhibit any sequentializing dependencies; when properly used, it does not change the semantics of the construct, but it can give the compiler more information to allow optimizations. Pure procedures are those that are sufficiently restricted (free of side effects) to be invoked within a ForAll.

The ForAll construct and related features

The ForAll construct can be viewed as an extension to the Fortran-90 array assignment and Where construct, but one that is intended to be more suggestive of local operations on each element of an array, and able to spec-
ify more general array sections than allowed by the basic array triplet notation.

The ForAll construct had its origins in the Fortran-8x definition; in the Fortran implementations of Compass, Digital, MasPar, and Thinking Machines; and in research at Rice, Syracuse, Vienna, and other universities. Precursors to the ForAll statement can be found as early as the Do ForAll statement in the Illiac IV Fortran compiler.

ForAll specifies array assignments in terms of array elements or groups of array sections, optionally masked with a scalar logical expression. It is similar to array assignment statements, but more general array sections can be assigned in ForAll. The general form of a ForAll construct is an element array assignment:

\[
\text{ForAll } \langle \text{triplet}, \ldots, \text{mask} \rangle \text{ assignment}
\]

or a block ForAll:

\[
\text{ForAll } \langle \text{triplet}, \ldots, \text{mask} \rangle \\
\text{statement}
\]

End ForAll

where triplet has the general form:

\[
\text{subscript} = \text{lowerbound} : \text{upperbound} : \text{stride}
\]

and where stride is optional, where assignment is an arithmetic or pointer assignment statement, and where statement is an assignment, a Where statement, or another ForAll statement. Any procedure referenced in a ForAll construct must be a pure function, syntactically guaranteed not to have side effects.

An element array assignment ForAll executes in the following steps:

1. Evaluation in any order of the lowerbound, upperbound, and stride expressions. The set of valid combinations of subscript values is then the Cartesian product of the sets defined by these triplets.
2. Evaluation of the mask for all valid combinations of subscript values. The mask elements can be evaluated in any order. The set of active combinations of subscript values is the subset of the valid combinations for which the mask evaluates to true. If the scalar mask expression is omitted, it is as if it were present with the value true.

3. Evaluation in any order of the expressions and subscripts contained in the assignment for all active combinations of subscript values.
4. Assignment of these values to the corresponding elements of the array on the left-hand side. The assignments can be made in any order. An assignment in a ForAll construct must not cause more than one value to be assigned to any array element.

Since a function called from a ForAll construct must be pure, it is impossible for that function's evaluation to affect other expressions' evaluations, either for the same combination of subscript values or for different combinations.

A block ForAll is roughly the same as replicating the ForAll header in front of each statement in the block, except that any expressions in the ForAll header are evaluated only once, rather than being reevaluated before each statement in the body. The exceptions are for nested ForAll statements and Where statements.

We can think of a block ForAll as synchronizing twice per contained assignment statement: once after handling the right-hand side and other expressions, but before performing assignments; and once after all assignments have been performed but before commencing the next statement. In practice, appropriate dependence analysis will often permit the compiler to eliminate unnecessary synchronizations.

In many cases, compiler optimizations, such as copy propagation, can eliminate the requirement for temporaries for lowerbounds, upperbounds and strides. Similarly, dependence analysis can eliminate the requirement for array temporaries. Thus a ForAll statement such as

\[
\text{ForAll } (I=1:M, J=1:N) A(I, J) = I \times B(J)
\]

can be implemented on a scalar machine as

\[
\text{Do } I=1, M \\
\text{Do } J=1, N \quad \\
A(I, J) = I \times B(J) \\
\text{End Do} \\
\text{End Do}
\]

High Performance Fortran provides a new statement (ForAll) and a new directive (Independent) to allow the explicit expression of parallel computation.
On a parallel SIMD or MIMD machine it can, of course, be implemented in parallel.

Examples

Examples of element array assignments with natural Fortran-90 equivalents include:

```fortran
ForAll (I=1:N, J=1:N, A(I, J) .NE. 0.0) B(I, J) = 1.0 / A(I, J)
```

! is the same as
Where (A /= 0.0) B = 1.0 / A

! the ForAll in effect gives a name to a section triplet
ForAll (I=1:L) R(I) = S(I)

! is the same as
R(I:L) = S(I:L)

! the implied spread
R = S
! since R and S are both of length L

ForAll (I=1:10:2, J=1:10:1) A(I, J) = B(I, J) * C(I, J)

! is equivalent to
A(1:10:2, 1:10:1) = B(1:10:2, 1:10:1)

! computational use of subscript values in one dimension
ForAll (I=1:100) R(I) = I

! is equivalent to the use of an array constructor
R = (/ (I = 1:100) /

! certain cases of spreading
ForAll (I=1:10, J=1:20) A(I, J) = S(I)

! is equivalent to the implied spread
A(1:10, 1:20) = Spread(S(1:10), Dim=2, NCopies=20)

! vector valued subscripts
ForAll (I=1:L) R(V(I)) = S(I)

! is equivalent to
R(V(I:L)) = S(1:L)

! is equivalent to
R(V) = S

! permutation of two axes, array transpose
ForAll (I=1:M, J=1:N) A(I, J) = B(J, I)

! is equivalent to
A = Transpose(B)

ForAll (I=1:M, K=1:N) A(I, K) = Sum(A(:, I) * B(:, K))

! is equivalent to
A = Matmul(A, B)

Examples of element array assignments without natural Fortran-90 equivalents include:

! scatter addressing
ForAll (I=1:100, J=1:100) A(U(I), V(J)) = S(I, J)

! array representing a binary tree

! parallel prefix operations
ForAll (I=1:100) R(I) = Sum(S(1:I))

! is equivalent to use of the HPF function
R = Sum_Reduce(S)

! zeros the upper right triangle of C
ForAll (I=1:M, J=1:N, L=J) C(I, J) = 0

! assigns consecutive integers to all elements of array A
ForAll (I=1:M, J=1:N) A(I, J) = (I-1) * N + J - 1

! assigns diagonal of A to R
ForAll (I=1:M) R(I) = A(I, I)

Examples of the ForAll construct include:

ForAll (I = 2:N-1, J = 2:N-1)
A(I, J) = A(I, J-1) + A(I, J+1) + A(I-1, J) + A(I+1, J)
B(I, J) = A(I, J)
End ForAll

ForAll (I = 1:N-1, J = 1:N)
A(I, J) = A(J, I)
End ForAll
End ForAll

ForAll (I = 1:N, J = 1:N)
A(I, J) = Merge(A(:, J), A(I, :) .EQ. J)
Where (...) .Not.(Done(I, J, 1:M))
B(I, J, 1:M) = B(I, J, 1:M) .AND. X
End Where
End ForAll

Pure procedures
A pure function obeys certain syntactic constraints that ensure it produces no side effects. The only effect of a pure function reference on the state of a program is to return a result — it does not modify the values, pointer associations or data mapping of any of its arguments or global data, and it performs no I/O.

A pure subroutine produces no side effects except for modifying the values or pointer associations of certain arguments. A pure procedure (function or subroutine) can be used in any way that a normal procedure can. A procedure is required to be pure if it is used in a ForAll statement or construct, in the body of a pure procedure, or as an actual argument in a pure procedure reference.

This freedom from side effects ensures that a pure
function can be invoked concurrently in a ForAll statement without undesirable consequences such as nondeterminism. The compiler can also perform more extensive optimizations when all functions are pure.

A pure function can be invoked concurrently at each "element" of an array if it is referenced in a ForAll. In these cases, a limited form of MIMD parallelism can be obtained via branches in the pure procedure that depend on arguments associated with array elements or their subscripts (the latter especially in a ForAll context). For example:

```fortran
Function f(x, i)
    !HPF$ Pure f
    Real x ! associated with array element
    Integer i ! associated with array subscript
    If (x > 0.0) Then
        ! content-based conditional
        x = x * 2
    Else If (i=1 OR i=n) Then
        ! subscript-based conditional
        x = 0.0
    EndIf
End Function

ForAll (i=1:n) a(i) = f(a(i), i)
```

This can sometimes provide an alternative to using sequences of masked ForAlls, with their potential synchronization overhead.

The Independent directive
The Independent directive asserts to the compiler that no data object is defined by one iteration of a Do loop and used (read or written) by another; it asserts similar information about the combinations of index values in a ForAll statement. In other words, the Independent directive asserts that the operations in a Do loop or ForAll statement or construct can be executed independently — in any order, or interleaved, or concurrently — without changing the program's semantics. A compiler can rely on this information to make optimizations, such as parallelization or reorganizing communications. If the assertion is true, the semantics of the program are not changed; if it is false, the program is not standard-conforming and has no defined meaning.

The following code asserts that array P does not have any repeated entries and that A and B are not storage associated:

```fortran
!HPF$ Independent
Do i=1,100
    A(P(i)) = B(i)
End Do
```

In the next example, the inner loop is not independent because each element of A is assigned repeatedly. However, the three outer loops are independent because they access different elements of A. It is not relevant that the outer loops read the same elements from B and C, because those arrays are not assigned.

```fortran
!HPF$ Independent (i1, i2, i3)
Do i1 = 1, N1
    Do i2 = 1, N2
        Do I3 = 1, N3
            Do i4 = 1, N4
                ! The inner loop is not independent!
                A(i1,i2,i3) = A(i1,i2,i3) + B(i1,i2,i4)*C(i2,i3,i4)
            End Do
        End Do
    End Do
End Do
```

**Intrinsic functions and library routines**
Fortran-90 anticipated some, but not all, of the basic operations that are valuable for parallel algorithm design. HPF adds several classes of parallel operations as intrinsic procedures: system inquiry intrinsics, new computational intrinsics and extensions of existing intrinsics, and distribution inquiry intrinsics. HPF also defines a standard library of computational functions.

**System inquiry intrinsic functions**
We can think of the processors in a multiprocessor system as being arranged in an implementation-dependent n-dimensional processor array. The system inquiry functions return values related to this underlying machine and processor configuration, including the size and shape of the array. Number_Of_Processors returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the array. Processors:_Shape returns the shape of the array. Their values remain constant for (at least) the duration of one program execution, and can be used to specify, for example, array bounds. HPF programs can be compiled to run on machines whose configurations are not known at compile time. System inquiry functions query the physical machine, and have nothing to do with any Processors directive that may occur.

Let's consider two real computers. For a DECmpp 12000/Sx Model 200 with 8,192 processors:

February 1993
while for a single processor DEC 3000 AXP workstation:

Number Of Processors( ) == 1
Number Of Processors(Dim=1) == 1
Processors Shape( ) == (/ 1 /)

References to system inquiry functions can occur in HPF directives, as in:

!HPF$ Template T(100, 3*Number Of Processors( ))

System inquiry function calls can occur in, for example, lower or upper bounds of array declarations:

Integer, Dimension(Size(Processors Shape( ))) :: PS
PS = Processors Shape( )
! note that PS(2) == Number Of Processors(Dim=2)

The expression Size(Processors Shape( )) returns the rank of the processor array.

Computational intrinsic functions
HPF extends Fortran-90’s Maxloc and Minloc intrinsics to have an optional Dim argument that works in the same way as the Dim argument of Fortran-90’s Maxval function. If such an argument is present, then the shape of the result equals the shape of the first argument with one dimension deleted (the one indicated by the Dim argument); it is as if a series of one-dimensional Maxloc or Minloc operations were performed.

Ilen is an elemental integer-length intrinsic. Its value when applied to a scalar is:

Ilen(x) = Ceiling(log2( If x < 0 Then –x Else x+1 ))

This value is the number of bits required to store a 2’s-complement signed integer x. For example, 2**Ilen(N–1) rounds N up to a power of 2 (for N > 0), and 2**(Ilen(N)–1) rounds N down to a power of 2.

Mapping inquiry intrinsic subroutines
HPF provides a rich set of data-mapping directives. Users might need to know to what extent the compiler took their advice, especially when a user calls a non-HPF subroutine that needs to know the exact mapping. HPF includes inquiry intrinsic subroutines that describe how an array is actually mapped onto a machine. To keep the number of intrinsics small, the inquiry intrinsics are structured as intrinsic subroutines — HPF_Alignment, HPF_Template, and HPF_Distribution — with a number of optional arguments. Details of these functions are provided in the HPF specification.

Computational library functions
HPF also defines a library of computational functions to be provided as a Fortran-90 module (details are provided in the HPF specification):

- Reduction functions: AND, OR, EOR, and Parity.
- Combining scatter functions, one for each reduction function: Sum_Scatter, Count_Scatter, Product_Scatter, All_Scatter, Any_Scatter, Maxval_Scatter, Minval_Scatter, AND_Scatter, OR_Scatter, EOR_Scatter, and Parity_Scatter.
- Parallel prefix functions, one for each reduction function: Sum_Prefix, Count_Prefix, Product_Prefix, All_Prefix, Any_Prefix, Maxval_Prefix, Minval_Prefix, AND_Prefix, OR_Prefix, EOR_Prefix, and Parity_Prefix.
- Parallel suffix functions, one for each reduction function: Sum_Suffix, Count_Suffix, Product_Suffix, All_Suffix, Any_Suffix, Maxval_Suffix, Minval_Suffix, AND_Suffix, OR_Suffix, EOR_Suffix, and Parity_Suffix.
- Sorting functions: Grade_Up, and Grade_Down.
- Bit manipulation: PopCnt, PopPar, and Leadz.

EXTRANSC PROCEEDURES
Some operations are difficult or impossible to express directly in a high-level, machine-independent language. For example, many applications benefit from finely-tuned systolic communications on certain machines; HPF’s global address space does not express this well. To address such problems, HPF provides an extrinsic procedure interface as an escape mechanism for calling non-HPF code from an HPF program. Using it to call SPM array code, for example, lets the programmer descend to a lower level of abstraction to handle problems that are not efficiently addressed by HPF, and to hand-tune sophisticated kernels or low-level parallel libraries. This interface can also be used to interface HPF to other languages, such as C.

A caller uses the same semantics to invoke an extrinsic HPF procedure as for a regular procedure, but the called procedure uses a different model. A call to an extrinsic procedure from “global” HPF code results in the transfer of control on each executing physical
processor to a copy of a local procedure. All global arrays accessible to the extrinsic procedure (arrays passed as arguments) are logically carved into pieces; the copy of the local procedure executing on a particular physical processor sees an array containing just those elements of the global array that are mapped to that physical processor.

It is important not to confuse the extrinsic procedure called from the HPF program with the local procedures executed on each node: Invoking an extrinsic procedure results in a separate invocation of a local procedure on each processor. Executing an extrinsic procedure consists of concurrently executing a local procedure on each processor.

An extrinsic procedure can be defined as explicit SPMD code by specifying the local procedure code that is to execute on each processor. HPF provides a mechanism for defining local procedures in Fortran-90. Extrinsic procedures can also be defined in any other parallel language that maps to this basic SPMD execution model. The HPF compiler will compile the calling sequence for an extrinsic procedure when the local procedures are defined outside HPF and compiled separately.

With the exception of returning from a local procedure to the global caller that initiated local execution, there is no implicit synchronization of the locally executing processors. As a result, a local procedure can use any control structures whatsoever. Accessing data outside the processor requires either preparatory communication to copy data into the processor before running the local code, or communication between the separately executing copies of the local procedure. Individual implementations can provide implementation-dependent ways to communicate, such as through a message-passing library or a shared-memory mechanism. Such mechanisms are beyond the scope of HPF, but many useful portable algorithms that require only independence of control structure can take advantage of local routines, without requiring a communication facility.

The extrinsic procedure interface assumes only that array axes are mapped independently to axes of a rectangular processor grid, each array axis to at most one processor axis (no “skew” distributions) and no two array axes to the same processor axis. But the mapping of an array axis to a processor axis can be any mapping whatever. This restriction ensures that each physical processor contains a subset of array elements that can be locally arranged in a rectangular configuration. (Of course, computing the global indices of an element given its local indices, or vice versa, can be quite a tangle, but it will be possible.)

HPF specifies:

- the HPF interface to extrinsic routines, and the contract between the caller and the callee;
- a specific version of this interface for the case where extrinsic procedures are defined as explicit SPMD code, and local procedures are written in Fortran-90; and
- an extension of HPF that allows local (extended) Fortran-90 procedures to be included in an HPF program.

**Fortran-90 anticipated some, but not all, of the basic operations that are valuable for parallel algorithm design.**

For example:

```fortran
! HPF Code
Interface
  Function F(X)
    Real, Dimension(:) X, :: X
  End Function F
End Interface

! Fortran-90 local implementation of F
Function F(X)
  Real, Dimension(:) :: X
  ! X is that part of the actual argument on this processor
  ...
End Function F
```

**Sequence and Storage Association**

Fortran-77 and Fortran-90 constrain the location of data in two ways:

1. Common and Equivalence statements constrain the alignment of different data items based on an underlying model of storage units and storage sequences: a single, linearly addressed memory.
2. Sequence association specifies the order of array elements that Fortran requires when an array ex-
pression or array element is associated with a dummy array argument, a natural concept only in systems with a linearly addressed memory.

As an extension to Fortran-90, HPF allows codes that rely on sequence and storage association, although full support of these concepts is not compatible with HPF’s goal of high performance through data distribution across multiple processors. HPF therefore provides directives to assert that full sequence and storage association for affected variables must be maintained. In the absence of these inhibiting features, reliance on the properties of association is not allowed. An optimizing compiler can then choose to distribute any variables across processor memories to improve performance. To protect program correctness, a given implementation should provide a mechanism to ensure that all such default optimization decisions are consistent across an entire program.

HPF defines the following relationship between HPF data mapping and Fortran sequence and storage association:

- Common blocks are nonsequential by default.
- Variables are nonsequential by default unless they have certain sequential properties.
- A Sequence directive allows explicit declaration that a variable or a Common block is to be sequential.

Although this approach is intuitively straightforward, the actual rules in the HPF specification are rather complex.

The HPF Subset

Because full Fortran-90 compilers may not be available quickly on all platforms, and because some HPF extensions are more complex to implement than others, the HPF Forum defined an HPF subset. The group encouraged vendors to provide more rather than less in their first implementations and to move rapidly to full HPF implementations. However, HPF users who are concerned about multimachine portability may stay within this subset initially.

The subset includes those Fortran-90 features that are closely related to high performance on parallel machines:

- All of Fortran-77 (except sequence and storage association).
- The MIL-STD-1753 features: Do While statement; End Do statement; Implicit None statement; Include line; scalar bit-manipulation intrinsic procedures; binary, octal, and hexadecimal constants for use in Data statements.
- Arithmetic and logical array features: array sections (subscript triplet notation and vector-valued subscripts); array constructors limited to one level of implied Do; arithmetic and logical operations on whole arrays and array sections; array assignment; masked array assignment (Where statement and block Where . . . ElseWhere construct); array-valued external functions; Automatic arrays; Allocatable arrays and the Allocate and Deallocate statements; assumed-shape arrays
- Many intrinsic functions
- Declarations: type declaration statements (but no Kind, derived type, or Pointer), and attribute specification statements (Allocatable, Intent, Optional, Parameter, and Save)
- Procedure features: Interface blocks (but no generics or modules), optional arguments, and keyword argument passing.
- Syntax improvements: long (31-character) names, lowercase letters, use of "_" in names, and full-line and trailing "!!!" initiated comments

The subset also includes all HPF directives and language extensions except:

- The directives Realign, Redistribute, Dynamic, View, and Pure.
- The full ForAll construct (the subset does include the simple ForAll element array assignment).
- The HPF library.
- Values of the optional Dim arguments to the Fortran-90 Maxloc and Minloc intrinsic functions that are not initialization expressions.
- The Extrinsic directive, the Local directive, the definition of extrinsic procedures, and the Fortran-90 SPMD binding.

Future work

HPF considerably increases Fortran’s expressive capabilities, but a number of simplifications were made to get agreement on the de facto standard within a year. In many cases, the HPF Forum agreed that a capability was needed without reaching consensus on how to provide it (without “agreeing on what to agree on”). For example, the group agreed on the need to specify both rec-
tangular and nonrectangular arrangements of processors, but not on how to specify nonrectangular arrangements. Thus, even with the Processors directive, we can only specify rectangular arrangements.

Another example is parallel I/O. From the beginning there was a strong feeling against adding explicitly parallel I/O statements because:

- Fortran I/O is already highly expressive.
- I/O systems on different parallel computers are too architecturally different for there to be a useful abstraction on which to build a language model.
- An HPF compiler will know when it is performing I/O on distributed arrays, and can optimize the I/O to distributed files without any extensions to the source language.
- The management of distributed files (and their implementation) is a matter for the operating system, not the language.
- The current lack of extensions does not limit features that may be added by vendors.

By a narrow vote, the HPF Forum declined to extend Fortran I/O, but it included the topic in an appendix to the HPF specification — called the “Journal of Development”— that describes several features that were considered but not accepted into HPF. Many other features, including support for explicit MIMD computation, message-passing, and synchronization, were rejected or limited for lack of time or consensus rather than because of technical flaws. Other major “missing” capabilities include support for irregular and user-defined distributions, de facto extensions to Fortran such as the “*” notation for data typing, and support for MIMD computing paradigms other than data parallel. This last topic includes support “above” HPF (the ability to “tie together” multiple data parallel programs in a model we call “communicating data parallel programs”) as well as support “below” HPF (in the form of extrinsic procedures).

Other topics covered in the “Journal of Development” are nested Where statements, Allocate in ForAll, generalized data references, ForAll with Independent directives, Execute-On-Home and Local-Access directives, and ForAll-ElseForAll construct.

At its December 1992 meeting, the HPF Forum discussed the possibility of holding an HPF workshop at the Supercomputing '93 conference this November in Portland, Oregon, followed by an HPF Forum II process beginning early in 1994.

So far, actual implementations related to Fortran-90 and HPF include SIMD implementations of Fortran-90 array operations, and a Fortran-90-to-C translator. There has also been university research on Fortran-D, Fortran-90 D, and Vienna Fortran, as well as SIMD experience with data locality directives defined in terms of the underlying machine.

HPF developers are confident they can build quality compilers, but as yet there is no real experience either with first compilers or with understanding the kind of code that users will actually write. Compilers under development are implementing the HPF subset, initially avoiding the harder HPF features and Fortran-90 features beyond the array features.

Following initial implementation of Fortran-90 and of HPF, there will be a necessary tuning phase in which the vendors discover how the features are actually used and optimize the compilers accordingly. Some constructs, such as ForAll, are not hard to implement correctly, but are rather difficult to implement in parallel efficiently.

**Acknowledgments**

This article would not have been possible without the long hours put in by the members of the High Performance Fortran Forum. I specifically thank the other subgroup leaders, especially Rob Knighten, Chuck Koelbel, Bob Schreiber, Marc Snir, Con Steele, and Mary Zodel, who wrote, integrated, and edited the chapters of the HPF Specification, from which I quoted and paraphrased liberally in this article without specific attribution. (The High Performance Fortran Language Specification is copyright 1992 by Rice University, Houston, Texas.) Thanks again to Chuck Koelbel for his extensive meeting notes, and to Ken Kennedy for his leadership in organizing and running the HPF Forum process. I also thank the members of the DEC Fortran-90 MPP and Alpha teams, especially Beth Benoit and Keith Kimball for input on the Fortran-90 sections of this article, and Bert Halstead for his leadership of Digital’s original internal HPF project. The errors of omission and commission are, of course, my own.
Related works

ILLIAC IV FORTRAN

FORTRAN-77

MIL-STD 1753

FORTRAN-D

FORTRAN-8x

FORTRAN-90

VIENNA FORTRAN

KALI

HIGH PERFORMANCE FORTRAN

FORTRAN IMPLEMENTATIONS
D.M. Pase, T. MacDonald, and A. Metzner, "MPP Fortran Programming Model," tech. report, Cray Research, Eagan, Minn., 1992; available by electronic mail request to mppgpr@cray.com.

David Loveman is a senior program manager at Digital Equipment Corporation. He is also Digital's representative to the High Performance Fortran Forum and editor of the HFP Language Specification. His technical interests include programming language strategies, compiler architectures, and parallel computing. Before joining Digital, Loveman was at Compass, Inc., where as president he planned and oversaw language design and compiler development for a number of massively parallel computers. He received an MS in applied mathematics from Harvard University and a BS in electrical engineering from Princeton University. He is a member of IEEE, the IEEE Computer Society, ACM, and SIGPlan. Readers can contact him at Digital Equipment Corporation, ML01-3/B11, 146 Main St., Maynard, MA 01754; Internet, loveman@dhvax.enet.dec.com.